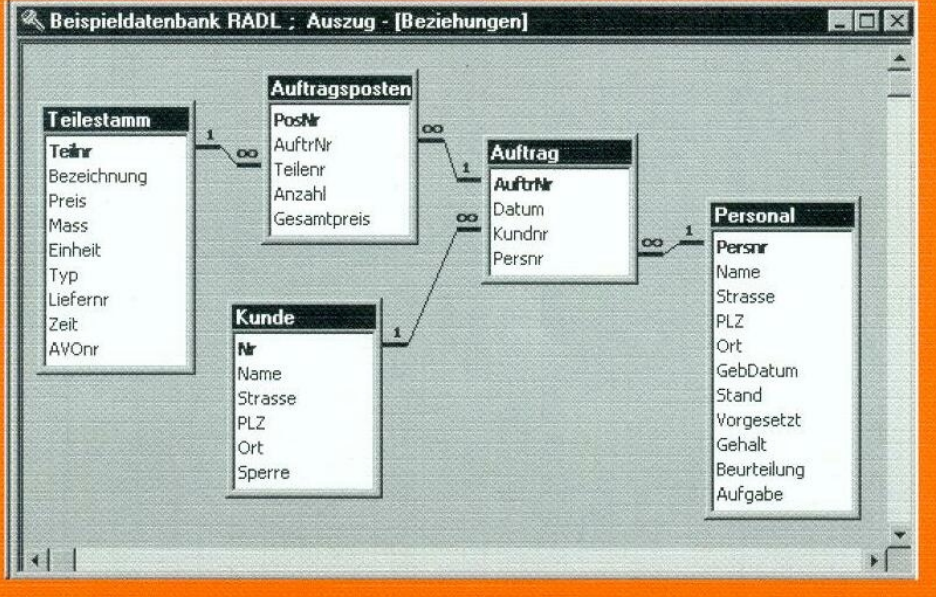
Edwin Schicker

Datenbanken und SQL

3. Auflage



B.G.Teubner Stuttgart · Leipzig · Wiesbaden

Informatik & Praxis

Herausgegeben von
Prof. Dr. Helmut Eirund, Fachhochschule Harz
Prof. Dr. Herbert Kopp, Fachhochschule Regensburg
Prof. Dr. Axel Viereck, Hochschule Bremen

Anwendungsorientiertes Informatik-Wissen ist heute in vielen Arbeitszusammenhängen nötig, um in konkreten Problemstellungen Lösungsansätze erarbeiten und umsetzen zu können. In den Ausbildungsgängen an Universitäten und vor allem an Fachhochschulen wurde dieser Entwicklung durch eine Integration von Informatik-Inhalten in sozial-, wirtschafts- und ingenieurwissenschaftliche Studiengänge und durch Bildung neuer Studiengänge – z.B. Wirtschaftsinformatik, Ingenieurinformatik oder Medieninformatik – Rechnung getragen.
Die Bände der Reihe wenden sich insbesondere an die Studierenden in diesen Studiengängen, aber auch an Studierende der Informatik, und stellen Informatik-Themen didaktisch durchdacht, anschaulich und ohne zu großen „Theorie-Ballast“ vor.
Die Bände der Reihe richten sich aber gleichermaßen an den Praktiker im Betrieb und sollen ihn in die Lage versetzen, sich selbständig in ein in seinem Arbeitszusammenhang relevantes Informatik-Thema einzuarbeiten, grundlegende Konzepte zu verstehen, geeignete Methoden anzuwenden und Werkzeuge einzusetzen, um eine seiner Problemstellung angemessene Lösung zu erreichen.

Datenbanken und SQL

Eine praxisorientierte Einführung mit Hinweisen zu Oracle und MS-Access

Von Prof. Dr. Edwin Schicker
Fachhochschule Regensburg

3., durchgesehene Auflage

B. G. Teubner Stuttgart Leipzip Wiesbaden 2000

Prof. Dr. Edwin Schicker

Geboren 1954 in Thanhausen (Oberpfalz). Von 1972 bis 1973 Studium der Elektrotechnik an der Fachhochschule Regensburg. Von 1973 bis 1979 Mathematikstudium mit Nebenfach Informatik an der Technischen Universität Berlin. Von 1980 bis 1984 wissenschaftlicher Angestellter am Fachbereich Mathematik der Universität Karlsruhe, dort 1984 Promotion zum Dr. rer. nat. Von 1985 bis 1990 Mitarbeiter bei der Siemens AG in München-Perlach in der Entwicklung des Betriebssystems BS2000. Seit 1990 Professor für Informatik an der Fachhochschule Regensburg. Spezialgebiete: Datenbanken, strukturierte und objektorientierte Programmierung und Warteschlangentheorie.

Die Deutsche Bibliothek – CIP-Einheitsausgabe
Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

3. Auflage September 2000

www.teubner.de

Gedruckt auf säurefreiem Papier
Umschlaggestaltung: Peter Pfitz, Stuttgart
Druck und buchbinderische Verarbeitung: Präzis-Druck GmbH, Karlsruhe
Printed in Germany

ISBN 3-519-22991-9

Vorwort

Datenbanken entstanden ab etwa 1960 aus der Notwendigkeit, die logischen Zugriffe auf die immer größer werdenden Datenmengen zu vereinfachen und zu normieren. Wurden diese Datenbanken über mehr als zwei Jahrzehnte hinweg ausschließlich auf Großrechnern eingesetzt, so haben sie inzwischen ihren Siegeszug auch auf kleinen Rechnern angetreten. Ermöglicht wurde dies aus dreierlei Gründen: erstens durch die enorm gestiegene Rechnerleistung der letzten Jahre, zweitens durch die Verwendung relationaler Datenbanken und drittens durch die Einführung grafischer Oberflächen.

Die Anfang der 70er Jahre entwickelten relationalen Datenbanken ermöglichen eine einfache Erstellung und Programmierung. Grafische Oberflächen unterstützen den Anwender und Datenbankdesigner dank leichter Benutzerführung und anschaulicher Musterbeispiele, so dass auch dem interessierten Laien diese Form der Datenhaltung mit all ihren Möglichkeiten offen steht.

Der Laie sei aber davor gewarnt, größere Datenbanken ohne theoretische Grundkenntnisse selbst zu erstellen. Denn hier leitet sich der Datenbankentwurf in der Regel nicht mehr direkt aus der Aufgabenstellung ab. Doch nur ein guter Entwurf garantiert übersichtliche und optimale Zugriffe und ermöglicht je nach Bedarf Ergänzungen und Erweiterungen der Datenbank. Auch wird nur dadurch die Konsistenz und Integrität der Datenbank ermöglicht, so dass fehlerhafte, widersprüchliche und nicht mehr zugreifbare Datenbestände verhindert werden. Grundlagen zu Datenbankentwurf und -programmierung sollten für den Datenbankprogrammierer daher selbstverständlich sein.

Dieses Buch entstand aus Vorlesungen zu Datenbanken, die ich für Informatiker an der Fachhochschule Regensburg gelesen habe. Doch ich will auch den Nicht-Informatiker und interessierten Laien gezielt ansprechen, indem mittels zahlreicher Beispiele die Theorie direkt in die Praxis umgesetzt wird. Regelmäßige Zusammenfassungen erleichtern die Wiederholung und Reflektion des behandelten Stoffes. Als Lernzielkontrolle dienen dem Autodidakten die zahlreichen Aufgaben und deren Lösungen. Der Praxisbezug wird unterstützt durch eine im Buch laufend verwendete Beispieldatenbank, die im Anhang ausführlich vorgestellt wird.

Dieses Buch wendet sich vor allem an Anwendungsprogrammierer, die mit Hilfe von SQL auf Datenbanken zugreifen, und an alle, die Datenbanken neu

entwerfen, erweitern oder abändern wollen. Da viele Kapitel voneinander unabhängig sind, ist es aber auch als Nachschlagewerk empfehlenswert. Mittels der hier vermittelten Kenntnisse ist es relativ leicht, sich auch in die Datenbankprogrammierung mit grafischen Oberflächen einzuarbeiten.

Ganz grob lässt sich dieses Buch in fünf Teile gliedern. Die Kapitel 1 und 2 geben einen Überblick und können bei Vorkenntnissen zur Datenorganisation und Datenbanken ohne weiteres übersprungen werden. Die Kapitel 3 und 5 beschäftigen sich intensiv mit dem Aufbau und dem Design von relationalen Datenbanken. Dieser Stoff wird durch die in den Kapiteln 4, 6 und 9 behandelte Datenbanksprache SQL noch vertieft. Die Kapitel 7 und 8 stellen Spezialthemen wie Schutz vor Datenverlusten oder unberechtigten Zugriffen, Korrektheit der gespeicherten Daten und Möglichkeiten zum parallelen Datenzugriff vor. Die Kapitel 10 und 11 geben schließlich einen Überblick über nichtrelationale, verteilte und objektorientierte Datenbanken.

Im Einzelnen beinhaltet das erste Kapitel eine Zusammenfassung über die Datenorganisation. Dieser Überblick ist für das tiefere Verständnis einiger Datenbankkonzepte erforderlich. Er zeigt auch die Abgrenzung zwischen der reinen physischen Speicherung von Daten und dem logischen Datenaufbau in Datenbanken auf.

Kapitel 2 bespricht zum Einen die Vorzüge von Datenbanken gegenüber eigenverwalteten Datenbeständen, erklärt zum Anderen die wichtigsten Datenbankbegriffe und gibt zum dritten einen kleinen Überblick über die Leistungsfähigkeiten und Möglichkeiten von Datenbanken. Als Übersicht zu Datenbanken sei dieses Kapitel verstanden und daher insbesondere auch dem Anfänger empfohlen.

Das dritte und fünfte Kapitel sind den relationalen Datenbanken gewidmet. Sie sind die zentralen Kapitel dieses Buches. Hier lernen wir die Methoden zum Aufbau einer relationalen Datenbank im Einzelnen kennen. Der Aufbau von Relationen wird erklärt, die beiden Integritätsregeln werden vorgestellt, und die Normalformen von Relationen und das Entity-Relationship-Modell als Werkzeug zum Design von Datenbanken werden ausführlich behandelt.

Begleitend und praxisorientiert führen die Kapitel 4, 6 und 9 in die Sprache SQL (insbesondere SQL-2) ein, wobei an Hand von Beispielen der Stoff der Kapitel 3 und 5 noch vertieft wird. In Kapitel 4 lernen wir die Zugriffsmöglichkeiten auf Datenbanken mittels SQL genau kennen. In Kapitel 6 deklarieren wir Datenbanken mittels SQL. In Kapitel 9 betten wir schließlich die Sprache SQL in die Programmiersprache C++ ein, um auch umfangreichere Programme erstellen zu können. Da die einzelnen Datenbankhersteller von der

SQL-Norm geringfügig abweichen, wird auf entsprechende Unterschiede in Oracle und MS-Access regelmäßig hingewiesen.

Weiter sei auf die in der Praxis sehr wichtigen Kapitel 7 und 8 verwiesen. Die hier behandelten Themen sind nacheinander Recovery, Concurrency, Sicherheit und Integrität. Es werden ausführlich die erforderlichen Maßnahmen beschrieben, um einem Rechnerausfall mit Gelassenheit entgegenzusehen, um Parallelzugriffe ohne Beeinträchtigung zuzulassen, um für Sicherheit auch im Netzbetrieb zu sorgen und die Integrität der Daten schon beim Deklarieren der Datenbank so weit wie möglich zu garantieren.

Zuletzt zeigen Kapitel 10 und 11 im Überblick den Aufbau nicht-relationaler Datenbanken. Es wird das Grobkonzept hierarchischer und netzwerkartiger Datenbanken vorgestellt, aber auch die Grundideen objektorientierter und objektrelationaler Datenbanken werden betrachtet. Zuletzt werden die Grundregeln aufgelistet, die verteilte Datenbanken erfüllen sollten.

Am Ende der meisten Kapitel finden wir eine Zusammenfassung und Übungsaufgaben. Lösungen zu den Aufgaben sind in Anhang C aufgeführt. Anhang A enthält eine Beispieldatenbank, genannt *Radl*. Mit dieser *Radl*-Datenbank wird in diesem Buch ständig gearbeitet, insbesondere in den SQL-Kapiteln 4, 6 und 9, da sie als Basis für alle praktischen Übungen dient. Programme zum automatischen Erstellen dieser Datenbank für Oracle und MS-Access können über das Internet kostenlos bezogen werden. Hinweise hierzu finden wir im Anhang D. Anhang B rundet schließlich das Buch mit der Angabe der Syntax der im Buch verwendeten SQL-Befehle ab.

Die zweite Auflage wurde erheblich überarbeitet. Neben zahlreichen kleineren Änderungen wurden folgende umfangreichen Anpassungen vorgenommen: Die Beispieldatenbank *Radl* im Anhang wurde erweitert; der Abschnitt über objektorientierte Datenbanken wurde durch aktuelle Entwicklungen ergänzt; neben Oracle wird jetzt auch auf MS-Access verwiesen, dBase wurde durch MS-Access ersetzt; weiter dient jetzt C++ statt C als Einbettungssprache. Die vorliegende dritte Auflage enthält viele zusätzliche, kleine Korrekturen.

Mein besonderer Dank gilt Herrn Holderried, der mir bei der Erstellung der ersten maschinengeschriebenen Form behilflich war und damit den Stein ins Rollen brachte, Frau Hörbrand mit der Ausarbeitung der Beispieldatenbank Radl und insbesondere meiner Frau Ingrid und meinen Töchtern Kerstin und Sabine, deren Rücksicht und Verständnis dieses Buch erst ermöglichten.

Regensburg, im August 2000

Inhaltsverzeichnis

1 Datenorganisation

Dieses Kapitel enthält eine Übersicht über die Organisationsformen zum Speichern von Daten in einem Rechnersystem. Alle in einem Rechner später wieder benötigten Daten müssen gemerkt, das heißt physisch gespeichert werden. Ob es sich um das physische Speichern eines einfachen Textes oder einer komplexen Datenbank handelt, immer müssen die Daten sicher und wiederauffindbar hinterlegt werden. Beim physischen Speichern stehen nun zwei Anforderungen miteinander in Konflikt: Zum Einen sollen möglichst einfache Datenstrukturen verwendet werden. Dies garantiert geringe Fehleranfälligkeit und leicht lesbare, übersichtliche und damit einfach wartbare Programme. Zum Anderen sollen diese Daten so schnell wie möglich wieder gefunden werden. Dies erfordert aber meist einen komplexen Datenaufbau und aufwendige Algorithmen. In der Praxis werden wir somit je nach Anforderung Kompromisse zwischen diesen beiden Extremen eingehen müssen.

In diesem Kapitel führen wir in die Technik der Datenorganisation ein, beginnend bei einfachen Speicherformen, um schließlich zu komplexen Strukturen wie ISAM oder Hash zu gelangen. Die frühe Behandlung der Datenorganisation erlaubt im nächsten Kapitel die saubere Abgrenzung zu Datenbanken. Weiter wird später bei Abhängigkeiten der Datenbanken von ihrer physischen Speicherung auf die entsprechenden Stellen in diesem Kapitel verwiesen. Wird auf die Datenorganisation weniger Wert gelegt, so kann dieses Kapitel auch übersprungen werden.

1.1 Überblick

Die Hauptaufgabe jeden Rechners ist die Verwaltung und Manipulation von Daten. Abb. 1 zeigt stark vereinfacht den schematischen Aufbau eines Computers. Das Rechenwerk des Rechners besitzt einen direkten Zugriff zum Arbeitsspeicher. Daten, die nicht im Arbeitsspeicher stehen, werden von externen Medien wie Magnetplatte, Magnetband, Diskette, CD-ROM oder über ein angeschlossenes Netzwerk in den Arbeitsspeicher geladen. Hier im Arbeitsspeicher werden diese gelesen und gegebenenfalls manipuliert. Geänderte Daten werden auf das externe Medium zurückgeschrieben. Der Arbeitsspeicher

kommuniziert mit den externen Medien über ein Bus-System. Die Überwachung dieses Bus-Systems übernimmt bei kleineren Rechnern das Hauptrechenwerk selbst, bei großen Rechnern existieren hierfür eigene Subsysteme. Jedes größere Programm manipuliert Daten nach dem oben geschilderten Prinzip. Uns interessiert nun, wie diese Daten, die mehr oder weniger häufig gelesen oder geändert werden, intern gespeichert und verwaltet werden.

Heute ist die Magnetplatte das mit Abstand wichtigste Speichermedium. Gegenüber dem Arbeitsspeicher hat sie den Vorteil der Nichtflüchtigkeit, d.h. dass Daten auch nach dem Abschalten des Rechners verfügbar bleiben. Diese Eigenschaft der Nichtflüchtigkeit besitzen zwar auch andere Medien wie Band oder Diskette. Wie wir noch sehen werden, ist das Band aber meist ungeeignet, und die Diskette ist als universelles Speichermedium zu klein und zu langsam.

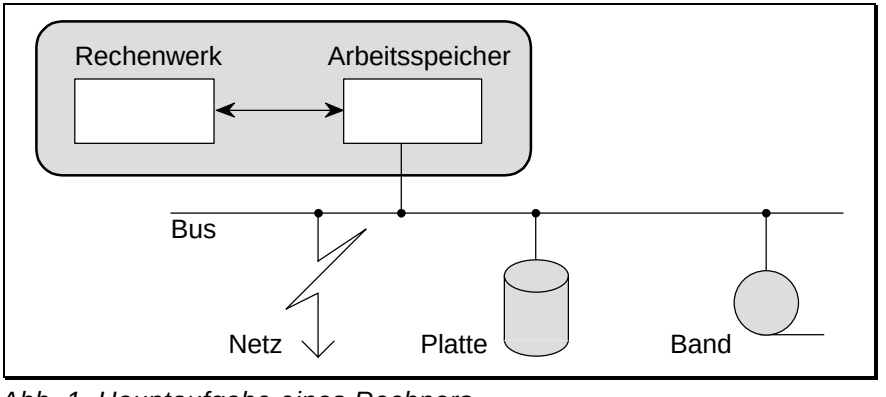


Abb. 1 Hauptaufgabe eines Rechners

Die Daten auf Magnetplatte (und den meisten anderen Medien) sind in logischen Einheiten zusammengefasst, den Dateien. Dateien sind die Speichereinheiten, die das Betriebssystem verwaltet, und die auch dem Anwender vom Betriebssystem direkt sichtbar gemacht werden. Mit Hilfe des Betriebssystems können wir daher nach Dateien suchen, diese verschieben, löschen oder neue Dateien anlegen. Da auch von Anwendern geschriebene Programme auf die Dienste der Betriebssysteme zurückgreifen, benutzen auch diese sehr intensiv die Dateistruktur. Zusätzlich greifen die meisten Programme über interne Schnittstellen auf die in diesen Dateien gespeicherten Daten zu, manipulieren diese Dateien also. Zur Manipulation werden Dateien (oder Teile davon) in den Arbeitsspeicher geladen. Von dort können sie vom Rechenwerk direkt bearbeitet und, falls gewünscht, wieder auf Magnetplatte zurückgeschrieben werden.

Zur einfacheren Verwaltung von Magnetplatten werden diese in eine Anzahl gleich großer Einheiten aufgeteilt, den sogenannten Blöcken. Blöcke sind je nach Betriebssystem und Plattenorganisation heute zwischen 512 Bytes und 4 kBytes groß. Für solche blockorientierte Speichermedien (auch Diskette und Band gehören dazu) unterstützt das Betriebssystem das blockweise Lesen und Schreiben. Hier werden die Daten nicht Zeichen für Zeichen von der Platte geholt, sondern es wird immer ein ganzer Block eingelesen und im Arbeitsspeicher zwischengepuffert. Das Gleiche gilt sinngemäß für das Schreiben.

Die Manipulierbarkeit von Daten, ob im Arbeitsspeicher, auf Magnetplatte oder anderen Medien, beinhaltet, dass einmal geschriebene Daten auch wieder auffindbar sein müssen, und das meist in möglichst kurzer Zeit. Da darüberhinaus häufig eher zu viel als zu wenig gespeichert wird, sind die Datenbestände entsprechend umfangreich. Denken wir nur an die Datenbestände von Großbanken, großen Versicherungsgesellschaften, der Verkehrssünderkartei in Flensburg oder den Verwaltungsdatenbanken von Großstädten.

Es stellt sich demnach die Frage, wie Daten abzuspeichern sind, damit diese auch bei sehr großen Datenbeständen schnell zugreifbar sind. Etwas haben wir bereits gelernt: Zunächst erfolgt eine Aufteilung der Gesamtdaten in Dateien. Wir benötigen jetzt noch eine Organisation innerhalb dieser einzelnen Dateien. Diese Datenorganisation sollte so optimiert werden, dass folgende Wünsche erfüllt werden:

Wünsche

☺ schnelles Finden von Daten innerhalb einer Datei (geringer Zeitaufwand!)

☺ wenig Speicherverbrauch beim Abspeichern aller Daten (geringe Kosten)

☺ wenig Aufwand beim Löschen, Einfügen, Ändern (geringer Zeitaufwand)

☺ einfacher Algorithmus (wenig Wartung und geringe Fehleranfälligkeit)

Leider sind alle diese vier sehr verständlichen Wünsche gleichzeitig nicht erfüllbar, Kompromisse müssen eingegangen werden. Wie wir noch sehen werden, muss zumindest auf einen der vier Punkte verzichtet werden. Wir beginnen mit einfachen Algorithmen, um die Speichermethodik kennenzulernen. Sind uns dann grundlegenden Probleme der Speichertechnik vertraut, so können wir zu den komplexeren Speicheralgorithmen übergehen.

1.2 Speicherung von Daten

In diesem Abschnitt werden grundlegende Speicherungsmöglichkeiten vorgestellt. Zunächst beschäftigen wir uns mit den Speichermedien selbst, wobei uns interessiert, wie auf diesen Medien Daten abgespeichert und wiedergefunden werden können. Anschließend lernen wir kennen, wie wir Daten auf diesen Speichermedien dann tatsächlich ablegen können. Beginnen wir mit den Speichermedien.

Hier unterscheiden wir zwischen sequentiellen und adressierbaren Speichermedien. Tab. 1 gibt einen Überblick über die typischen Eigenschaften dieser beiden Medien. Beim sequentiellen Speicher ist wesentlich, dass alle Daten, beginnend am Anfang des Mediums, direkt hintereinander geschrieben werden. Das spätere Löschen einzelner Daten ist nicht möglich. Gegebenenfalls müssen alle Daten gemeinsam gelöscht werden. Dieses Löschen erfolgt in der Regel durch Überschreiben. Eine Folgerung ist der geringe Speicherverbrauch, da weder Lücken zwischen einzelnen Daten existieren noch entstehen können. Nachteilig wirkt sich die Inflexibilität aus: Einfügen ist grundsätzlich nicht möglich, Anfügen geschieht ausschließlich am Ende des bereits beschriebenen Teils, wobei dieses Ende erst vom Anfang an sequentiell gesucht werden muss. Auch jeder Suchvorgang erfolgt grundsätzlich sequentiell, beginnend am Datenanfang. Dies zieht zwangsläufig lange Suchzeiten nach sich.

Tab. 1 Vergleich von Speichermedien

Speichermedium	**Sequentiell**	**Adressierbar**
Vorteile	billiges Medium, geringer Speicherbedarf	Direktzugriff möglich, Einfügen möglich
Nachteile	Einfügen nicht möglich, Suchen nur sequentiell	Medium nicht ganz billig, oft ungenutzte Lücken
Typisches Medium	Magnetband, Magnetbandkassette	Magnetplatte, Disketten, Arbeitsspeicher
Typischer Einsatz	Archivierung, Datensicherung	Überall, wo auf Daten direkt zugegriffen wird

Anders verhält es sich beim adressierbaren Speicher. Jeder Speicherplatz ist direkt erreichbar, z.B. beim Arbeitsspeicher, oder zumindest extrem schnell, z.B. bei der Magnetplatte über den beweglichen Plattenarm. Dies erlaubt ein gezieltes Suchen über die Speicheradresse und ein beliebiges Einfügen und Löschen von Datenteilen. Dieser Vorteil kostet zunächst einen gewissen ma-

teriellen Aufwand, etwa teure Speicherchips oder aufwendige Mechanik für Plattenarm und Plattendrehgeschwindigkeit. Darüberhinaus muss eine Organisation existieren, die jedes an einer bestimmten Stelle abgespeicherte Datum wieder findet. Dies ist ein Grund, warum Magnetplatten und Disketten formatiert werden müssen. Eine 2 MByte-Diskette besitzt nach dem Formatieren allerdings nur noch 1,44 MBytes nutzbaren Speicher!

Zusammenfassend lässt sich sagen, dass heute fast ausschließlich adressierbare Speichermedien verwendet werden. Nur da, wo die Vorteile dieser Medien nicht genutzt werden können, etwa bei der Archivierung oder Datensicherung, wird gerne auf die wesentlich billigeren Magnetbandkassetten zurückgegriffen (eine Streamer-Kassette kann viele GBytes an Daten speichern!).

Wichtig für das weitere Verständnis ist:

- ☞ auf sequentielle Speichermedien kann grundsätzlich nur sequentiell zugegriffen werden,
- ☞ auf adressierbare Speichermedien kann sowohl sequentiell als auch über die Adresse zugegriffen werden.

Diese beiden physischen Speicherformen beeinflussen auch direkt die Datenorganisation: Dateien lassen sich daher logisch gesehen immer einer der beiden folgenden Dateiarten zuordnen:

1) **Sequentielle Dateien:**
   Die einzelnen Datensätze sind innerhalb der Datei nacheinander (sequentiell) ohne Lücken abgespeichert.
   Anwendung*:* Archivierung, Sicherung, kleine Dateien (< 10 MBytes)

2) **Adressierte Dateien:**
   Die einzelnen Datensätze sind innerhalb der Datei an bestimmten Adressen abgespeichert. Auf diese Datensätze kann direkt über die Adresse zugegriffen werden, vorausgesetzt diese Adresse wurde gemerkt. Eine aufwendige Suche innerhalb der Datei entfällt.
   Anwendung: große Dateien (im MByte- bis GByte-Bereich), wo kurze Antwortzeiten beim Zugriff auf bestimmte Daten gefordert werden.

Unter einem Datensatz verstehen wir generell eine logische Einheit innerhalb einer Datei. Bei Textdateien ist dies meist ein Satz oder eine Zeile, bei Tabellen oder Karteien eine Zeile.

Es versteht sich von selbst, dass für direktadressierte Dateien auch ein adressierbares Speichermedium erforderlich ist. Die vielfältigen Möglichkeiten des Speicherns von Daten auf adressierbaren Medien wollen wir im Folgenden noch genauer betrachten.

1.3 Speicherung auf adressierbarem Speicher

Wie bereits erwähnt, kann bei adressierbaren Speichermedien auf die Daten direkt zugegriffen werden, falls die Speicheradresse bekannt ist. Doch woher kennen wir die Speicheradresse der gesuchten Daten? Das Problem des Suchens von Daten wird auf das Suchen der dazugehörigen Speicheradresse abgebildet. Wir gewinnen anscheinend nichts. Wir werden uns mit diesem Problem noch eingehend bei den mehrstufigen physischen Datenstrukturen beschäftigen. Es gibt aber auch relativ einfache Möglichkeiten, dieses Suchproblem zu lösen:

1.3.1 Direktadressierung

Bei der Direktadressierung ist die Speicheradresse eine Funktion der dort gespeicherten Daten. Wir wollen dies am Beispiel eines elektronischen Notizbuchs (Kalenders) aufzeigen. Gehen wir davon aus, dass in unserem Notizbuch pro Tag 100 Bytes Speicher zur Verfügung stehen. Dies entspricht dem bewährten Terminkalender, wo pro Tag ein fester Bereich für Notizen existiert. Werden nun Einträge zum 2. Februar gesucht, so wird der Rechner diese Daten an Hand folgender Rechnung sofort finden:

```
Der 2. Februar ist der 33. Tag des Jahres.
Infos zu diesem Tag stehen im Speicherbereich 3200 - 3300 ab Dateibeginn.
```

Völlig analog kann zu jedem Datum die dazugehörige Speicheradresse ermittelt werden, eine sequentielle Suche ist überflüssig.

Dieses Beispiel zeigt uns folgendes: Es wird Information zu einem Thema gesucht. Das Thema, oder genauer der Suchbegriff, war im obigen Beispiel das Kalenderdatum. Dieser Suchbegriff enthielt verschlüsselt auch die Information zum direkten Auffinden der Daten. Wir nennen ihn daher den Suchschlüssel der Datei. Betrachten wir diesen Suchvorgang mathematisch, so ist die gesuchte Adresse $A_{gesucht}$ eine Funktion f des Suchschlüssels:

$A_{gesucht}$ = f (*Suchschlüssel*)

Im Falle des obigen Beispiels mit dem Kalendertag als Suchschlüssel gilt:

$A_{gesucht}$ = 100 * (*Kalendertag* - 1) + *Startadresse*

1.3.2 Geordnete Speicherung mit Suchschlüssel

Bei der Direktadressierung beeinflusst der Suchschlüssel den Aufbau der Datei maßgeblich: Pro Schlüssel existiert ein eindeutig identifizierbarer, fester Speicherbereich. Der Gewinn ist der sofortige und damit extrem schnelle Zugriff. Aber auch andere Datenorganisationen, die Daten in Abhängigkeit vom Suchschlüssel speichern, ermöglichen einen schnellen Zugriff. Denken wir dabei nur an die Möglichkeit, alle Daten einer Datei nach dem Suchschlüssel geordnet anzulegen. Ein typischer Vertreter dieser Speicherungstechnik ist das Telefonbuch. Der Suchbegriff ist hier der Name des Fernsprechteilnehmers, die gesuchte Information ist die Telefonnummer, eventuell auch die Adresse.

Beim Telefonbuch versagt die im vorherigen Abschnitt angesprochene Direktadressierung, da zu jedem nur denkbaren Namen Speicher reserviert werden müsste (beim Notizbuch existierte zu jedem Tag des Jahres ein Speicher von 100 Bytes!). Bei Namen mit maximal 20 relevanten Zeichen und bei 26 verschiedenen Buchstaben wären das 26^{20} mögliche Namenskombinationen. Wir würden Trilliarden von GBytes an Speicher benötigen, was den Rahmen jedes nur denkbaren Speichermediums bei weitem sprengen würde.

Stattdessen wird im Telefonbuch jeder Eintrag direkt hinter dessen Vorgänger in alphabetischer Reihenfolge des Namens abgespeichert. Wir können zwar jetzt nicht aus dem Namen eines Fernmeldeteilnehmers direkt auf den Platz schließen, wo dessen Telefonnummer gespeichert ist, doch es gibt einen weiteren, relativ schnellen Zugang: die Binäre Suche. Der Algorithmus der Binären Suche ist in Abb. 2 angegeben.

Bei der Binären Suche betrachten wir zunächst die gesamte Datei, lesen den in der Mitte dieser Datei stehenden Suchschlüssel *S* und überprüfen, ob der zu suchende Schlüssel mit *S* übereinstimmt oder größer oder kleiner als *S* ist. Im ersten Fall können wir die Suche sofort beenden, andernfalls wissen wir (wegen der alphabetischen Anordnung der Schlüssel), dass eine weitere Suche nur in der rechten bzw. linken Hälfte Erfolg haben kann. Mit einem einzigen Zugriff schränken wir demnach die noch zu durchsuchenden Daten auf die Hälfte

ein. Durch Fortsetzen dieses Verfahrens nähern wir uns rasch dem gesuchten Eintrag.

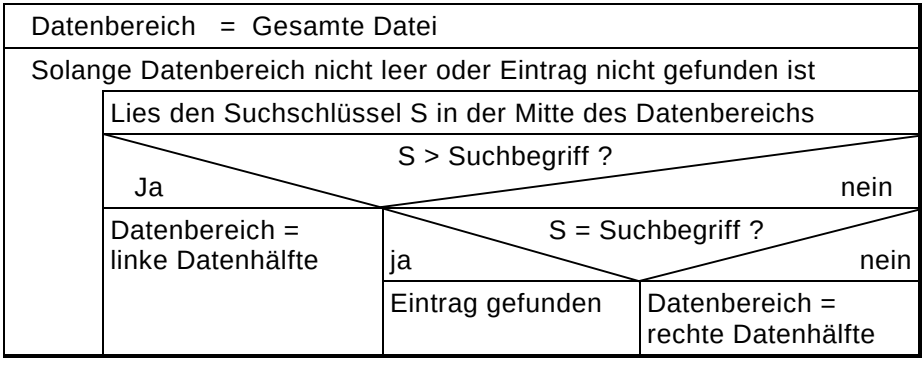


Abb. 2 Algorithmus der Binären Suche

Dass dieser Algorithmus wirklich sehr effizent ist, zeigt ein kleines Beispiel: Nehmen wir an, wir hätten ein elektronisches Telefonbuch mit etwa einer Million nach Namen sortierten Einträgen. Ein sequentieller Zugriff würde im Mittel 500000 Suchvorgänge nach sich ziehen, da der zu suchende Name mal weiter vorne, mal weiter hinten im Alphabet stehen wird. Da 1 Million etwas weniger als 2^{20} ist, sind zur erfolgreichen Suche nur maximal 20 Halbierungen erforderlich. Und auch bei einer Milliarde Einträge bräuchten wir mit Hilfe der Binären Suche nie mehr als 30 Zugriffe!

Tab. 2 Vor- und Nachteile von Speichertechniken

	Direktadressierung	**Binäre Suche**
Vorteile	Direktzugriff, direktes Einfügen, direktes Löschen, einfacher Algorithmus	relativ schnelle Suche, minimaler Speicherverbrauch, leicht anwendbar
Nachteile	nur bedingt anwendbar (wegen des zum Teil sehr hohen Speicherverbrauchs)	Löschen aufwendig, Einfügen aufwendig
Abhilfe	⇒ Hash-Verfahren	⇒ B-Baum ⇒ ISAM-Datei

Dass damit trotzdem noch nicht alle Probleme gelöst sind, zeigt das Problem des Einfügens und Löschen in einer solchen Datei. Beides ist nur durch Verschieben der nachfolgenden Einträge möglich, ein aufwendiges Verfahren.

Wir wollen hier nicht auf weitere Details eingehen, sondern stattdessen eine kleine Zusammenfassung der Direktadressierung und der Binären Suche angeben. Wir können Tab. 2 entnehmen, dass die Vorteile des einen Algorithmus im wesentlichen die Nachteile des anderen und umgekehrt sind. Es sind in der Tabelle zusätzlich in der Zeile *Abhilfe* Zugriffsmethoden erwähnt, die die Nachteile größtenteils beseitigen, allerdings auf Kosten der Einfachheit der bisherigen Algorithmen. Wir kommen auf diese komplexeren Algorithmen ab Abschnitt 1.5 zurück.

1.4 Einstufige physische Datenstrukturen

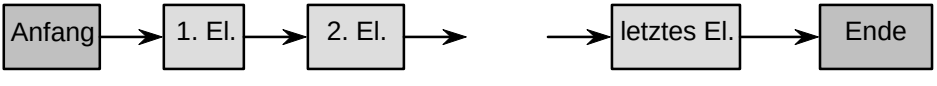
Während es im letzten Abschnitt um die Speicherungsmöglichkeiten allgemein ging, wollen wir hier die vier grundsätzlichen Arten der physischen Datenspeicherung kennenlernen. Wir zeigen hier also die Technik, Daten physisch auf den Speichermedien abzulegen.

1.4.1 Listen auf sequentiellem Speicher

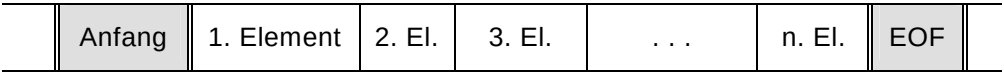
Listen sind die einfachste Methode der Datenspeicherung und eignen sich besonders gut für sequentielle Speicher, aber auch allgemein bei kleineren Datenbeständen. Merkmale dieser Listen auf sequentiellem Speicher sind:

- der Anfang der Liste ist bekannt,
- zu jedem Element existiert ein direktes Nachfolgeelement,
- am Ende steht ein Ende-Signal.

Dies lässt sich wie folgt darstellen:



Beachten Sie allerdings, dass es sich bei den Zeigern nicht um Adresszeiger handelt. Die Elemente stehen hier auch physisch hintereinander. Auf Magnetbanddateien, der Hauptanwendung von Listen auf sequentiellem Speicher, sieht die Speicherung der Daten wie folgt aus:



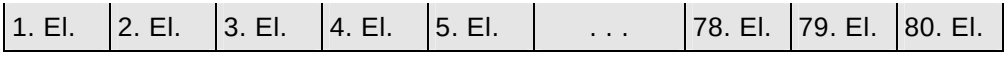
Hier steht „EOF“ für die Endemarkierung (End of File). Neben den bereits mehrfach erwähnten Magnetbändern und -bandkassetten werden diese Listen wegen ihrer Einfachheit auch für viele andere Zwecke angewendet, etwa für Textdateien. Bei Textdateien übernehmen zum Beispiel Editoren die Aufgabe der Aufbereitung dieser ansonsten sequentiell geschriebenen Daten. Hier entspricht ein Element in der Regel einer Zeile eines Textes. Mit Hilfe solcher Editoren lassen sich beliebige Änderungen vornehmen. Beachten Sie aber, dass diese Editoren immer die gesamte Textdatei in den Arbeitsspeicher holen, hier ändern und beim Speichern immer den gesamten Text komplett wieder auf Magnetplatte sichern müssen.

1.4.2 Tabellen auf adressierbarem Speicher

Bei Tabellen auf adressierbarem Speicher liegt die in Unterabschnitt 1.3.1 beschriebene Direktadressierung zugrunde. Die Merkmale sind:

- Jedes Element ist direkt zugreifbar,
- Jedes Element belegt gleich viel Speicherplatz.

Gerade die letzte Eigenschaft ist bezeichnend für diese Speicherungstechnik. Es muss auch für im Augenblick nicht belegte Elemente einer Tabelle Speicherplatz reserviert werden. Hier gibt es auch keine Anfangs- oder Ende-Informationen. Dies ist auch nicht notwendig, denn die Tabelle ist statisch fest vorgegeben. Eine Tabelle mit 80 Elementen besitzt daher folgendes Aussehen:



Anwendungen neben der Datenorganisation gibt es speziell in der Mathematik und in der Programmierung: Vektoren, Matrizen und Felder. Diese Tabellenstruktur wird immer dann gerne verwendet, wenn es direkt aufeinanderfolgende Suchschlüssel gibt, etwa die Daten im Kalender, Zimmernummern, Matrikelnummern oder Personalnummern. Nicht geeignet ist dieses Verfahren leider für einige der wichtigsten und häufigsten Suchschlüssel: der Name einer Person oder die Bezeichnung eines Gegenstandes.

1.4.3 Geordnete Listen auf adressierbarem Speicher

Im Gegensatz zu den Listen auf sequentiellem Speicher sind bei geordneten Listen die Daten nach dem Suchbegriff sortiert abgespeichert. Wir benötigen hierzu einen adressierbaren Speicher, denn mit einem sequentiellen Speicher müssten wir weiterhin sequentiell suchen (mit dem nur minimalen Vorteil, dass wir nicht erst am Dateiende erkennen, wenn ein Suchbegriff nicht in der Liste enthalten ist). Die Vorteile kommen durch die Binäre Suche zum Tragen. Merkmale der geordneten Listen sind:

- der Anfang ist bekannt,
- zu jedem Listenelement existiert ein direktes Nachfolgeelement,
- die Elemente sind nach dem Suchbegriff geordnet,
- das Binäre Suchverfahren ist anwendbar,
- am Ende steht ein Ende-Signal.

Der vierte Punkt ist eigentlich eine Folge des dritten, wurde aber wegen seiner Wichtigkeit trotzdem explizit aufgeführt. Es sei noch erwähnt, dass nicht notwendigerweise alle Einträge gleich lang sein müssen. Bei gleich großen Einträgen ist die Implementierung der Binären Suche allerdings wesentlich einfacher. Der Anfang eines Elements kann so leicht direkt ermittelt werden. Andernfalls muss ein spezielles Kennzeichen als Trennsymbol zwischen den einzelnen Elementen eingefügt und dann beim Zugriff jeweils gesucht werden.

Anwendung finden geordnete Listen in elektronischen Karteien und Büchern. Diese Struktur wird gerne gewählt, wenn nur sehr selten Daten geändert werden müssen. Weitere Informationen, insbesondere zur Binären Suche, finden sich in Unterabschnitt 1.3.2.

1.4.4 Geordnete verkettete Listen

Zwischen den im vorigen Abschnitt vorgestellten Listen und den geordneten verketteten Listen gibt es einen kleinen Unterschied in der physischen Speicherung. Während bei den Listen aus dem letzten Unterabschnitt alle Elemente physisch hintereinander angeordnet sind, ist dies bei verketteten Listen nicht notwendigerweise der Fall. Die Reihenfolge wird vielmehr über Zeigerketten hergestellt. Bei dieser Datenstruktur handelt es sich um eine Listenstruktur. Die Merkmale sind hier:

- der Anfang der Liste ist bekannt,
- zu jedem Element existiert ein Verweis auf das nächste Element,
- das Ende ist markiert.

Die einzelnen Elemente stehen also in der Regel nicht physisch geordnet hintereinander, über Zeiger ist allerdings eine logische Ordnung sichergestellt. Abb. 3 stellt einen Ausschnitt aus einem Speicher dar, die Elemente sind nur logisch, nicht physisch geordnet.

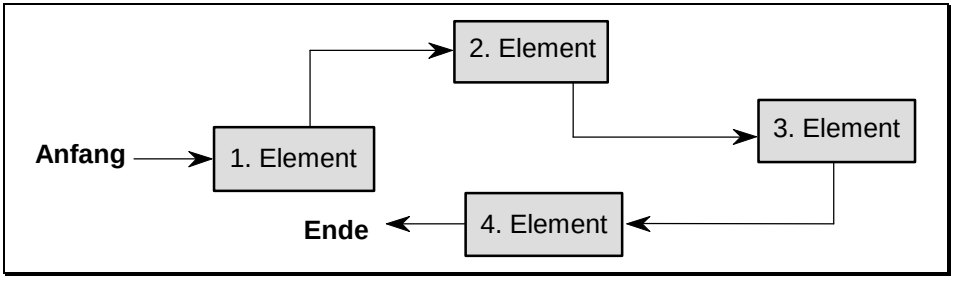


Abb. 3 Geordnete verkettete Listen auf adressierbarem Speicher

Diese Speichertechnik erlaubt das einfache Ein- und Ausfügen von Elementen. Es muss zum Beispiel beim Einfügen nur ein Element an einer beliebigen Stelle im Speicher geschrieben und mit Hilfe des Umbiegens von Zeigern an der gewünschten Stelle logisch eingefügt werden. Diesen großen Vorteil erkaufen wir uns allerdings mit dem Nachteil, dass hier die Binäre Suche nicht mehr anwendbar ist.

Verwendung finden verkettete Listen vor allem im Arbeitsspeicher beim dynamischen Anlegen von Daten. Aber auch in vielen anderen Fällen, zum Beispiel in Texteditoren und zur Datei- und Prozessverwaltung in Betriebssystemen, wird dieses Verfahren angewendet.

1.4.5 Zusammenfassung

Um einen kleinen Überblick zu erhalten, wollen wir die bisherigen Verfahren einschließlich ihrer Vor- und Nachteile nochmals tabellarisch zusammenfassen und einander gegenüberstellen.

Studieren wir Tab. 3 genau, so werden wir feststellen, dass wir mit keiner der vier Datenstrukturen übermäßig glücklich werden; es sei denn, dass im entsprechenden Anwendungsfall die eine oder andere negative Eigenschaft

nicht weiter stört. Ferner fällt auf, dass nur die Tabellen sowohl eine kurze Zugriffsdauer als auch einen geringen Änderungsaufwand garantieren. Die anderen Verfahren zeigen in mindestens einem dieser beiden Fälle Schwächen.

Tab. 3 Zusammenfassung der einstufigen Datenstrukturen

	sequentielle Listen	**Tabellen**	**geordnete Listen**	**geordnete verkettete Listen**
Speicherung	sequentiell	adressiert	sequentiell geordnet	adressiert
Speicher	sequentiell	adressierbar	adressierbar	adressierbar
Speicherverbrauch	sehr gering, da keine Lücken (++)	sehr hoch, große Lücken (--)	gering, aber Elemente gleich groß (+)	gering, etwas Speicher für Adressen (+)
Zugriffsdauer	sequentielle Suche (-)	genau 1 Zugriff (++)	Binäre Suche (+)	sequentielle Suche (-)
Änderungsaufwand	Anfügen nur am Ende (--)	direkte Änderung (++)	Verschieben der Folgeelemente (-)	nur Adressen ändern (+)
Einsatzmöglichkeit	universell	stark eingeschränkt	universell auf adressierbarem Speicher	universell auf adressierbarem Speicher
Komplexität	gering	gering	gering	gering

Diese Schwächen lassen sich beseitigen, indem wir nicht mehr direkt (einstufig), sondern über Zwischenschritte (mehrstufig) auf die Daten zugreifen. Dies erlaubt zum Einen, bei Tabellen auf die Lücken zu verzichten, und zum Anderen, die geordneten Listen und die geordneten verketteten Listen geschickt miteinander zu verknüpfen. In beiden Fällen verschwinden die bisherigen Nachteile, allerdings auf Kosten der Komplexität der Algorithmen.

1.5 Mehrstufige Datenstrukturen

Bei einstufigen Datenstrukturen wird direkt auf die Daten zugegriffen. Dies garantiert einfache Algorithmen, zieht aber entweder lange Zugriffszeiten oder einen hohen Änderungsaufwand nach sich. Hier helfen nur Datenstrukturen weiter, die hierarchisch aufgebaut sind. Der Zugriff erfolgt dann mehrstufig über diese Hierarchie. Dabei stehen in diesen Zwischenstufen meist keine Nutzdaten, sondern nur Verwaltungsinformationen (z.B. Adressen) zu den

eigentlichen Daten. Besonders anschaulich lässt sich diese Struktur am Beispiel von Bäumen, insbesondere binären Bäumen, darstellen.

Ein solcher binärer Baum entsteht beispielsweise bei der Implementierung der Binären Suche in einer Datenstruktur. Vollziehen wir dies an einem „Telefonbuch“ mit 15 Einträgen nach. Die Wurzel des binären Baums ist die erste Hierarchiestufe. Diese enthält den Suchbegriff und die Speicheradresse des mittleren, hier also des achten Elements. Von diesem Element gehen außerdem Verweise zu den Suchbegriffen der mittleren Elemente der linken und rechten Hälfte. Diese beiden neuen Verweise bilden die zweite Hierarchiestufe. In unserem Beispiel folgt noch eine dritte, bevor wir schließlich mit Sicherheit bei den Nutzdaten ankommen. Anschaulich ist diese Hierarchie in Abb. 4 dargestellt.

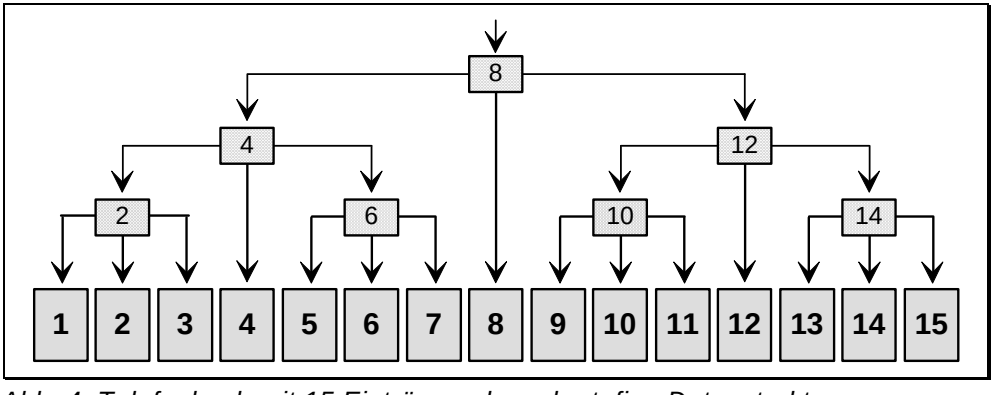


Abb. 4 Telefonbuch mit 15 Einträgen als mehrstufige Datenstruktur

Dass diese mehrstufigen Strukturen bei großen Datenbeständen sogar im Vergleich zur Binären Suche noch erhebliche Vorteile einbringen können, wollen wir im Folgenden aufzeigen: Stellen wir uns dazu eine Datei mit etwa einer Million Einträgen à 1000 Bytes vor. Diese Datei kann nicht mehr komplett im Arbeitsspeicher gehalten werden. Jeder einzelne Suchzugriff erfordert daher bei der Binären Suche einen Zugriff auf die Festplatte. In der Summe ergibt dies etwa $\log_2 10^6 \approx 20$ Festplattenzugriffe, was insbesondere im Mehrbenutzerbetrieb, verschärft durch Zugriffe auch von anderen Nutzern auf diese Festplatte, zu einigen Verzögerungen führt. Ein ganz anderes Bild ergibt sich im obigen Fall des mehrstufigen Zugriffs. Da hier in den Hierarchiestufen außer den Suchschlüsseln nur noch Adressen gemerkt werden, benötigen wir für die Hierarchiestufen kaum mehr als 20 MBytes Speicherplatz. Diese Hierarchiedaten lassen sich daher leicht permanent im Arbeitsspeicher eines größeren Rechners halten. Somit geschieht die Suche über die Hierarchiestufen im

Arbeitsspeicher, zum Einlesen der Nutzdaten ist schließlich nur ein einziger Plattenzugriff erforderlich. Die Antwortzeit ist unübertroffen schnell.

Mit dieser mehrstufigen Datenstruktur besitzen wir in der Praxis alle Vorteile der geordneten Listen auf adressierbarem Speicher, insbesondere ist die Zugriffsdauer auf einzelne Daten gering. Aus Abb. 4 erkennen wir, wie wir bereits weiter oben erwähnten, dass alle Nutzdaten über die Hierarchiestufen verkettet werden können. Die Nutzdaten müssen daher nicht physisch nacheinander angeordnet sein. Dies bringt uns den weiteren Vorteil ein, dass Änderungen, einschließlich Einfügen und Löschen, relativ einfach möglich sind.

Wir dürfen aber auch die Nachteile nicht verschweigen. Zunächst brauchen wir zusätzlichen Speicherplatz für die Hierarchiestufen. Da dieser in der Regel aber nur wenige Prozent von der Gesamtdatenmenge ausmacht, fällt dies meist kaum ins Gewicht. Viel gravierender ist, dass jedes Löschen und Einfügen in den Nutzdaten auch eine Änderung in den Hierarchietabellen erfordert. Weiter müssen die Bäume durch ein häufiges Ein- und Ausfügen von Nutzdaten mit der Zeit nicht mehr so schön ausgeglichen sein wie in Abb. 4. Wir wollen dieses Problem der „Schieflastigkeit“ von Bäumen an dieser Stelle nicht weiter vertiefen. Es sei nur erwähnt, dass dadurch die Suchzeiten merklich anwachsen können, und dass dieses Problem mittels ausgeglichener Bäume, gewichtsbalancierter Bäume oder B-Bäume gelöst werden kann. Einen auf B-Bäume beruhenden Algorithmus werden wir im nächsten Abschnitt kennenlernen. Eine Verwendung dieser Möglichkeiten führt aber gleichzeitig zu immer komplexeren Algorithmen. Zu Einzelheiten sei auf [Wirt83], [OtWi93] und [Mehl88] verwiesen.

Wir haben die Ergebnisse dieses Abschnitts in Tab. 4 knapp zusammengefasst. Wir erkennen die Vorteile einer Baumstruktur gegenüber einstufigen Datenstrukturen. Allerdings steigt die Komplexität der benötigten Algorithmen erheblich.

Anwendungen von Bäumen finden wir außer bei obigen Datenstrukturen sehr häufig beim Aufbau von dynamischen Strukturen im Arbeitsspeicher. Als einfache Beispiele sei der Verzeichnisbaum von Dateisystemen und das Zerlegen von Ausdrücken bei Compilern erwähnt.

Das Hauptproblem bleibt, dass diese Baumstrukturen nur wirklich effektiv sind, wenn möglichst viele Baumdaten im Hauptspeicher stehen. Ansonsten kann der Ein-/Ausgabebereich auf Magnetplatte leicht zum Engpass werden. Wir stellen daher im folgenden Abschnitt eine Methode vor, die auf den bisherigen Überlegungen der mehrstufigen Datenstrukturen aufbaut und gleichzeitig

die Anzahl der Zugriffe auf das langsame Speichermedium (Magnetplatte) minimiert.

Tab. 4 Zusammenfassung der Baumstrukturen

	Baumstrukturen
Speicherung	adresssiert
Speicher	adressierbar
Speicherverbrauch	noch relativ gering (+)
Zugriffsdauer	kurz, da über Hierarchiestufen zugegriffen wird (+)
Änderungsaufwand	nur Adressen ändern, eventuell auch Hierarchiedaten (+)
Einsatzmöglichkeit	universell auf adressierbarem Speicher
Komplexität	hoch

1.6 Index Sequentielle Dateien

Die indexsequentielle Zugriffsmethode auf Dateien ist eine konsequente Weiterentwicklung der Baumstrukturen für Anwendungen auf externen, adressierbaren Speichermedien. Zugrunde liegt dieser Methode die Theorie über B-Bäume. Diese indexsequentielle Zugriffsmethode ist in der Praxis weit verbreitet. Sie sorgt dafür, dass beim Zugriff auf Daten solcher Dateien die Anzahl der Dateizugriffe minimiert wird. Es existieren heute zwei gebräuchliche Abkürzungen für diese Zugriffsmethode:

ISAM für Index Sequential Access Method
VSAM für Virtual Sequential Access Method

Der letzte Begriff wird vor allem von IBM für dessen indexsequentielle Dateiformate verwendet, in der Literatur verbreiteter ist der Begriff ISAM. Der Name *Index* leitet sich davon ab, dass die Hierarchiestufen nur die nötigsten Informationen enthalten, im wesentlichen nur die Suchbegriffe (Indizes). Die einzelnen Zugriffs- oder Hierarchiestufen heißen auch Indexstufen.

Um die Plattenzugriffe so gering wie möglich zu halten, nutzt ISAM die Struktur der auf Magnetplatten gespeicherten Daten aus. Es wird an dieser Stelle der grundsätzliche Aufbau von externen adressierbaren Massenspeichern als bekannt vorausgesetzt. Wir wollen hier nur eine kurze Zusammenfassung angeben: Alle modernen Betriebssysteme verwalten den Massenspei-

cher block- und nicht zeichenweise. Das externe Speichermedium ist dabei in logisch gleich große Teile, den Blöcken, partitioniert. Typische Blockgrößen in Betriebssystemen sind:

512 Bytes	in MS-DOS,
512 - 4096 Bytes	in UNIX,
variabel, meist 2048 Bytes	in MVS,
2048 Bytes	in BS2000

In UNIX muss die Blockgröße beim Installieren des Systems vorgegeben werden, in MVS (Großrechnerbetriebssystem von IBM) kann die Blockgröße vom Benutzer eingestellt werden, was allerdings einen erheblichen Verwaltungsaufwand in diesem System nach sich zieht. Wesentlich ist nun, dass bei jeder Ein-/Ausgabe immer mindestens ein Block vollständig ein- oder ausgelesen wird, auch wenn beispielsweise nur wenige Bytes benötigt werden.

Und gerade darin liegt das Geheimnis von ISAM. Die Indexdaten werden so weit wie möglich in einen einzigen Block geschrieben. Zum Lesen mehrerer Indexdaten ist dann nur das Einlesen eines einzigen Blocks erforderlich. Um den Aufbau innerhalb eines Indexblocks einfach zu halten, werden die Indizes in diesem Block sequentiell abgespeichert. Für einen Rechner stellt die Suche eines Indexeintrags in einem einzelnen Block kein Problem dar. Betrachten wir nach diesen Vorüberlegungen den Aufbau einer ISAM-Datei etwas ausführlicher.

Eine ISAM-Datei besteht aus zwei Teilen, zum Einen den abgespeicherten Daten in den Datenblöcken und den zum schnellen Suchen erforderlichen Indexstufen in den Indexblöcken. Die Daten sind logisch nach dem Suchschlüssel geordnet. Physisch sind die Daten allerdings nur innerhalb jedes einzelnen Datenblocks geordnet. Die Blöcke selbst können nämlich beliebig auf dem Speichermedium verteilt sein. Die Indexblöcke enthalten die Informationen (Indizes), in welchen Blöcken welche Daten zu finden sind. Auch diese Indizes sind logisch aufsteigend geordnet. Wie wir noch sehen werden, besteht auch hier eine physische Ordnung nur innerhalb jedes einzelnen Indexblocks.

Die genaue Funktionsweise der indexsequentiellen Zugriffsmethode wollen wir an Hand eines Beispiels kennenlernen. Betrachten wir dazu eine Kartei, die pro Person folgende Daten speichert:

Name (Suchschlüssel)	20 Bytes	Vorname	20 Bytes
Adresse	40 Bytes	Geburtsdatum	6 Bytes
Telefonnummer	10 Bytes	Beruf	20 Bytes

Die Byte-Angaben geben den jeweils benötigten Speicherplatz an. Ein kompletter Eintrag braucht demnach 116 Bytes Speicherplatz. Nehmen wir eine Blockgröße von 1024 Bytes an, so stehen maximal 8 Sätze in jedem Block.

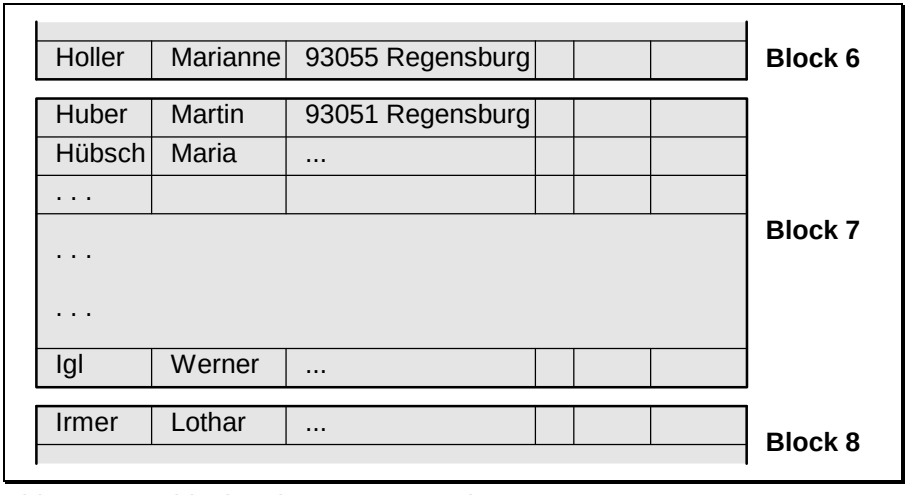


Abb. 5 Datenblöcke einer ISAM-Datei

Betrachten wir jetzt auszugsweise die logischen Blöcke 6 bis 8 unserer fiktiven Kartei in Abb. 5. Beachten Sie, dass es sich in der Abbildung nur um die logische Anordnung der Blöcke handelt. Physisch müssen die Blöcke nicht notwendigerweise hintereinander abgespeichert sein.

Von besonderem Interesse ist jetzt der Aufbau des zu diesen Datenblöcken gehörigen Indexblocks. Dieser ist in Abb. 6 ausschnittweise angegeben. Jeder Eintrag in diesem Indexblock besteht aus einem Suchschlüssel und einer Adresse. Es wird nur jeder erste Suchschlüssel eines Datenblocks zusammen mit der Adresse dieses Blocks eingetragen. Ein Datenblock wird also mit seinem ersten Suchschlüssel identifiziert. In unserem Fall werden dafür nur 24 Bytes (20 Bytes für den Namen und 4 Bytes für die Adresse) pro Eintrag und damit pro Datenblock benötigt.

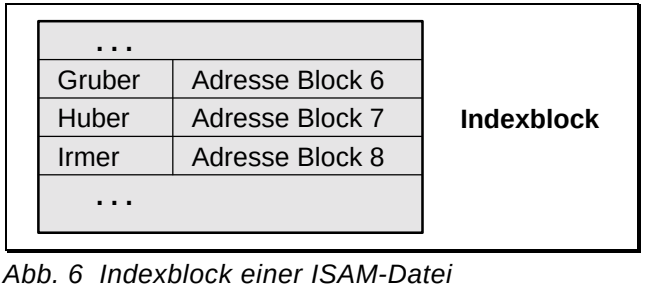


Abb. 6 Indexblock einer ISAM-Datei

Um nun Einträge zu bestimmten Namen zu finden, wird zunächst der Indexblock in den Arbeitsspeicher geladen. Dessen Einträge werden sequentiell durchsucht, bis die Adresse des gewünschten Datenblocks gefunden ist. Anschließend wird dieser Datenblock von der Platte eingelesen. Auch dieser Datenblock wird sequentiell bis zum gewünschten Eintrag durchsucht.

Vollziehen wir diese Suche nochmals an einem Beispiel nach: Wir suchen Daten zu Frau *Hübsch*. Der Indexblock wird eingelesen und durchsucht. Der Name *Hübsch* liegt alphabetisch zwischen *Huber* und *Irmer*. Folglich liegt *Hübsch* im gleichen Datenblock wie *Huber*. Dieser Datenblock wird eingelesen, und beim sequentiellen Suchen wird der Name *Hübsch* bereits an zweiter Stelle gefunden. Dieses Vorgehen gilt für alle zu suchenden Namen. Immer sind nur genau zwei Einlesevorgänge erforderlich. Dieses schematische Vorgehen bei der Suche ist typisch für ISAM und ist grafisch in Abb. 7 aufgezeigt.

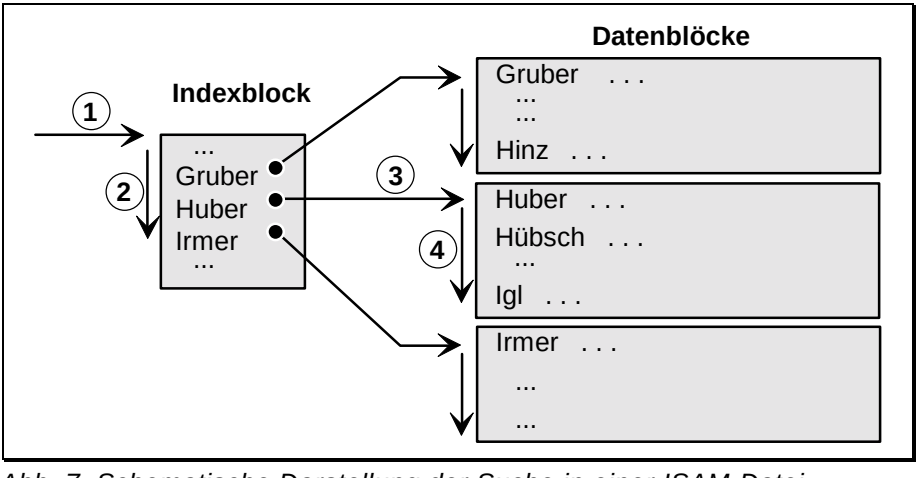


Abb. 7 Schematische Darstellung der Suche in einer ISAM-Datei

Doch nicht nur die Suche ist sehr schnell, auch das Ändern, Löschen und Einfügen ist meist relativ einfach. Beginnen wir mit dem Löschen von Einträgen:

Zum Löschen eines Eintrags werden die entsprechenden Daten gesucht und dann einfach entfernt. Die folgenden Einträge des aktuellen Datenblocks werden nachgerückt. Die dadurch entstehende Lücke am Ende des Blocks wird in Kauf genommen. Es handelt sich beim Löschen also um einen lokalen Vorgang, nur ein Datenblock wird manipuliert. Zusätzlich zum Einlesen beim Suchen ist nur eine einzige Ausgabe zum Schreiben des geänderten Blocks auf Platte erforderlich. Im Spezialfall des Löschens des ersten Eintrags in einem

Block muss auch der dazugehörige Indexblock angepasst werden. Sollte darüberhinaus das zu löschende Element der einzige und damit letzte Eintrag im Block gewesen sein, so wird der Datenblock einfach freigegeben und der dazugehörige Eintrag im Indexblock gelöscht (Nachrücken der Folgeelemente im Indexblock). In beiden letztgenannten Fällen ist demnach auch ein Schreiben des Indexblocks erforderlich.

Das Einfügen oder Ändern eines Eintrags ist immer dann einfach, falls für diesen Vorgang noch genügend freier Platz im Datenblock existiert. In diesem Fall werden die nachfolgenden Einträge im Block nach hinten verschoben, und der gewünschte Eintrag wird vorgenommen. Reicht hingegen der freie Speicherplatz im Datenblock nicht aus, so muss ein neuer Datenblock angefordert werden. Anschließend werden die Daten im zu ändernden Datenblock auf beide Blöcke (bisheriger und neuer) gleichmäßig verteilt. Dieses Verfahren heißt Splitting.

Das Splitting-Verfahren ist in Abb. 8 schematisch dargestellt. Die Abbildung zeigt auf, wie aus einem vollen Datenblock zwei halbgefüllte Blöcke entstehen, wobei der zweite Block neu angelegt wird. Genaugenommen wird also nur die zweite Hälfte des bisherigen Datenblocks in den neuen kopiert und dann gelöscht. Dieses Splitting-Verfahren erfordert auch einen Neueintrag im Indexblock. Dort ist der Suchschlüssel des ersten Eintrags des neuen Datenblocks samt der Adresse dieses Blocks abzuspeichern. Auch hier erfolgt das Einfügen analog dem Vorgehen in einem Datenblock (Verschieben der nachfolgenden Einträge nach hinten).

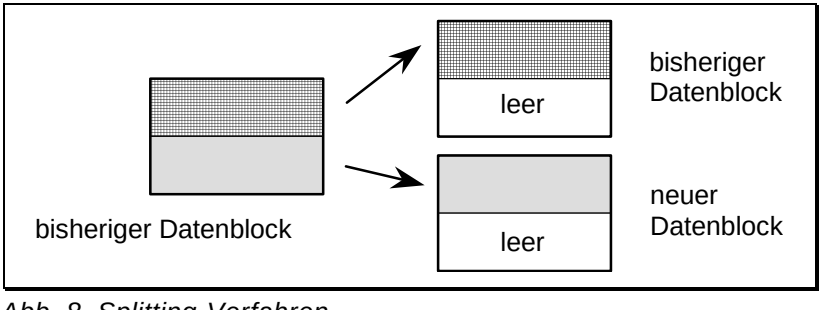


Abb. 8 Splitting-Verfahren

Ganz wichtig ist, dass dank dieses Splitting-Verfahrens auch das Einfügen von lokaler Natur ist. Es wird nur der betreffende Datenblock manipuliert, selten ein neuer Datenblock angelegt und selten der Indexblock geändert. Um das Splitting-Verfahren nicht zu häufig ausführen zu müssen, sind die einzelnen Blöcke im Schnitt meist nur zu 70 - 80% gefüllt. Dies ist ein guter Kom-

promiss zwischen Speicherbedarf und Laufzeit. Bei häufigem Löschen und Einfügen empfiehlt sich ein gelegentliches Reorganisieren des gesamten Datenbestands, um fast leere und zu volle Blöcke zu vermeiden.

Bisher sind wir immer von nur einem Indexblock ausgegangen. Wird der Datenbestand immer größer, so wird der freie Speicherplatz im Indexblock bald zu klein. In unserem Beispiel zu Abb. 6 sind in einem 1024-Byte-Block maximal 42 Indexeinträge möglich. Das Vorgehen zum Einfügen in einen vollen Indexblock ist völlig identisch zu dem Vorgehen bei Datenblöcken: Es erfolgt ein Aufsplitten. Auf diese Art und Weise kann mit der Zeit eine größere Anzahl von Indexblöcken entstehen.

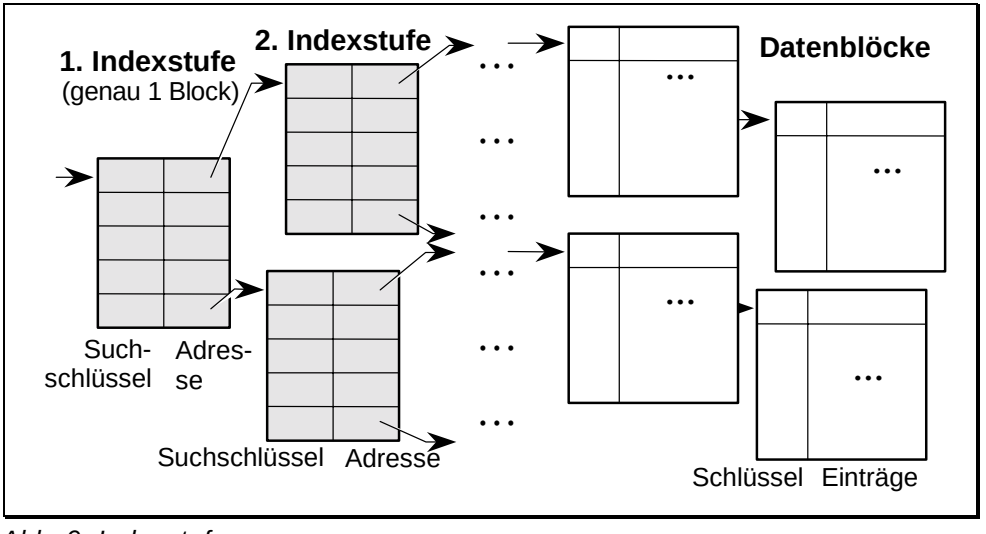


Abb. 9 Indexstufen

Das sequentielle Durchsuchen all dieser Indexblöcke wäre für sehr große Dateien zu aufwendig, insbesondere wären zu viele Ein-/Ausgaben erforderlich. Aus diesem Grund werden mehrere Indexblöcke durch einen vorgelagerten Indexblock verwaltet. Wir nennen diesen vorgelagerten Indexblock die erste Indexstufe. In diesem einzigen Indexblock der ersten Stufe werden die Indexblöcke der zweiten Indexstufe verwaltet. Bei sehr großen Datenbeständen können auch drei oder mehr Indexstufen existieren. Die letzte Indexstufe schließlich verwaltet die eigentlichen Datenblöcke. Es liegt eine Baumstruktur vor, der Einsprung zum Suchen erfolgt immer über den Indexblock der ersten Stufe. Diese Baumstruktur ist in Abb. 9 gut erkennbar.

Zum Suchen eines beliebigen Eintrags einer ISAM-Datei mit n Indexstufen benötigen wir also genau n+1 Einlesevorgänge (falls kein Indexblock bereits

im Arbeitsspeicher zur Verfügung steht). Das Löschen oder Einfügen in einem Datenblock schlägt sich nur extrem selten auf das Ändern aller vorgelagerten Indexstufen durch. Wenn überhaupt, so ist meist nur der Block der letzten Indexstufe betroffen. Dass wir ganz allgemein mit ganz wenigen Indexstufen auskommen, wollen wir an unserem obigen Beispiel demonstrieren:

Bei einer Blockgröße von 2048 Bytes, einer Datengröße von 116 Bytes und einer Indexeintragsgröße von 24 Bytes ergibt sich unter Berücksichtigung eines Füllgrads von ca. 75% folgende mittlere Anzahl von Einträgen: im Indexblock 64 und im Datenblock 13 Einträge. Dies ermöglicht mit nur wenigen Indexstufen eine große Anzahl von zu verwaltenden Daten. Im Einzelnen ergibt sich im Mittel:

1 Indexstufe	$\Rightarrow$	64 Blöcke	$\Rightarrow$	$64 \cdot 13 = 830$ Einträge	$\Rightarrow$	97 kBytes Daten
2 Indexstufen	$\Rightarrow$	64^2 Blöcke	$\Rightarrow$	53000 Einträge	$\Rightarrow$	6 MBytes Daten
3 Indexstufen					$\Rightarrow$	400 MBytes Daten
4 Indexstufen					$\Rightarrow$	25 GBytes Daten

Mit nur vier Indexstufen lassen sich viele GBytes an Daten effizient verwalten. Um eine beliebige Information zu finden, benötigen wir in diesem Fall genau fünf Plattenzugriffe (vier Indexzugriffe und ein Zugriff auf die gewünschten Daten). Dieses Beispiel stellt die Leistungsfähigkeit von ISAM-Dateien eindrucksvoll unter Beweis. In Tab. 5 sind die wichtigsten Leistungsdaten der ISAM-Dateien zusammengefasst.

Tab. 5 Zusammenfassung der ISAM-Datenstruktur

	ISAM-Datenstruktur
Speicherung	adressiert
Speicher	adressierbar
Speicherverbrauch	noch gering (+), im ungünstigen Fall zufriedenstellend (o)
Zugriffsdauer	kurz, über Indexstufen (+)
Änderungsaufwand	lokal, nur manchmal auch Indexstufe betroffen (+)
Einsatzmöglichkeit	universell auf adressierbarem Speicher, bei häufigen Änderungen ist gelegentliches Reorganisieren erforderlich
Komplexität	sehr hoch

1.7 Hash-Verfahren

Der Zugang zum Hash-Verfahren ist konträr zu dem zu ISAM-Dateien. Bei letzteren wurden die Vorteile der verketteten Listen und der Binären Suche zu einer Baumstruktur miteinander verknüpft; weiter wurde die Blockstruktur bei adressierbaren externen Medien ausgenutzt. Demgegenüber greift das Hash-Verfahren auf die Vorteile der Direktadressierung zu. Hier waren allerdings Namen als Suchschlüssel völlig ungeeignet (siehe Unterabschnitt 1.3.2). Das Ziel ist nun, die Direktadressierung so zu erweitern, dass auch Namen als Suchschlüssel verwendet werden können. Beginnen wir folglich mit Details zur Direktadressierung, um daraus das Hash-Verfahren abzuleiten.

Sei S die Menge aller möglichen Suchschlüssel und A die Menge aller zur Verfügung stehenden Speicheradressen. Dann liegt bei der Direktadressierung eine eineindeutige (bijektive) Funktion f mit

$$f: S \rightarrow A$$

vor, die jedem Element aus S eine eindeutige Adresse in der Menge A zuordnet. Wegen der Eineindeutigkeit der Funktion f müssen mindestens so viele Adressen existieren, wie es Elemente in der Menge S gibt. Dies ist im Terminkalender-Beispiel mit 365 bis 366 Kalendertagen kein Problem: Es wird Speicherplatz für 366 Einträge reserviert. Anders verhält es sich mit Dateien, deren Suchbegriff ein Name ist. Wegen der enormen Anzahl der möglichen Namen, wir haben in Unterabschnitt 1.3.2 von 26^{20} Kombinationen gesprochen, müssten auch so viele Speicheradressen existieren. Um auch für diesen sehr wichtigen Anwendungsfall die Direktadressierung anwenden zu können, muss deshalb die Eineindeutigkeit der Funktion f aufgegeben werden. Wir sind damit beim Hashverfahren angekommen, dessen formale Definition im Folgenden angegeben ist:

Definition (Hash-Verfahren)

☞ Gegeben sei eine Funktion h: $S \rightarrow A$, die jedem Suchschlüssel $s \in S$ eine (nicht notwendigerweise eindeutige) Adresse $a \in A$ zuordnet. Sei $E \subset S$ die Menge aller tatsächlich vorkommenden Suchschlüssel. Dann heißt eine Funktion h, eingeschränkt auf die Menge E, eine Hashfunktion zur Menge E:

$$h: E \subset S \rightarrow A \quad .$$

Die Vorgehensweise des Hash-Verfahrens ist damit vorgezeichnet. Jeder Eintrag wird an der Adresse abgespeichert, die die Hashfunktion *h* ermittelt. Es versteht sich von selbst, dass zumindest so viele Speicherplätze existieren, wie Einträge vorhanden sind. Auch die Suche eines gespeicherten Eintrags gestaltet sich einfach: Durch Anwenden der Hashfunktion *h* auf den Suchbegriff erhalten wir die gewünschte Speicheradresse.

Leider haben wir in unserer Euphorie fast vergessen, dass die Funktion *h* auf der Menge *S* nicht notwendigerweise eineindeutig ist. Das Gleiche gilt damit auch für die Einschränkung auf die Menge *E*. Es kann also vorkommen, dass mehrere Einträge mittels der Hashfunktion auf die gleiche Adresse gelegt werden. Dies erfordert eine Ausnahmebehandlung: Wird versucht, einen Eintrag auf einen bereits belegten Speicherplatz zu legen, so wird mittels einer weiteren Abbildung h_2, der Hashfunktion 2. Ordnung, eine Ersatzadresse ermittelt.

Wir zeigen dies an einem Beispiel auf. Gegeben sei eine Datei mit insgesamt 600 Einträgen à 100 Bytes. In diesen 100 Bytes sei der Suchschlüssel, bestehend aus einem 20-stelligen Namen, enthalten. Zum Abspeichern steht eine Datei mit 100 kBytes Größe zur Verfügung. Wir können demnach bis zu 1000 Einträge unterbringen. Als Hashfunktion wählen wir beispielsweise

$$\mathrm{h}(name) = 100 \cdot \left(\left(\sum_{i=1}^{20} i \cdot \mathrm{ord}\left(name[i]\right) \right) \bmod 1000 \right) \quad .$$

Hier ist *mod* der Modulooperator und *ord* die Ordnungsfunktion, die jedem Zeichen den Wert des dazugehörigen Zeichencodes zuordnet. Die innere Klammer besteht aus einer Summe und ergibt einen relativ willkürlichen Wert. Selbstverständlich sind auch andere Algorithmen denkbar, die eine vom Namen abhängige Zahl liefern. Da die Summe den Bereich zwischen 0 und 1000 leicht überschreitet, wird der Modulooperator gewählt, um eine gültige Speicheradresse zu erhalten. Der Faktor 100 am Anfang gibt die Größe jedes einzelnen Eintrags wieder.

Dieses einfache Beispiel zeigt bereits die typische Arbeitsweise einer Hashfunktion auf: Mit Hilfe der Modulofunktion zerhackt die Hashfunktion den meist sehr großen Bildbereich in viele gleich große Teile und überlagert diese. Die Größe der einzelnen Teile wird dabei genau der Menge der zur Verfügung stehenden Adressen angepasst. Übrigens hat die Hashfunktion von diesem Vorgehen auch ihren Namen: Das englische Verb *to hash* hat die Bedeutung

zerhacken. Auch wird offensichtlich, dass, wie schon bei der Direktadressierung gesehen, jeder Eintrag gleich groß sein muss.

Obige Funktion garantiert keine Eindeutigkeit der 600 Namen in den 1000 Adressen. Aus diesem Grund wird bei jeder ermittelten Adresse überprüft, ob bereits ein Eintrag abgespeichert ist. Ist dies der Fall, so wird zu dieser Adresse noch ein Versatz hinzugerechnet. Die Versatzfunktion h_2 könnte im einfachsten Fall etwa lauten:

$$h_2(adr) = 100 \cdot \left((adr / 100 + 43) \bmod 1000\right) \quad .$$

Zu jeder Adresse *adr* wird die um 43 Plätze versetzte Adresse ermittelt. Sollte auch dieser Platz belegt sein, so wird h_2 erneut angewendet und so fort.

Es sei nicht verschwiegen, dass die beiden Funktionen h und h_2 in unserem Beispiel nicht optimal gewählt wurden. Insbesondere h_2 weist erhebliche Schwächen auf. Ein fester Versatz (in unserem Fall die willkürlich gewählte Zahl 43) ist grundsätzlich ungünstig. Wir wollen es an dieser Stelle aber dabei bewenden lassen und verweisen zur Vertiefung auf [OtWi93], [Wirt83] und [Mehl88]. Das Suchen, Einfügen und Löschen in einer Hashtabelle dürfte jetzt im Prinzip klar sein:

Zum Suchen eines Eintrags in einer Hashtabelle wird auf den Suchbegriff die Hashfunktion angewendet. An der dabei ermittelten Speicheradresse wird der dortige Schlüssel mit unserem Suchbegriff verglichen. Sind beide gleich, so ist der Eintrag gefunden. Wenn nicht, so wird mit der Hashfunktion zweiter Ordnung h_2 die Versatzadresse berechnet. Dies geschieht so lange, bis entweder der Eintrag gefunden wurde, oder die ermittelte Adresse leer ist. Im letzteren Fall existiert der gesuchte Eintrag in der Hashtabelle nicht.

Das Einfügen eines Eintrags in eine Hashtabelle wurde bereits am Beispiel vorgeführt. Die Hashfunktion liefert die Speicheradresse. Ist diese Adresse bereits belegt, so wird so lange die Versatzfunktion h_2 angewendet, bis ein freier Speicherplatz gefunden wird. Hier wird dann der Eintrag abgespeichert.

Das Löschen eines Eintrags in einer Hashtabelle sieht ebenfalls einfach aus. Es wird das zu löschende Element gesucht (siehe oben), und dieses Element wird dann entfernt. Doch dieses Entfernen ist nicht so ohne weiteres erlaubt, da dadurch Suchwege zu anderen Einträgen mittels der Hashfunktion zweiter Ordnung zerstört werden können. Hier kommt es auf eine besonders geschickte Wahl dieser Funktion h_2 an. Weiter ist eine Struktur erforderlich, die sich diese Suchpfade merkt. Leider kompliziert dies den Algorithmus nicht unerheblich.

Die Erfahrung zeigt, dass der Hash-Algorithmus in vielen Anwendungsfällen hervorragend geeignet ist, die Suchzeiten erheblich zu reduzieren. Wir geben im Folgenden noch einen Überblick, in der auch unterschiedliche Einsätze im Vergleich zu ISAM aufgeführt sind:

- Aus wahrscheinlichkeitstheoretischen Berechnungen ist bekannt, dass es bei einem Füllungsgrad von 50% in 20 - 40% der Fälle zu Kollisionen kommt. Bei einem Füllungsgrad von 80% sind im Mittel bereits 5 Zugriffe (einer über h, vier über h_2) erforderlich. In der Praxis haben sich Füllstände der Tabellen von 40 - 60% bewährt. Dies ist ein guter Kompromiss zwischen Speicherverbrauch und Laufzeit.
- Das Verfahren ist effektiv und schnell, allerdings wird es bei großen Dateien wegen des nicht unerheblichen Speicherverbrauchs recht selten verwendet (ISAM wird dort bevorzugt). Außerdem ist ISAM optimaler auf die Formatierung der Festplatte ausgerichtet.
- Der einmal gewählte Adressbereich und damit die Dateigröße sind unveränderlich. Die Datei kann also nicht beliebig erweitert werden. Schon beim Anlegen der Hashtabelle muss die spätere Größe der Datei genau bekannt sein. Andernfalls sind aufwendige Reorganisationen erforderlich. Weiter muss freier Speicher als solcher gekennzeichnet sein, was beim Anlegen eine komplette Initiierung verlangt.

In Tab. 6 haben wir wieder die Ergebnisse zusammengefasst. Wir erkennen die insgesamt guten Eigenschaften dieser Zugriffsmethode, insbesondere fällt die extrem kurze Zugriffsdauer auf.

Tab. 6 Zusammenfassung der Hash-Datenstruktur

	Hash-Tabelle
Speicherung	direktadresssierend
Speicher	adressierbar
Speicherverbrauch	zufriedenstellend (o)
Zugriffsdauer	sehr kurz, da direktadressierend (++)
Änderungsaufwand	sehr gering, etwas höherer Löschaufwand (+)
Einsatzmöglichkeit	universell auf adressierbarem Speicher, nicht geeignet bei schwankender Dateigröße
Komplexität	hoch bis sehr hoch

Im Vergleich zu ISAM-Dateien sind die Zugriffe mit Hilfe von Hash-Datenstrukturen meist etwas schneller, dafür ist der Speicherplatzbedarf etwas höher. In der Praxis werden Hashtabellen für viele Spezialanwendungen verwendet, insbesondere bei Anwendungen im Arbeitsspeicher. Ein Beispiel sind auch invertierte Dateien. Wir werden diese im nächsten Abschnitt kennenlernen.

1.8 Primär- und Sekundärschlüssel

In den bisherigen Fällen der Datenspeicherung wurden die Daten immer nach einem Suchbegriff geordnet abgespeichert. Dies hat den Vorteil, dass entsprechende Einträge sehr schnell gefunden werden können, vorausgesetzt der Suchbegriff ist bekannt. Dass der Suchbegriff aber nicht immer das Sortierkriterium ist, wollen wir an einem einfachen Beispiel illustrieren:

Betrachten wir die KFZ-Kartei einer Kraftfahrzeugzulassungsstelle. Eine einfache Version ist in Tab. 7 dargestellt.

Tab. 7 KFZ-Kartei

KFZnr	Name	Vorname	GebDatum	Wohnort	KFZ_Typ	. . .
R-E 95	Meier	Fridolin	05.06.1966	Regensburg	...	...
L-ZM 6	Müller	Maria	13.03.1938	Leipzig	...	...
M-J 11	Huber	Walter	24.12.1944	München	...	...
...	...	...	...	...	...	...

Diese Kartei ist nach der KFZ-Nummer sortiert (in Tab. 7 dunkler dargestellt). In einer ISAM-Datei ist es nun ein Kinderspiel, zu einer gegebenen KFZ-Nummer den Besitzer dieses Fahrzeugs ausfindig zu machen. In Sekundenbruchteilen wird die gesuchte Nummer ausgegeben, unabhängig von der Karteigröße. Doch häufig kommen auch Anfragen nach Kraftfahrzeugen, die auf eine bestimmte Person zugelassen sind. In diesem Fall ist der Name der Suchbegriff. Jetzt muss der Computer die gesamte Kartei Name für Name sequentiell durchsuchen, um diese Frage beantworten zu können, was bei sehr großen Karteien Stunden dauern kann. Wegen dieser großen Laufzeitunterschiede in Abhängigkeit von den Suchbegriffen definieren wir:

Definition (Primär- und Sekundärschlüssel)

☞ Suchschlüssel, nach denen die Datei geordnet ist, heißen Primärschlüssel.

☞ Suchschlüssel, nach denen die Datei nicht geordnet ist, heißen Sekundärschlüssel.

Im obigen Beispiel ist die KFZ-Nummer also ein Primärschlüssel und der Name ein Sekundärschlüssel. Ganz allgemein gilt: Ist in einer geordneten Datei der Primärschlüssel bekannt, so ist die Suche des Eintrags immer schnell; die Suche mittels anderer Suchbegriffe (Sekundärschlüssel) muss grundsätzlich sequentiell erfolgen und ist daher langsam.

Wir können dies auch mathematisch formulieren: Geordnete Dateien (z.B. ISAM- oder Hash-Dateien) besitzen eine schnelle Zugriffsfunktion *f* vom Primärschlüssel auf die dazugehörigen Daten. Leider kann aus dieser Funktion *f* nicht ohne weiteres die Umkehrfunktion f^{-1} ermittelt werden.

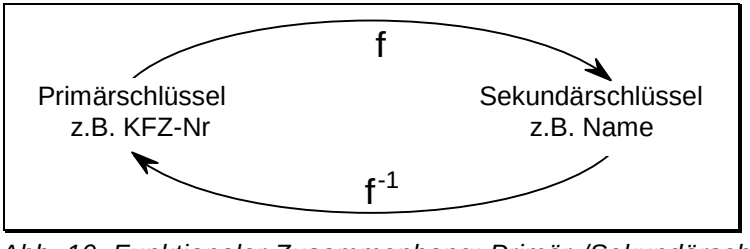


Abb. 10 Funktionaler Zusammenhang: Primär-/Sekundärschlüssel

Würden wir auch die Umkehrfunktion f^{-1} kennen, so wäre die Suche eines Eintrags mittels eines Sekundärschlüssels ebenfalls schnell: Wir könnten aus der Umkehrfunktion direkt den Primärschlüssel ermitteln und daraus mit Hilfe der bekannt schnellen Zugriffsfunktion *f* die gewünschten Daten finden.

Um also auch mittels Sekundärschlüssel schnell zugreifen zu können, benötigen wir die Umkehrfunktion f^{-1}. Leider ist es alles andere als trivial, zu Baum-, ISAM- oder Hash-Strukturen die Inverse einfach zu berechnen. Wir benötigen eine eigene Verwaltung für diese Umkehrfunktion. Diese zusätzliche Datenhaltung heißt invertierte Datei. Erst diese invertierten Dateien ermöglichen die schnelle Suche von Einträgen mittels Sekundärschlüsseln.

Invertierte Dateien sind Dateien, die nach dem Sekundärschlüssel geordnet sind und neben diesen Sekundärschlüsseln nur die dazugehörigen Primärschlüssel enthalten. Eine invertierte Datei kann wie jede andere Datei zusätz-

lich mittels ISAM oder Hash performant verwaltet werden. Dies garantiert schnellste Zugriffe.

Besitzt nun eine Datei eine invertierte Datei, so gilt folgendes:

- Ist der Primärschlüssel bekannt, so kann die Datei direkt gelesen werden. Eine Datenstruktur wie ISAM oder Hash ermöglicht extrem schnelle Zugriffe.
- Ist der Sekundärschlüssel bekannt, so wird zunächst die invertierte Datei gelesen, um den Primärschlüssel zu ermitteln. Wieder beschleunigen komplexe Datenstrukturen den Zugriff. Anschließend wird mit dem jetzt bekannten Primärschlüssel wie oben auf die eigentliche Nutzdatei zugegriffen.
- Existiert zu einem Sekundärschlüssel keine invertierte Datei, so muss die Datei sequentiell durchsucht werden.

Sind also die Zugriffsfunktion f und die dazugehörige Inverse f^{-1} bekannt, so lässt sich der Zugriff auf die Daten wie folgt darstellen:

$$Primärschlüssel \xrightarrow{f} Daten$$

$$Sekundärschlüssel \xrightarrow{f^{-1}} Primärschlüssel \xrightarrow{f} Daten$$

Es sei angemerkt, dass bisher nie die Eindeutigkeit des Primärschlüssels gefordert wurde. Bei Nichteindeutigkeit kann demnach f^{-1} mehrere Ergebnisse liefern. Ebenso kann es mehrere gleiche Sekundärschlüssel geben. In unserer obigen KFZ-Kartei können neben der eindeutigen KFZ-Nummer (Primärschlüssel) und dem nicht notwendigerweise eindeutigen Namen (Sekundärschlüssel) auch weitere Sekundärschlüssel existieren. Besonders gerne wird zum Beispiel das Geburtsdatum als weiteres Suchkriterium gewählt. Zu jedem Sekundärschlüssel benötigen wir allerdings eine eigene invertierte Datei.

Eine invertierte Datei enthält nur wenige Informationen: den Sekundär- und den dazugehörigen Primärschlüssel. Sie umfasst daher meist weniger als 10% der Originaldatei. Bei mehreren Sekundärschlüsseln summieren sich diese Werte allerdings auf. Wie wir leicht erkennen können, sind diese invertierten Dateien um so größer, je länger der Primärschlüssel ist. Dieser sollte daher möglichst klein gewählt werden. Dies ist nur einer der Gründe, warum Namen meist nicht als Primärschlüssel geeignet sind.

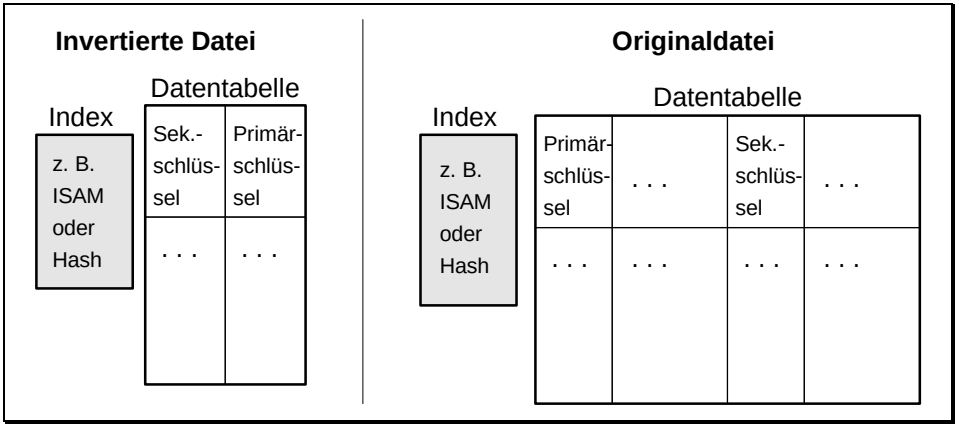


Abb. 11 Invertierte Dateien

Wie wir an Hand des Zugriffs über Sekundärschlüssel erkennen können, sind auch die Zugriffe über den Sekundärschlüssel sehr schnell. Sie dauern in der Regel nur knapp doppelt so lange wie Zugriffe mittels des Primärschlüssels.

Leider handeln wir uns mit den invertierten Dateien nicht nur Vorteile ein. Mit dem Ändern, Löschen oder Einfügen von Daten in der Originaldatei müssen auch alle invertierten Dateien angepasst werden. Wir befinden uns auf einer Gratwanderung zwischen Zugriffszeit, Speicherverbrauch und Fehleranfälligkeit. Einige Daten liegen jetzt mehrfach vor (außer in der Originaldatei auch in invertierten Dateien und eventuell in den Indexstufen), was bei geforderten schnellen Zugriffszeiten nicht zu vermeiden ist. Diese Mehrfachhaltung, auch Redundanz genannt, kann zu Inkonsistenzen in den Datenbeständen führen. Es ist immer möglich, dass wegen eines Rechnerabsturzes nur eines von mehreren Daten geändert wurde. Welche dieser sich widersprechenden Daten dann als gültig anzusehen ist, ist nicht einfach zu beantworten. Mit dem Lösen dieser Konsistenzprobleme, verursacht durch die Redundanz von Daten, wollen wir uns an dieser Stelle nicht weiter befassen. Wir kommen bei Datenbanken noch ausführlich darauf zurück.

1.9 Übungsaufgaben

1) Geben Sie Anwendungsbeispiele zur
   a) Direktadressierung,
   b) Speicherung in geordneter Reihenfolge.

2) Welche der vier einstufigen physischen Datenstrukturen kommt für ein elektronisches Telefonbuch höchstens in Frage? Welcher gewichtige Nachteil bleibt? Denken Sie bei der Beantwortung dieser Frage an die Hauptaufgabe eines Telefonbuchs.
3) Welcher Nachteil kann sich im Baumaufbau beim häufigen Ein- und Ausfügen in Bäumen einstellen? Geben Sie Möglichkeiten an, diesen Nachteil zu beseitigen. Welcher neue Nachteil stellt sich dann ein?
4) Ein Zugriff auf eine Magnetplatte benötigt etwa 10 Millisekunden, ein Zugriff auf den Arbeitsspeicher unter 100 Nanosekunden. Worauf sollte deshalb eine leistungsorientierte Speichertechnik zwingend achten? Kennen Sie eine solche Datenorganisation?
5) Wie reagiert die Datenorganisation ISAM, wenn ein Daten- oder Indexblock überläuft? Schildern Sie dieses Verfahren.
6) Betrachten Sie Abb. 7. Der Eintrag *Huber* soll gelöscht werden. Dazu muss er zunächst gesucht und dann entfernt werden. Gehen wir davon aus, dass der (einzige) Indexblock im Arbeitsspeicher steht. Wie viele Dateizugriffe sind dann insgesamt erforderlich? Wie viele davon sind schreibend?
7) Die Dateiorganisationen ISAM und Hash sind zwei sehr schnelle Zugriffsverfahren. Welches dieser beiden Verfahren ist bei großen Dateien (> 1 GBytes) vorzuziehen und warum?
8) Welchen Nachteil besitzt die Datenstruktur ISAM, wenn häufig ein- und ausgefügt wird? Wie kann man diesen Nachteil beseitigen?
9) In der Dateiorganisation Hash spielt die Hashfunktion zweiter Ordnung eine wichtige Rolle bei der Beseitigung von Häufungen von Einträgen. Überlegen Sie bessere als die im Buch angegebene Hashfunktion zweiter Ordnung.
10) Sind in der Datenorganisation die Primärschlüssel immer eindeutig? Sind dies die Sekundärschlüssel? Betrachten Sie dazu ein paar Beispiele.
11) Welchen wichtigen Zweck erfüllen Sekundärschlüssel in der Datenorganisation?
12) Die Suche mittels eines eindeutigen Suchbegriffs (z.B. Personalnummer) und mittels eines nichteindeutigen Begriffs (z.B. Name) ergeben ein grundsätzlich anderes Suchergebnis. Worin liegt der Unterschied? Worauf muss deshalb der Programmierer achten?
13) Betrachten wir eine Datei mit zwei Sekundärschlüsseln und die Verwaltung dieser Datei und deren Schlüssel mit Hilfe von Indizes (siehe auch Abb. 11). Geben Sie alle möglichen Redundanzen an einem Beispiel an.
14) Sie stellen fest, dass sich in der Organisation einer Datei die eigentliche Datei und die dazugehörige invertierte Datei in einigen Daten widersprechen. Welche der beiden Dateien werden Sie an die andere anpassen?

2 Übersicht über Datenbanken

Wir haben im letzten Kapitel wichtige Grundbegriffe zur Datenorganisation kennengelernt. Diese sind für das Verständnis von Datenbanken außerordentlich wichtig; denn auch in Datenbanken müssen die Daten physisch gespeichert und schnell wiederauffindbar sein. In diesem Kapitel werden wir eine kleine Übersicht zu Datenbanken geben und einige wichtige Begriffe kennenlernen. Zunächst werden wir Datenbanken definieren und damit die Abgrenzung zur Datenorganisation herstellen. Anschließend stellen wir einen Wunschkatalog zusammen, den wir von Datenbanken erfüllt haben wollen. Dieser Wunschkatalog lässt bereits die Mächtigkeit und Komplexität moderner Datenbanken erahnen. In weiteren Abschnitten bekommen wir einen ersten Eindruck zur Systemadministration von Datenbanken, und wir stellen die einzelnen Datenbankmodelle kurz vor. Zum Schluss gewinnen wir einen Überblick über Datenbankzugriffe und stoßen schließlich auf den enorm wichtigen Transaktionsbegriff. Der Leser, der bereits Grundkenntnisse in Datenbanken besitzt, kann dieses Kapitel überspringen.

2.1 Definition einer Datenbank

Im letzten Kapitel über Datenorganisationen haben wir Strukturen kennengelernt, mit denen wir problemlos große und größte Dateien effizient verwalten können. Wir haben gesehen, dass Daten sehr schnell über den Primär- oder die Sekundärschlüssel gefunden, und dass Änderungen ohne großen Aufwand rasch vollzogen werden können. Es stellt sich danach fast automatisch die Frage: Benötigen wir überhaupt Datenbanken? Und wir sind geneigt, *nein* zu sagen, da wir beliebig viele Daten performant verwalten können. Um diese Frage letztendlich doch mit einem eindeutigen *Ja* zu beantworten, müssen wir uns zuallererst darüber im klaren sein, was wir überhaupt unter einer Datenbank verstehen. Erst danach können wir die wichtigen Unterschiede zwischen Datenbanken und komplexen Datenorganisationen herausarbeiten.

Beginnen wir mit den aus dem letzten Kapitel bekannten Datenstrukturen. Gehen wir wieder an Hand eines Beispiels vor. Betrachten wir einen Mehrbenutzerbetrieb (Multi-User-Betrieb): Auf einen großen Datenbestand greifen

Benutzer von vielen Terminals aus mehr oder weniger gleichzeitig zu. Betrachten wir das Beispiel des Buchens von Flugreisen. Von vielen Reisebüros aus wird nach freien Plätzen gefragt, um diese, falls vorhanden und vom Kunden letztendlich gewünscht, zu reservieren. Im Augenblick der Buchung sind diese Plätze für alle andere Kunden als belegt markiert. Die Daten müssen also für alle gleichzeitig zugreifbar und aktuell sein.

Die gegebene Situation ist zusammengefasst folgende (siehe auch Abb. 12): Mehrere Benutzer greifen auf die gleichen Datenbestände zu. Dies geschieht in der Praxis ausschließlich über Anwendungsprogramme, da ansonsten der Anwender bei Direktzugriffen die Struktur und die Zugriffsbefehle kennen müsste; denken wir nur an Zugriffe über den Sekundärschlüssel bei ISAM-Dateien!

Analysieren wir diese Zugriffsweise. Das Anwendungsprogramm (und damit der Anwendungsprogrammierer) muss ebenso wie der direkt zugreifende Endbenutzer die physische Struktur aller Daten exakt kennen. Leider sind die Daten auf verschiedenen Hardware-Plattformen unterschiedlich abgespeichert. Das Anwendungsprogramm muss daher an jede Hardware angepasst sein. Ein unter UNIX laufendes Zugriffsprogramm muss nicht nur neu übersetzt werden, um auch unter MS-Windows oder MVS (Großrechnerbetriebssystem von IBM) ablauffähig zu sein, es muss mehr oder weniger komplett umgeschrieben werden.

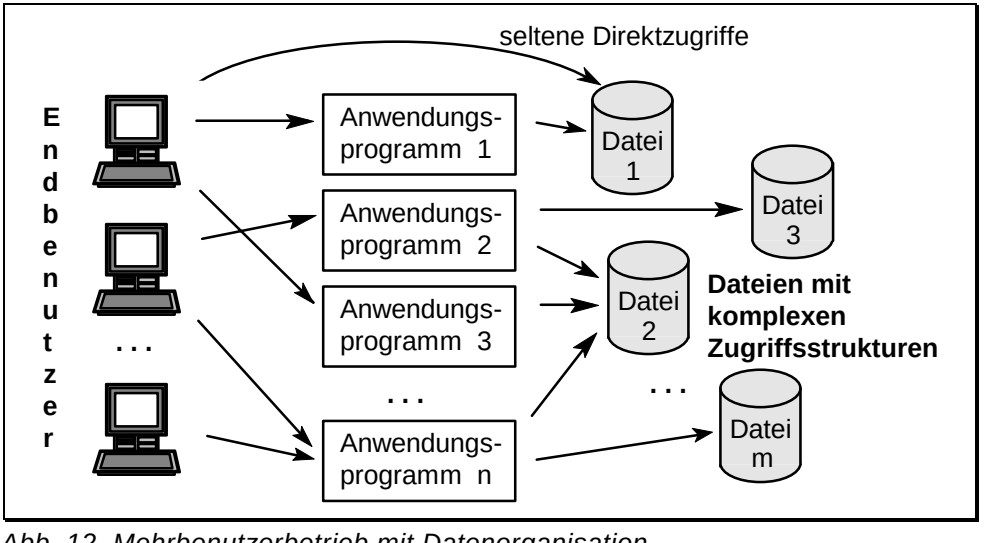


Abb. 12 Mehrbenutzerbetrieb mit Datenorganisation

Auch wenn gar kein Umstieg auf ein anderes System geplant ist, treten früher oder später Probleme auf; denn die Sammlung von riesigen Datenmengen geschieht nicht einfach aus Spaß, sondern um Daten auch noch nach Jahren oder Jahrzehnten wiederfinden zu können. Denken wir nur an Lebensversicherungen mit vertraglichen Laufzeiten von 25 oder mehr Jahren. In diesem Zeitraum ändern sich Computer und Speichermedien und damit auch die Schnittstellen zu den Daten erheblich. Jede dieser Änderungen erfordert aufwendige Korrekturen in all den Programmen, die diese Schnittstellen verwenden. In Abb. 12 sind dies alle Anwendungsprogramme!

Aber schon allein die Tatsache, dass jeder Anwendungsprogrammierer die physische Struktur aller Daten kennen muss, ist unbefriedigend. Hohe Kosten wegen einer aufwendigen Einarbeitung neuer Mitarbeiter und eine hohe Fehleranfälligkeit wegen der vielen unterschiedlichen und komplexen Strukturen sind die Folgen. Datenbanken versprechen hier Abhilfe.

Definition (Datenbank)

☞ Eine Datenbank ist eine Sammlung von Daten, die untereinander in einer logischen Beziehung stehen und von einem eigenen Datenbankverwaltungssystem (Database Management System, DBMS) verwaltet werden.

Der erste Teil der Definition ist nichts neues und sollte eigentlich selbstverständlich sein: Nicht zusammengehörige Daten werden gesondert verwaltet, etwa in separaten Datenbanken. Eine große Versicherungsgesellschaft wird beispielsweise seine Daten zur Lebensversicherung und zur KFZ-Versicherung gesondert in eigenen Datenbanken halten, was die Übersicht erheblich verbessert. Wesentlich bei Datenbanken ist der zweite Teil der Definition: Eine Datenbank impliziert eine Datenhaltung und eine dazugehörige Verwaltung dieser Daten. Wir können diese Erweiterung deutlich in Abb. 13 erkennen.

Neu gegenüber Abb. 12 ist die Abschottung der physischen Daten durch das Datenbankverwaltungssystem. Dieses DBMS besitzt eine logische Schnittstelle. Jeder Benutzer und damit auch jedes Anwendungsprogramm greifen ausschließlich über diese Schnittstelle auf die Daten zu. Das DBMS selbst sorgt für die physischen Zugriffe. Da die logische Schnittstelle eine „normale“ Programmierschnittstelle ist, ist ein Direktzugriff durch den Endbenutzer einfacher und damit nicht mehr so selten. Insbesondere werden aber auch die Anwendungsprogramme einfacher und damit weniger fehleranfällig. Auch der Leser von Datenbankbüchern wird erfreut darüber sein, dass er nur am Rande von der physischen Datenorganisation belästigt wird.

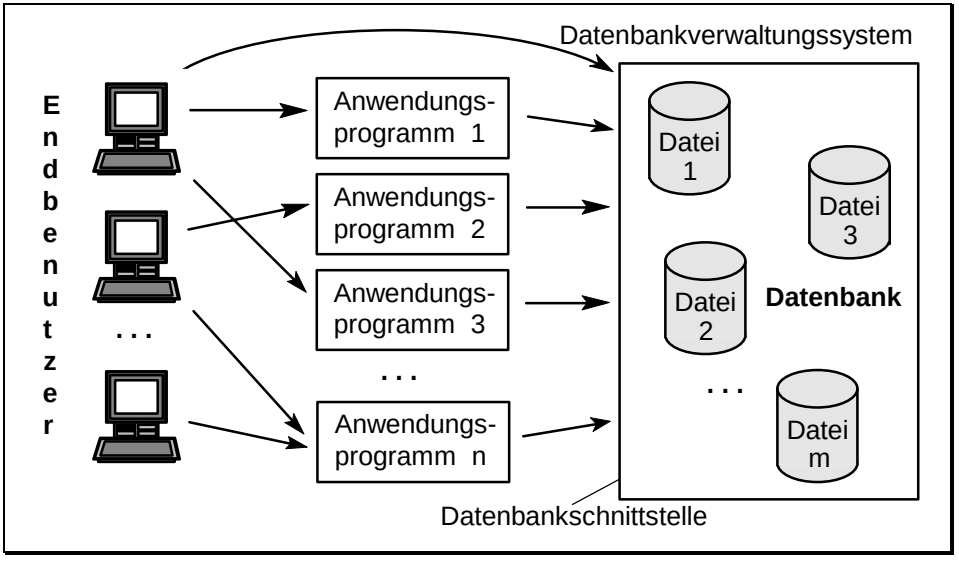


Abb. 13 Schematischer Zugriff auf Datenbanken

Zu einer wichtigen internationalen Datenbankschnittstelle hat sich in den letzten Jahren SQL entwickelt. SQL steht für Structured Query Language und ist eine Programmiersprache, mit deren Hilfe auf relationale Datenbanken zugegriffen werden kann. Sie wurde von E. F. Codd etwa ab 1970 bei IBM entwickelt, hieß zunächst SEQEL (daher auch die englische Aussprache „säquel" für SQL), und wurde dann 1979 von Oracle unter dem Namen SQL vorgestellt. Ihre Bedeutung gewann sie dadurch, dass sie in den Jahren 1986 bis 1992 einheitlich normiert wurde (1986: SQL-1, 1989: SQL-1+, 1992: SQL-2). Im Augenblick wird eine weitere Norm (SQL-3) mit zahlreichen Erweiterungen erarbeitet, vor allem in Richtung Objektorientierung. Heute wird SQL von den meisten Datenbanken unterstützt, unter anderem von Oracle, Informix, Ingres, Sybase, UDS, DB2, Adabas, dBase, Foxpro, Paradox, MS-Access.

SQL ist eine nicht-prozedurale Programmiersprache, mit der ein Anwender dem Datenbanksystem mitteilt, welche Daten er benötigt, und nicht, wie er an diese Daten gelangt. Dies vereinfacht die Programmierung erheblich. Wir werden bereits im Folgenden einfache SQL-Befehle kennenlernen. Ausführlich werden wir uns mit SQL in den Kapiteln 4, 6 und 9 beschäftigen.

Die Vorteile einer Datenbank sind sicherlich bereits deutlich zu Tage getreten. Sie seien hier nochmals zusammengefasst:

- Der Anwendungsprogrammierer und der Endbenutzer können über eine komfortable, mächtige und normierte Schnittstelle (SQL) des DBMS

auf die Daten der Datenbank zugreifen. Kenntnisse über die innere physische Struktur der Daten in der Datenbank sind nicht erforderlich. Die Umsetzung der logischen in physische Zugriffe übernimmt das DBMS selbst.

Beginnen wir mit einem ersten Beispiel. Gegeben sei ein Getränkelager mit einer Fülle unterschiedlicher Biersorten. Ein Ausschnitt über den Inhalt dieses Lagers enthält Tab. 8. Wie diese Daten physisch abgespeichert sind, ist für den Anwender nebensächlich, solange der Datenbankhersteller dafür sorgt, dass Daten schnell gefunden und geändert werden können. Sowohl die Suche (Retrieval) als auch das Ändern erfolgt mit einer Hochsprache. In SQL stehen dazu die Befehle *Select*, *Update*, *Insert* und *Delete* zur Verfügung.

Tab. 8 Getränkelager

Nr	Sorte	Hersteller	Typ	Anzahl
1	Hell	Lammsbräu	Träger	12
3	Roggen	Thurn und Taxis	Träger	10
4	Pils	Löwenbräu	Träger	22
8	Export	Löwenbräu	Fass	6
11	Weißbier	Paulaner	Träger	7
16	Hell	Spaten	6er Pack	5
20	Hell	Spaten	Träger	12
23	Hell	EKU	Fass	4
24	Starkbier	Paulaner	Träger	4
26	Dunkel	Kneitinger	Träger	8
28	Märzen	Hofbräu	Träger	3
33	Leichtes Weizen	Lammsbräu	Träger	6
36	Alkoholfreies Pils	Löwenbräu	6er Pack	5
39	Weißbier	Erdinger	Träger	9
47	Pils	Bischofshof	Fass	3

Eine Zeile einer solchen Datenbanktabelle heißt Satz (im Englischen: record) oder Tupel, ein Spaltenelement wird meist als Feld oder Attribut bezeichnet. Die vorliegende Anordnung der Daten in Tabellenform wird nicht nur in relationalen Datenbanken verwendet. Wir kommen bei der Behandlung der einzelnen Datenbankmodelle darauf zurück.

Selbstverständlich wäre für die wenigen Einträge in Tab. 8 keine Datenbank erforderlich. Handelt es sich hier hingegen nur um einen kleinen Ausschnitt aus dem umfangreichen Sortiment eines Getränkehändlers, so ist bei jeder Be-

stellung nachzusehen, ob die entsprechende Biersorte vorrätig ist; und wenn ja, ist vom Bestand die bestellte Ware abzuziehen. Dies ist natürlich mit Papier und Bleistift möglich, aber zeitintensiv und fehleranfällig. Hat sich unser Getränkehändler hingegen bereits für eine SQL-Datenbank entschieden, so wird ihm beispielsweise die Anfrage nach einem guten Weißbier leicht fallen. Sie lautet in SQL:

```
SELECT  Sorte, Hersteller, Anzahl
 FROM   Getränkelager
 WHERE  Sorte = 'Weißbier' ;
```

Als Ergebnis wird in unserem Beispiel folgende Tabelle erscheinen:

Tab. 9 Weißbiere im Getränkelager

Sorte	Hersteller	Anzahl
Weißbier	Paulaner	7
Weißbier	Erdinger	9

Auch das Einfügen eines weiteren dunklen Biers, das Hinzufügen eines Trägers Paulaner Weißbier oder das Löschen des Bischofshof Pils sind schnell ausgeführt:

```
INSERT
 INTO    Getränkelager
 VALUES (43, 'Dunkel', 'Kaltenberg', 'Träger', 6) ;

UPDATE Getränkelager
 SET    Anzahl = Anzahl +1
 WHERE  Nr = 11 ;

DELETE
 FROM   Getränkelager
 WHERE  Nr = 47 ;
```

Die genaue Syntax dieser SQL-Befehle werden wir in Kapitel 4 kennenlernen. In den drei SQL-Befehlen wird jeweils in der Tabelle *Getränkelager* eine Änderung vorgenommen. Genauso einfach sind auch die Zugriffe auf große Datenbestände. Jedes Mal erhalten wir bei relationalen Datenbanken das Ergebnis von Abfragen in Tabellenform mitgeteilt, und Änderungen werden direkt in der Datenbank ausgeführt. Schon mit diesem Beispiel ist ersichtlich,

dass Anwendungsprogramme einfacher und übersichtlicher werden als ohne Verwendung von Datenbanken.

Gehen wir nun in unserem Gedankenbeispiel weiter. Wegen guter Umsätze baut der Getränkehändler sein Händlernetz aus. Jetzt benötigt er auch eine Kunden-, eine Personal- und eine Verkaufsdatei. Auch Rechnungen, Mahnungen, Statistiken, Finanzen und Steuerabrechungen wollen verwaltet werden. Schließlich sollen diese Daten auch von allen Zweigstellen aus zugreifbar sein. Dies wiederum erfordert Zugriffsschutz (Passwortkontrolle) und Parallelverarbeitungsfähigkeit. Jetzt erkennen wir langsam die Mächtigkeit und Notwendigkeit einer Datenbank, denn all diese genannten Punkte sollte eine leistungsfähige Datenbank „quasi automatisch“ unterstützen.

2.2 Anforderungen an eine Datenbank

Nachdem wir im letzten Abschnitt am Beispiel des Getränkehändlers schon aufzeigten, welche Anforderungen an eine Datenbank gestellt werden können, wollen wir in diesem Abschnitt ausführlich darauf eingehen. Wir überlegen uns, welche Probleme in Zusammenhang mit der Haltung und Verwaltung großer Datenbestände auftreten können, und fordern, dass diese Probleme mit modernen Datenbanksystemen mittels entsprechender Mechanismen gelöst werden. Beginnen wir mit allgemeinen Wünschen und Problemen, die auch von komplexen Datenstrukturen erfüllt werden:

- Sammlung logisch verbundener Daten:
  Diese schon fast triviale Anforderung schließt ein, dass die Daten auf mehrere Dateien verteilt sein können. Die Anforderung unterstreicht nochmals, dass Daten, die nicht miteinander in Verbindung stehen, auch getrennt zu verwalten sind. Dies vereinfacht den Aufbau von Datenbanken.
- Speicherung der Daten mit möglichst wenig Redundanz:
  Wir haben bereits in Kapitel 1 die Mehrfachhaltung von Daten angesprochen. Da diese Mehrfachhaltung mit Problemen wie Inkonsistenzen verbunden ist, würden wir gerne auf sie verzichten. Dies ist aber aus Laufzeitgründen in der Regel nicht immer möglich. Eine Datenbank sollte aber einen möglichst redundanzarmen Aufbau der Daten durch entsprechende Datenstrukturen unterstützen. Desweiteren sollte das Datenbankverwaltungssystem die nicht vermeidbaren Redundanzen

kennen und bei Änderungen eines dieser redundanten Daten auch alle davon abhängigen Daten automatisch anpassen. Das Ziel ist, den Benutzer nicht mit dem Problem der Redundanz zu belasten.
Über Redundanz im Zusammenhang mit schnellen Datenzugriffen wurde bereits im letzten Kapitel gesprochen. Ein Beispiel für Redundanz in der Datenhaltung ist gegeben, wenn der Getränkehändler neben der Lagertabelle auch eine Einkaufs- und Verkaufstabelle führt. Allein aus der Differenz der letzten beiden Tabellen lässt sich der Lagerbestand berechnen. Die Lagertabelle enthält demnach ausschließlich redundante Daten, trotzdem wird diese Tabelle gerne mitgeführt. Die Datenbank hat jetzt beim Einkauf für die ordentliche Buchung sowohl in der Lager- als auch in der Einkaufstabelle zu sorgen. Analoges gilt für den Verkauf. Wir werden in diesem Kapitel noch sehen, dass mit der Redundanz weitere Probleme auftauchen, die natürlich ebenfalls von Datenbanken gemeistert werden sollen.

- Abfragemöglichkeit und Änderbarkeit von Daten:
  Diese selbstverständliche Forderung schließt ein, dass das Abfragen und Ändern von Daten den Anforderungen entsprechend schnell ausgeführt wird. Die geforderte Ausführungsgeschwindigkeit liegt meist im Bereich des menschlichen Reaktionsvermögens. Die Folge ist, dass Antwortzeiten des Rechners von mehr als zwei Sekunden nicht gern gesehen werden. Da Abfragen sehr komplex sein können und sich nicht selten über mehrere Dateien erstrecken, wird eine große Datenbank intern mit invertierten Dateien und mehrstufigen Datenstrukturen arbeiten müssen, was leider die Redundanz erhöhen wird. Das DBMS hat für diese optimale physische Speicherung zu sorgen, der Benutzer will wieder nicht mit diesen Details belästigt werden.

Die bisher aufgeführten Punkte werden auch von komplexen Datenstrukturen erfüllt, falls diese mit einer entsprechenden Oberfläche versehen werden. Ein DBMS besitzt natürlich eine solche Oberfläche, die die physischen Strukturen dem Benutzer gegenüber zu verbergen sucht. Doch eine Datenbank muss noch erheblich mehr leisten. Unsere Wunschliste ist noch nicht zu Ende:

- Logische Unabhängigkeit der Daten von ihrer physischen Struktur:
  Es genügt nicht, dass die Datenbank für die optimale Speicherung der Daten sorgt, der Benutzer will darüberhinaus nicht einmal wissen, wie die Daten physisch abgespeichert sind. Natürlich muss der Benutzer die Daten kennen, aber es reicht ihm, wenn er über den logischen Auf-

bau dieser Daten Bescheid weiß. Die Folgerung ist, dass der Benutzer logisch auf die Daten zugreift, und die Datenbank dann selbständig diesen logischen Zugriff in den entsprechenden physischen umwandelt und ausführt.

Wenn wir das Beispiel unseres Getränkelagers betrachten, so könnte in der Tabelle die Herstellerspalte zwecks schnelleren Zugriffs mit einem Index (invertierte Datei) versehen werden. Die Zugriffssprache, die nur die logischen Zusammenhänge kennt, darf sich dadurch aber nicht ändern!

Diese Forderung entspricht in ihrer Bedeutung etwa dem Übergang von Assembler zu einer strukturierten Hochsprache wie C oder PASCAL. Sie ermöglicht zum Einen das schnellere und einfachere Programmieren, zum Anderen eröffnet es aber auch große Freiheiten für die Datenbank. Das DBMS kann intern optimieren, ohne die Anwenderschnittstelle zu ändern, die Daten können jederzeit umorganisiert und anders gespeichert werden, sei es mit Hash- oder ISAM- oder Baumstrukturen, sei es auf Band oder Platte. Es kann die Hardware, das Betriebssystem, das abgespeicherte Blockformat und vieles mehr geändert werden, ohne dass dies an der Anwendungsprogrammierschnittstelle sichtbar würde.

- Zugriffsschutz:

  Betriebssysteme kennen nur den Schutz von Dateien, nicht den Schutz einzelner Daten innerhalb einer Datei. Dieser interne Schutz ist bei sensiblen Daten wie den Personaldaten aber erforderlich. Zum Einen soll die Lohnbuchhaltung auf die Adresse und die Kontonummer von Mitarbeitern zugreifen können, andererseits soll es der Firmenleitung vorbehalten sein, Einblicke in die Gehalts- und Beurteilungsdaten zu bekommen, ganz zu schweigen vom Ändern dieser letztgenannten sensiblen Daten.

  Die Datenbanken haben daher für den Zugriffsschutz zu sorgen, sie verwalten die Zugangskennungen und Passwörter beim Zugang zur Datenbank und geben je nach Authorisierung nur die entsprechend erlaubten Teile (Sichten) der Datenbank frei. Das DBMS ermöglicht darüberhinaus, dass solche Zugangsberechtigungen jederzeit widerrufen oder neu eingerichtet werden können, was auch beinhaltet, dass die erlaubten Sichten umdefiniert werden können.

- Integrität:

  Integrität kommt von *integer* und bedeutet hier, dass alle gespeicherten Daten korrekt sein müssen. Integrität ist eine der Forderungen, die als

selbstverständlich angesehen wird, aber in der Praxis nie ganz erfüllt ist. Natürlich wäre es schön, wenn zum Beispiel dieses Kapitel frei von Schreibfehlern wäre, ich gebe mich aber nicht der Illusion hin, dass dies tatsächlich zutrifft. Integrität umfasst einen sehr weiten Bereich, beginnend bei der physischen über die logische zur semantischen Integrität. Zum Einen müssen also die Daten physisch korrekt gespeichert sein, einschließlich aller invertierter Dateien und Indizes. Weiter müssen die Daten logisch richtig sein, mehrfach vorhandene Daten müssen den gleichen Wert besitzen, Änderungen müssen sich auf alle redundanten Daten beziehen.

Auch wenn dies alles erfüllt ist, bleibt als Hauptschwachstelle die semantische Integrität, womit die Eingabe und damit indirekt der Mensch gemeint ist. Der Mensch ist keine Maschine, beim Einlesen von Daten geschehen Fehler (nobody is perfect). Ein gutes DBMS sollte aber durch Plausibilitätskontrollen zumindest die gröbsten Fehler von vornherein entdecken können. Eine wöchentliche Arbeitszeit von 400 statt 40 Stunden ist beispielsweise nicht möglich und sollte daher als Eingabe grundsätzlich abgewiesen werden.

- Mehrfachzugriff (Concurrency):
  Auf die Forderung nach Datenzugriffen von mehreren Rechnern oder Terminals aus kann in der Regel heute nicht mehr verzichtet werden. Denken wir nur an Buchungssysteme, Versicherungen oder Banken. Immer muss es möglich sein, von verschiedenen Eingabeplätzen aus auf Daten mehr oder weniger gleichzeitig zugreifen zu können.

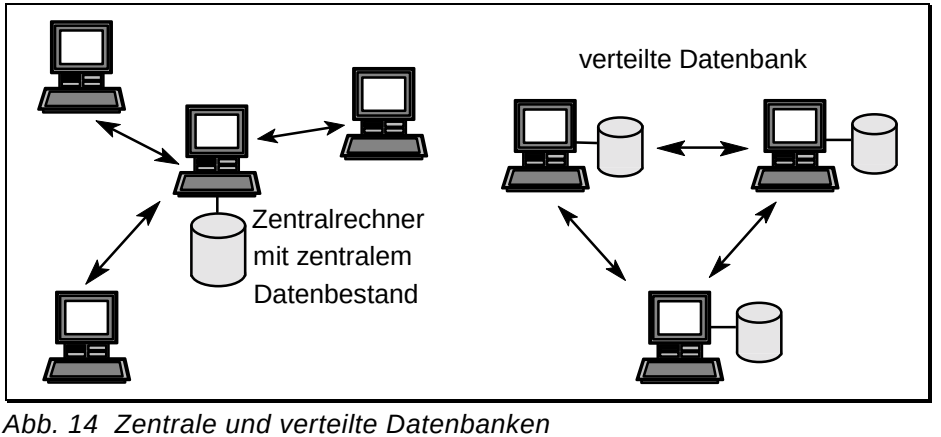


Abb. 14 Zentrale und verteilte Datenbanken

Bis heute werden die Daten meist zentral in einem Rechner gehalten, wobei von anderen Rechnern aus zugegriffen wird. Vereinzelt finden wir aber auch Anwendungen, wo die Daten auf mehrere Rechner verteilt werden, wobei jeder Rechner auf jedes Datum zugreifen kann (siehe Abb. 14).

Das Problem des Mehrfachzugriffs ist, dass bei konkurrierenden Zugriffen Konflikte auftreten können. Beispielsweise dürfen die letzten drei Träger Märzen nicht gleichzeitig an zwei Kunden verkauft werden. Wir fordern daher, dass die Datenbank dafür zu sorgen hat, Konflikte zu verhindern. Mehr noch, wir fordern, dass sich die Datenbank aus der Sicht des Benutzers so verhält, als sei er der einzige Benutzer. Die Lösung dieses Problems führt zum Begriff der Transaktion. Auf Transaktionen werden wir noch ausführlich zu sprechen kommen.

Mit den bisherigen Anforderungen ist unsere Wunschliste noch immer nicht vollständig abgearbeitet. Weitere Wünsche wollen wir jedoch nur noch kurz auflisten:

- Zuverlässigkeit bedeutet auch Sicherheit vor unerlaubten Zugriffen. Da solche unerlaubten Zugriffe nie gänzlich zu verhindern sind, sollten zumindest Zugriffe auf die Datenbank einschließlich den Angaben zum Benutzer protokolliert werden (Fachbegriff: Audit). Dadurch wäre zumindest eine nachträgliche Ermittlung unbefugter, meist krimineller Zugriffe möglich.
- Ausfallsicherheit bedeutet Sicherheit der Daten in allen nur erdenklichen Fällen, angefangen vom Rechnerausfall bis zur Zerstörung durch Feuer. Immer müssen alle Daten rekonstruiert werden können, Inkonsistenzen dürfen dabei nicht entstehen. Dies erfordert die unabhängige Protokollierung aller geänderten Daten der Datenbank.
- Kontrolle heißt, dass die Datenbank ständig Informationen über den augenblicklichen Zustand ausgibt, etwa die Anzahl der Benutzer, die Auslastung der Magnetplatten, die Auslastung des DBMS. Dies beinhaltet die Aufbereitung dieser Daten in Statistiken und natürlich die Abrechnung für die Benutzer der Datenbank.

Mit diesen letzten drei Wünschen beenden wir unsere inzwischen recht lange Wunschliste. Wir erkennen, dass eine Datenbank mehr ist als nur die Verwaltung von ein paar Daten. Heutige professionelle Datenbanken auf Großrechnern besitzen eine Größe von mehreren Millionen Zeilen Code. Aber

auch die mittleren Datenbanken für UNIX und sogar für PCs erfüllen heute schon alle oder zumindest fast alle obigen Punkte.

Zusammenfassend können wir sagen, dass eine Datenbank die Verschmelzung einer komplexen Datenorganisation mit einem umfangreichen Angebot an Dienstleistungen darstellt, und zwar mit dem Ziel, dem Anwender eine einfache, logische und trotzdem mächtige Schnittstelle anzubieten.

2.3 Der Datenbank-Administrator

Aus einigen obigen Punkten, insbesondere dem Zugriffsschutz und der Datenbankkontrolle, ist erkennbar, dass eine einzelne Person oder eine kleine Personengruppe, die Datenbank verwalten muss. Dieser Verwalter der Datenbank heißt Datenbank-Administrator. Zu seinen Aufgaben zählen:

- das Einrichten der Datenbank: Dazu gehört der vollständige Aufbau der Datenbank, das Erzeugen der Datenbanktabellen und das Festlegen der inneren Struktur (z.B. Indizes für schnelle Zugriffe). Dies geschieht in der Regel beim Erzeugen einer Datenbank.
- die Kontrolle der Datenbank: Hier handelt es sich um die Kontrolle im laufenden Betrieb. Neue Benutzer werden zugelassen, Schutzwörter werden vergeben, Sichten werden eingerichtet. Auf Meldungen des DBMS wird reagiert. Im Fehlerfall werden die erforderlichen Anweisungen an das DBMS gegeben. Im Bedarfsfall werden weitere Indizes erstellt.

Der Datenbank-Administrator besitzt für diese Verwaltungsaufgaben alle Zugriffsrechte zur Datenbank. Er ist ein Datenbankspezialist, kennt die verwendete Datenbanksoftware genau, weiß bei Performance-Problemen zu reagieren und leistet gegenüber dem Anwender gegebenenfalls technische Unterstützung.

Demgegenüber besitzt der „normale“ Benutzer keine Rechte zum Aufbau oder Ändern der internen logischen Datenbankstruktur. Diese klare Aufgabenteilung ist sehr zweckmäßig, das Interesse des Anwenders liegt schließlich vor allem an den in der Datenbank gespeicherten Daten und deren Zugriff. Der Anwender besitzt dazu Rechte zum Abfragen und Ändern dieser Daten, wobei diese Rechte vom Datenbank-Administrator vergeben werden. Der Administrator wird diese Rechte bei Bedarf auf bestimmte Teile der Datenbank ein-

schränken. Es existiert also eine genau definierte Aufgabenteilung zwischen Anwender und Administrator. Diese Aufgabenteilung tritt auch an der Datenbankschnittstelle deutlich hervor. Diese Schnittstelle lässt sich in drei Teile trennen:

- DDL (Data Description Language)
- Kontrollsprache (oft in DDL integriert)
- DML (Data Manipulation Language)

Die ersten beiden Teile werden vom Administrator verwendet. Sie dienen zur Beschreibung und Kontrolle der Datenbank. Mit Hilfe von DML werden Daten gespeichert, gelesen, geändert und gelöscht. Die DML-Befehle von SQL haben wir bereits kennengelernt. Sie hießen:

SELECT	zum Abfragen,
UPDATE	zum Ändern,
DELETE	zum Löschen und
INSERT	zum Einfügen von Daten in eine Datenbank.

Intensiv werden wir auf diese Zugriffsbefehle in Kapitel 4 eingehen. Einige typische DDL-Befehle in SQL sind:

CREATE TABLE	zum Erzeugen einer Tabelle,
DROP TABLE	zum Löschen einer Tabelle,
CREATE VIEW	zum Erzeugen einer Sicht,
CREATE INDEX	zum Erzeugen eines Index,
GRANT	zum Gewähren von Zugriffsrechten.

Weitere DDL-Befehle werden wir in den Kapiteln 6 und 8 kennenlernen, ebenso das Arbeiten mit diesen Befehlen. Darüberhinaus ist im Anhang B die Syntax der in diesem Buch behandelten Befehle komplett wiedergegeben.

2.4 Datenbankmodelle

Es ist bereits das Stichwort *relationale Datenbank* gefallen. Wir unterscheiden heute im wesentlichen vier Datenbankmodelle. Dies sind die bereits erwähnten relationalen, die modernen objektorientierten und die bereits in die Tage gekommenen hierarchischen und netzwerkartigen Datenbanken. Die Unterschiede dieser vier Modelle liegen in der Art des logischen Aufbaus der

Datenbank. In den letzten 15 Jahren haben die relationalen Datenbanken einen großen Aufschwung erlebt. Da aber ein Großteil der heutigen Datenbanken älter als 15 Jahre ist, sind die meisten Daten immer noch in hierarchischen oder insbesondere netzwerkartigen Datenbanken gespeichert. Die groben Unterschiede dieser vier Modelle wollen wir im Folgenden kurz vorstellen.

2.4.1 Relationale Datenbanken

Eine relationale Datenbank besteht ausschließlich aus Tabellen. Ein Zugriff erfolgt immer über diese Tabellen. Da leicht neue Tabellen hinzugefügt oder gelöscht werden können, sind spätere Änderungen des logischen Datenbankaufbaus relativ leicht möglich. Weiter sind Zugriffe auf Tabellen einfach zu programmieren, was zu der großen Beliebtheit dieses Datenbankmodells führte: Fast alle neuen Datenbanken sind relational aufgebaut.

Die Zusammenhänge zwischen den einzelnen Tabellen werden über Beziehungen hergestellt. Diese Beziehungen sind in den Tabellen mit abgespeichert. Der Aufbau von Datenbeständen über Tabellen und ihre Beziehungen zueinander sind mathematisch fundiert (im Relationenkalkül und in der relationalen Algebra, siehe Abschnitt 3.4 und [Date95]).

Die relationalen Datenbanken besitzen aber auch einige Nachteile. Zugriffe erfolgen oft über mehrere Tabellen, was längere Laufzeiten und eine hohe Anzahl von Ein-/Ausgaben zur Folge haben kann. Auch können einige Daten nur mit einer gewissen Redundanz in Tabellenform abgespeichert werden. Die Vor- und Nachteile sind in Tab. 10 kurz zusammengefasst:

Tab. 10 Zusammenfassung zu relationalen Datenbanken

	Relationale Datenbanken
Vorteile	leichte Änderbarkeit des Datenbankaufbaus, mathematisch fundiert, leicht programmierbar und verwaltbar
Nachteile	häufig viele Ein-/Ausgaben notwendig, erfordert eine hohe Rechnerleistung, erzeugt Redundanz

2.4.2 Objektorientierte Datenbanken

Eine objektorientierte Datenbank besteht ausschließlich aus Objekten. Ein Objekt ist entweder ein realer Gegenstand, etwa ein Flugzeug, eine Person oder ganz allgemein ein abstrakter Gegenstand, etwa eine Adresse, eine Rechnung oder eine Abteilung einer Firma.

Da viele Objekte auch in Tabellenform gespeichert werden können, werden objektorientierte Datenbanken häufig als eine Erweiterung relationaler Datenbanken angesehen. Dies trifft allerdings nur teilweise zu. Schließlich gehören zu einer objektorientierten Datenbank auch objektorientierte Ansätze wie Klassen, Datenkapselung oder Vererbung – Ansätze wie sie auch in der objektorientierten Programmierung Verwendung finden.

In der Praxis dürfte sich in den nächsten Jahren eine Mischform zwischen objektorientierten und relationalen Datenbanken durchsetzen: sogenannte objektrelationale Datenbanken. Diese Datenbanken sind eine echte Erweiterung der relationalen Datenbanken in Richtung Objektorientierung.

Objektorientierte und objektrelationale Datenbanken haben einen komplexeren Aufbau als relationale Datenbanken (fast beliebige Objekte statt einfacher Tabellen). Als Konsequenz müssen Designer und Anwendungsprogrammierer einen höheren Aufwand in Entwurf und Programmierung investieren. Auch die interne Verwaltung der Datenbank ist umfangreicher. Als Vorteil erhalten wir insbesondere bei technischen und multimedialen Anwendungen einen anschaulicheren Aufbau (komplexe Objekte müssen nicht zwangsweise auf Tabellen abgebildet werden). Dies kann erhebliche Laufzeitvorteile nach sich ziehen. Die Vor- und Nachteile sind kurz in Tab. 11 zusammengefasst. Eine übersichtliche Einführung finden wir in [Date95], eine kurze Zusammenfassung in Abschnitt 11.2.

Tab. 11 Objektorientierte Datenbanken

	Objektorientierte und -relationale Datenbanken
Vorteile	objektorientierter Aufbau, universell einsetzbar, noch relativ einfach programmierbar und verwaltbar, (teilweise) aufwärtskompatibel zu relationalen Datenbanken
Nachteile	relativ viele Ein-/Ausgaben notwendig, erfordert eine relativ hohe Rechnerleistung, teilweise recht komplexer Aufbau

2.4.3 Hierarchische und netzwerkartige Datenbanken

Die ältesten Datenbanken sind hierarchische Datenbanken. Der logische Aufbau dieser Datenbanken entspricht einer Baumstruktur. Der Zugriff erfolgt immer über die Wurzel in Richtung des gewünschten Knotens. Dies gewährleistet geringste Redundanz, da direkt über die Baumstruktur zugegriffen wird, und garantiert kürzeste Zugriffszeiten. Der Nachteil ist eine extrem hohe Inflexibilität gegenüber Änderungen in der Datenbankstruktur.

Aus diesem Grund wurde dieses Modell schon bald durch die netzwerkartige Datenbank ergänzt. Hier besteht der logische Aufbau aus Daten, die nicht mehr hierarchisch, sondern über ein beliebig aufgebautes Netz miteinander in Verbindung stehen. Dies erhöht die Flexibilität erheblich, allerdings auf Kosten der Komplexität des Aufbaus. Beide Modelle können aus heutiger Sicht nicht mehr voll befriedigen.

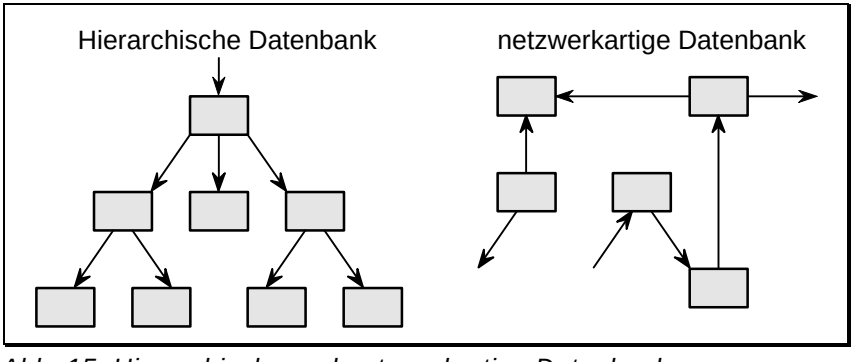


Abb. 15 Hierarchische und netzwerkartige Datenbanken

Der wichtigste Vertreter hierarchischer Datenbanken ist IMS von IBM, netzwerkartige Datenbanken sind IDMS (Computer Associates) und UDS (Siemens-Nixdorf). Einzelheiten können in [Gee77], [UDS83] und [UDS86] nachgelesen werden. Eine sehr schöne Darstellung finden wir auch in [Date95]. Fassen wir die Ergebnisse kurz zusammen:

Tab. 12 Hierarchische und netzwerkartige Datenbanken

	hierarchisch	netzwerkartig
Vorteile	sehr kurze Zugriffszeiten, minimale Redundanz	kurze Zugriffszeiten, geringe Redundanz
Nachteile	Strukturänderung kaum möglich, komplexe Programmierung	Strukturänderung nicht einfach, relativ komplexe Programmierung

2.5 Transaktionen

Wie wir bisher gesehen haben, lässt sich das Arbeiten mit Datenbanken in zwei Teile zerlegen: in das Erstellen und Verwalten der Datenbank und in das eigentliche Arbeiten mit den Daten der Datenbank. Das Erstellen und Verwalten ist notwendig, dient aber letztlich nur dazu, dass der Anwender auf die Datenbank zugreifen kann. Das Ermöglichen des Lesens und Manipulierens von Daten ist der eigentliche Sinn und Zweck einer Datenbank. Wir wollen daher jetzt aufzeigen, wie auf die Daten einer Datenbank generell zugegriffen wird. Wir unterscheiden drei verschiedene Zugriffsarten:

① Abfrage (Query): Es wird ein Ausschnitt der Datenbank ausgewählt und abgegrenzt. Anschließend wird der Inhalt dieses Ausschnitts gelesen. Die Datenbank selbst bleibt unverändert.
In SQL erfolgen Abfragen mit dem *Select*-Befehl. Im Beispiel

```
SELECT Spalten FROM Tabelle WHERE Bedingungen ;
```

sind die Angaben *Spalten*, *Tabelle* und *Bedingungen* Abgrenzungen der Datenbank. Als Ergebnis erhalten wir einen Ausschnitt aus der Datenbank.

② Mutation: Es wird ein Ausschnitt der Datenbank ausgewählt und abgegrenzt. Der Inhalt dieses Ausschnitts wird geändert, gelöscht, oder neuer Inhalt wird hinzugefügt. Der Datenbankinhalt wird dadurch verändert. Die SQL-Befehle zur Mutation sind *Update*, *Delete* und *Insert*.

③ Transaktion: Eine Transaktion ist eine konsistenzerhaltende Operation auf einer Datenbank. Eine Operation besteht aus einer beliebigen Anzahl von Abfragen und Mutationen. Unter konsistenzerhaltenden Operationen verstehen wir Operationen, die, ausgehend von konsistenten Datenbanken, diese Konsistenz erhalten.

Die Begriffe *Abfrage* und *Mutation* sollten keine Probleme bereiten. Wir unterscheiden sie deshalb, da Mutationen im Gegensatz zu Abfragen auch schreibend auf Teile der Datenbank zugreifen. Neu ist der Begriff *Transaktion*. Wir wollen die Notwendigkeit von Transaktionen an einem einfachen Beispiel aufzeigen:

Betrachten wir ein Geldinstitut mit mehreren Zweigstellen. Dieses Institut führt in ihrer Datenbank unter anderem auch die Tagessalden, also die Zahlungsein- und -ausgänge der Bank und der einzelnen Zweigstellen. Hebt nun ein Kunde 1000 DM von seinem Konto ab, so ist diese Mutation für sich keine Transaktion. Erst aus der zusätzlichen Anpassung des Saldos der Zweigstelle und des Saldos der Bank erhalten wir eine Transaktion. Waren also vor dieser Abbuchungs-Transaktion die Bedingungen

$$\sum \textit{Ein-/ Ausgänge der Zweigstelle} = \textit{Saldo der Zweigstelle}$$

$$\sum \textit{Salden aller Zweigstellen} = \textit{Saldo der Bank}$$

erfüllt, und damit die Datenbank in sich konsistent, so müssen diese Bedingungen auch nach der Transaktion gelten.

Konsistenz heißt die Freiheit von Widersprüchen innerhalb einer Datenbank. Konsistenz ist in Datenbanken enorm wichtig. Ist die Datenbank nach ihrer Erzeugung konsistent, und werden als Zugriffe auf die Daten dieser Datenbank ausschließlich Transaktionen zugelassen, so wird diese Datenbank per definitionem konsistent bleiben.

Wir erkennen daraus zweierlei: Erstens wird eine Datenbank um so schwieriger konsistent zu halten sein, je mehr Redundanz in dieser Datenbank vorkommt, da dadurch die logischen Zusammenhänge der Daten untereinander wachsen; zweitens sind alle Zugriffe in einer kommerziellen Datenbankanwendung Transaktionen.

Es sei noch angemerkt, dass eine Transaktion nicht nur im Normalbetrieb die Konsistenz erhalten muss, sondern auch im Fehlerfall; z.B. muss auch bei einem Rechnerabsturz während der Ausführung einer Transaktion für die Konsistenz der Daten gesorgt werden. Eine Transaktion muss daher immer entweder komplett oder gar nicht ausgeführt werden. Eine Transaktion ist demnach ein atomarer Zugriff! Die Garantie, dass eine Transaktion wirklich atomar abläuft, muss das DBMS übernehmen.

Wegen der Wichtigkeit von Transaktionen in einem Datenbanksystem werden die Begriffe

Datenbanksystem ⇔ Transaktionssystem

meist synonym verwendet. Wir werden auf Transaktionen in den Kapiteln 7 und 8 noch ausführlich zu sprechen kommen. Zunächst müssen wir uns aber

mit den elementaren Zugriffen und dem Aufbau von Datenbanken, insbesondere von relationalen Datenbanken, beschäftigen.

2.6 Übungsaufgaben

1) Welches sind die Vorteile einer Datenbank gegenüber der direkten Verwaltung von Daten? Welches sind die Nachteile?
2) Gehen wir davon aus, dass unser Getränkelager nicht mittels einer relationalen Datenbank, sondern mit einer ISAM-Datei verwaltet wird. Beschreiben Sie verbal, wie dann etwa die Abfrage nach allen Weißbiersorten aussieht. Beachten Sie, dass hier die physische Datenstruktur voll durchdringt.
3) Betrachten Sie eine Großdatenbank (z.B. Datenbank einer Versicherung oder eines Geldinstituts). Geben Sie zu allen Anforderungen aus Abschnitt 2.2 einzeln an, ob und wenn ja, warum diese notwendig sind.
4) Warum ist die Trennung zwischen Administrator und Anwendern (Benutzer) in Datenbanken so wichtig?
5) Wenn eine (kleinere) Datenbank im transaktionslosen Einbenutzerbetrieb gefahren wird, so muss die Konsistenz der Daten vom Anwender selbst kontrolliert werden. Wie könnte dies aussehen? Diskutieren Sie den Aufwand? Warum arbeiten heute trotzdem noch viele kleine Datenbanken ohne Transaktionsmechanismen?
6) Was ist eine relationale Datenbank? Wie unterscheidet sie sich von nichtrelationalen?
7) Geben Sie die Ergebnisse folgender SQL-Abfrage-Operationen zum Getränkelager aus:

a)
```
SELECT Sorte, Hersteller
FROM   Getränkelager
WHERE Typ = 'Fass';
```

b)
```
SELECT Sorte, Hersteller, Anzahl
FROM   Getränkelager
WHERE Anzahl < 4;
```

c)
```
SELECT Hersteller, Anzahl
FROM   Getränkelager
WHERE Sorte = 'Pils'
AND    Typ = 'Träger';
```

8) Geben Sie die Ergebnisse folgender SQL-Änderungs-Operationen zum Getränkelager aus:

a)
```
INSERT INTO Getränkelager
VALUES (18, 'Export', 'EKU', '6er Pack', 8);
```

b)
```
DELETE FROM Getränkelager
WHERE       Typ = 'Träger'
  AND       Anzahl < 5;
```

c)
```
UPDATE Getränkelager
SET    Anzahl = Anzahl + 2
WHERE Nr = 28
   OR  Nr = 47;
```

9) Schreiben Sie SQL-Anweisungen, die folgendes Ergebnis liefern:
   a) Geben Sie alle Sorten mit Hersteller an, die 6er-Packs vertreiben.
   b) Geben Sie alle Sorten an, die vom Hersteller *Löwenbräu* im Depot vorrätig sind.
   c) Entfernen Sie alle Sorten des Herstellers *Kneitinger*.
   d) Entfernen Sie 10 Träger Pils des Herstellers *Löwenbräu*.
   e) Fügen Sie die Biersorte *Dunkles Weißbier* der Firma *Schneider* mit der Nummer 10, dem Typ *Träger* und der Anzahl 6 hinzu.

3 Das Relationenmodell

Der logische Aufbau relationaler Datenbanken ist denkbar einfach: Sie bestehen ausschließlich aus Tabellen. Auch der Zugriff auf die Daten relationaler Datenbanken erfolgt ausschließlich über diese Tabellen. Für den Laien sieht es nun fast trivial aus, eine relationale Datenbank zu erzeugen, darin Daten zu speichern und bei Bedarf abzurufen. Solange wir nur kleine Datenbanken erstellen, die mehr zum Spielen als zum ernsten Arbeiten gedacht sind, mag dies auch zutreffen. In der Praxis bestehen relationale Datenbanken aber leicht aus einhundert bis über eintausend Tabellen. Dies erfordert allein aus Übersichtsgründen einen sauberen Aufbau. Wir dürfen aber weitere wichtige Gesichtspunkte beim Arbeiten mit Datenbanken nicht außer acht lassen:

- geringe Redundanz,
- gute Handhabbarkeit,
- einfache Zugriffe über möglichst wenige Tabellen und
- Sicherstellung von Konsistenz und Integrität.

Die Berücksichtigung dieser Merkmale erfordert fundierte Kenntnisse über den logischen Aufbau und das Design von Tabellen. Mit dem logischen Aufbau einzelner Tabellen und ihrer Beziehungen untereinander werden wir uns in diesem Kapitel ausführlich beschäftigen. Mit dem Design einer ganzen Datenbank befassen wir uns in Kapitel 5.

Zur Motivation beginnen wir dieses Kapitel mit einem Negativbeispiel, einer nichtnachahmenswerten Tabelle. Anschließend gehen wir dann ausführlich auf die relationalen Datenstrukturen ein und werden dann relationale Datenbanken exakt definieren. Es folgt ein Abschnitt über Integritätsregeln, die zur Sicherstellung der Integrität in Datenbanken erfüllt sein müssen, und die deshalb den Aufbau von Tabellen und die Beziehungen zwischen den Tabellen entscheidend mitprägen. Wir beenden dieses Kapitel mit einem theoretischen Abschnitt zur relationalen Algebra, der mathematischen Grundlage des Zugriffs auf relationale Datenbanken.

3.1 Beispiel zu relationalen Datenbanken

In diesem Abschnitt zeigen wir auf, dass es alles andere als trivial ist, Tabellen optimal zu wählen. Zwar gewähren uns relationale Datenbanken hier große Freiheiten, doch die Komplexität des späteren Zugriffs hängt wesentlich vom Aufbau eben dieser Tabellen ab. Demonstrieren wir dies an einem einfachen Negativbeispiel, der Datenbanktabelle *VerkaeuferProdukt* aus Tab. 13.

Auf den ersten Blick scheint die Tabelle völlig in Ordnung zu sein. Sie enthält alle wichtigen, benötigten Daten: die Daten des Verkäufers und die verkauften Produkte mit Wertangaben. Herr Meier hat zum Beispiel Waschmaschinen im Wert von 11000 DM, Herde im Wert von 5000 DM und Kühlschränke im Wert von 1000 DM verkauft.

Bei näherer Betrachtung fallen allerdings Redundanzen auf. Der Name, die Postleitzahl und die Adresse der Verkäufer kommen mehrfach vor. Natürlich muss jetzt die Änderung einer dieser Daten in allen Zeilen durchgeführt werden, in denen der Verkäufer aufgeführt ist. Aus einer einzigen beabsichtigten Mutation werden mehrere Mutationen. Um Inkonsistenzen zu vermeiden, müssen diese Mutationen innerhalb einer Transaktion ausgeführt werden. Wir haben uns deshalb mit dieser gutgemeinten Tabelle Redundanzprobleme eingehandelt. Redundanz birgt nicht nur in diesem Beispiel immer eine Gefahrenquelle für Fehler und Inkonsistenzen.

Tab. 13 VerkaeuferProdukt

Verk_Nr	Verk_Name	PLZ	Verk_Adresse	Produktname	Umsatz
V1	Meier	80075	München	Waschmaschine	11000
V1	Meier	80075	München	Herd	5000
V1	Meier	80075	München	Kühlschrank	1000
V2	Schneider	70038	Stuttgart	Herd	4000
V2	Schneider	70038	Stuttgart	Kühlschrank	3000
V3	Müller	10183	Berlin	Staubsauger	1000

Dieses harmlose Beispiel enthält aber noch ein weiteres schwerwiegendes Problem: Nehmen wir an, die Firma entschließt sich, den Artikel *Staubsauger* aus dem Sortiment zu nehmen. Dieser Artikel soll daher in der Datenbank nicht mehr auftauchen. Es werden deshalb alle Zeilen, die den Artikel *Staubsauger* enthalten, aus der Datenbank entfernt. Leider verliert damit die Datenbank auch alle Informationen zum Verkäufer *Müller*! Die Tabelle *VerkaeuferProdukt* ist folglich alles andere als nachahmenswert.

An diesem Beispiel wird klar, dass zur Erstellung von Tabellen gewisse Grundkenntnisse zu relationalen Datenbanken erforderlich sind. Bereits 1970 begann E. F. Codd mit der Untersuchung des optimalen Aufbaus von Tabellen. Seine und andere Arbeiten auf diesem Gebiet führten zu mathematisch fundierten Theorien (relationale Algebra, Relationenkalkül, Normalformenlehre). Diese Theorien zeigen, dass Tabellen so aufgebaut werden können, dass sowohl Redundanzen in den Daten vermieden als auch Einzeldaten isoliert gelöscht werden können. Wir werden uns im Folgenden mit diesen Theorien eingehend beschäftigen.

3.2 Relationale Datenstrukturen

Bevor wir uns mit den theoretischen Grundlagen relationaler Datenbanken näher befassen, benötigen wir einige Begriffe. Betrachten wir dazu Tab. 14, in der formale relationale Ausdrücke den mehr informellen alltäglichen Ausdrücken gegenübergestellt sind.

Tab. 14 Begriffe in relationalen Datenstrukturen

Formale relationale Ausdrücke	Informelle Ausdrucksweise
Relation	Tabelle
Tupel	eine Zeile (Reihe) einer Tabelle
Kardinalität	Anzahl der Zeilen einer Tabelle
Attribut	eine Spalte (Feld) einer Tabelle
Grad	Anzahl der Spalten einer Tabelle
Primärschlüssel	eindeutiger Bezeichner
Gebiet	Menge aller möglichen Werte

Wir werden im Folgenden meist die formalen relationalen Ausdrücke verwenden, insbesondere benutzen wir in Zukunft den Begriff *Relation* anstelle des Begriffs *Tabelle*. Zur weiteren Vertiefung der Begriffe aus Tab. 14 betrachten wir Abb. 16. Dort steht eine kleine Relation von Lieferanten im Mittelpunkt. Die einzelnen Zeilen heißen Tupel, die einzelnen Spalten Attribute. Insgesamt besitzt diese Relation 5 Zeilen und 4 Spalten, die Kardinalität dieser Relation ist daher 5, der Grad ist 4. Der Grad ist in einer Relation fest vorgegeben, während die Kardinalität durch Einfügen oder Löschen eines Tupels jederzeit geändert werden kann.

Der Primärschlüssel ist in dieser Relation das erste Attribut, die Lieferantennummer. Anders als in der Datenorganisation wird in relationalen Datenbanken die Eindeutigkeit des Primärschlüssels zwingend gefordert. Die Einträge zu jedem Attribut sind nicht willkürlich wählbar. Vielmehr werden sie einer Menge von gültigen Werten entnommen, der Definitionsmenge. Dies ist zum Beispiel die Menge aller gültigen Lieferantennummern, die Menge aller möglichen Namen, die Menge aller Städte im Umkreis usw. Es existiert demnach für jedes Attribut ein eigenes Definitionsgebiet, oder kurz Gebiet.

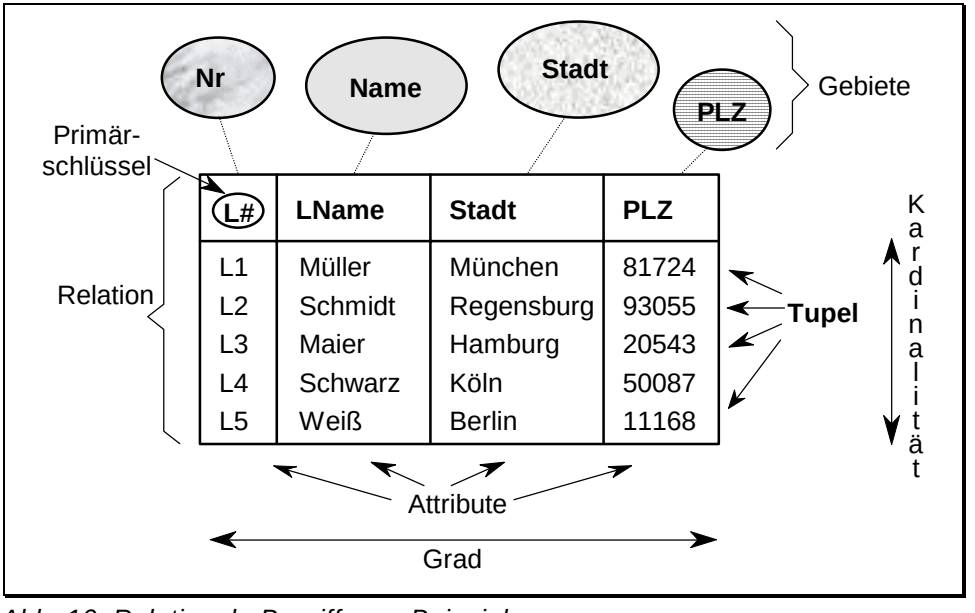


Abb. 16 Relationale Begriffe am Beispiel

Wie wir gesehen haben, wird eine Tabelle als Relation bezeichnet. Genaugenommen ist eine Relation R auf einer Ansammlung von Gebieten definiert und besteht aus zwei Teilen, einem Kopf und einem Rumpf.

Der Kopf wiederum besteht aus einer festen Anzahl von Attributen

$$\{ (A_1;D_1) , (A_2;D_2) , (A_3;D_3) , ... , (A_n;D_n) \} \quad ,$$

wobei jedes Attribut A_i genau mit einem Gebiet D_i korrespondiert ($1 \le i \le n$). Informell können wir sagen, dass der Kopf aus den Spaltenbezeichnungen aufgebaut ist.

Der Rumpf besteht aus einer variablen Anzahl von Tupeln. Jedes Tupel besitzt die Form:

$\{ (A_1;v_{i1}) , (A_2;v_{i2}) , (A_3;v_{i3}) , \ldots , (A_n;v_{in}) \}$

mit dem Tupelzähler $i = 1 \ldots m$, wobei $v_{ij} \in D_j$ ein Element des Gebiets D_j ist. In jedem Tupel ist also jedes Attribut einem festem Element des dazugehörigen Gebietes zugeordnet. Welches das dazugehörige Gebiet ist, wurde durch den Kopf festgelegt.

Wir bezeichnen die Zahl *m* als die Kardinalität und die Zahl *n* als den Grad der Relation *R*. Es sei nochmals angemerkt, dass der Grad bereits beim Erzeugen der Relation festgelegt wird. Das Hinzufügen einer Spalte (eines Attributs) ändert zwar den Grad, genaugenommen handelt es sich dabei aber um die Überführung einer Relation in eine andere. Die Kardinalität hingegen ist nach dem Erzeugen der Relation zunächst gleich Null und wird sich mit jedem Einfügen eines neuen Tupels in die Relation um eins erhöhen.

Definition (Relation)

☞Eine Relation ist eine Tabelle obiger Gestalt, bestehend aus einem Kopf und einem Rumpf, mit folgenden vier Eigenschaften:

① Es gibt keine doppelten Tupel.

② Tupel sind nicht geordnet (etwa von oben nach unten).

③ Attribute sind nicht geordnet (etwa von links nach rechts).

④ Alle Attribute sind atomar.

Eine solche Relation wird auch als normalisierte Relation bezeichnet.

Die ersten drei Punkte ergeben sich aus der streng mathematischen Betrachtungsweise der Relationen: Eine Relation ist eine auf Einzelmengen (Tupel) aufgebaute Menge, und bei Mengen existieren keine doppelten Einträge (entweder das Element ist in der Menge oder nicht; nach der Häufigkeit des Vorkommens wird nicht gefragt). Aber auch ohne Mathematik mögen wir uns fragen, welchen Gewinn an Information wir mit zwei oder sogar drei gleichen Tupeln erzielen können. Wir würden nur die Redundanz erhöhen, ohne dadurch auch nur den geringsten Vorteil als Gegenleistung zu erhalten.

Etwas anders verhält es sich mit der Anordnung der Tupel und Attribute. Zwar gibt es in Mengen keine Reihenfolge, für den Betrachter spielt es aber eine nicht unerhebliche Rolle, in welcher Reihenfolge die Information aufgezeichnet wird. Die Reihenfolge in Relationen dient jedoch nur der menschlichen Anschauung, bei der Abfrage werden wir sie nicht ausnutzen. Eine explizite Reihenfolge würde auch die Unabhängigkeit zwischen logischer und physischer Anordnung verletzen. Es ist aber gerade ein Ziel der relationalen Da-

tenbanken, die physische Speicherung der Daten dem Anwender zu verbergen. Wir werden demnach beim Zugriff auf Relationen nie die Anordnung der Tupel oder Attribute innerhalb der Relation ausnutzen. SQL besitzt deshalb auch gar keine entsprechende Abfragemöglichkeit!

Wir sollten nicht unterschätzen, welche Vorteile uns die Punkte 2 und 3 der Definition bringen. Wir brauchen uns beim Zugriff keine Gedanken über die Reihenfolge der Tupel machen, die Plazierung beim Hinzufügen eines neuen Tupels überlassen wir der Datenbank, ein komplexes Einfügen an einer bestimmten Stelle ist somit nicht möglich, und auch gar nicht nötig. Genauso verhält es sich beim Hinzufügen eines weiteren Attributs. Für uns heißt dies: Was nicht erlaubt ist, müssen wir auch nicht lernen. Dies ist ein Grund, warum die Zugriffssprachen für relationale Datenbanken so einfach sind.

Beim vierten Punkt müssen wir den Begriff atomar genauer festlegen. Atomar heißt, dass jedes Attribut v_{ij} genau aus einem Element des Gebietes D_j besteht. Ebenso bestehen die Gebiete nur aus Elementen, die im Sinne der Anwendung nicht noch weiter zergliedert werden können. Dies wollen wir an dem einfachen Beispiel des Gebiets *Stadt* veranschaulichen. Dieses Gebiet enthält ausschließlich Städtenamen, wobei in unserem Beispiel diese Menge etwa auf die Menge der deutschen Städte mit mehr als 10000 Einwohner eingeschränkt werden könnte. Alle Städtenamen, die diese Eigenschaft erfüllen, wären demnach Elemente dieses Gebiets. Die Attribute im Relationenrumpf enthalten je Tupel genau eines dieser Elemente. Die Städtenamen sind also im Sinne obiger Definition atomar, auch wenn sie vom inneren Aufbau her in Einzelbuchstaben zerlegt werden können.

Die Forderung der Atomarität ist nicht sofort einsichtig, die Handhabung von Relationen wird dadurch aber wesentlich vereinfacht. Dies wollen wir an unserem Beispiel der Relation *VerkaeuferProdukt* aus Tab. 13 aufzeigen. Wir hatten Redundanzen erhalten, da wir zu jedem verkauften Produkt den Namen und die Adresse des Verkäufers notierten. Diese Redundanzen würden wir sofort beseitigen, wenn wir im Attribut *Produktname* alle verkauften Produkte eines Verkäufers aufzählen würden. Die sich dabei ergebende Tabelle ist in Tab. 15 wiedergegeben. Wir erkennen, dass wir jetzt nur noch eine Zeile je Verkäufer benötigen.

Wenn wir uns nicht für die Einzelumsätze jedes Mitarbeiters interessieren, könnten wir geneigt sein, dieser Tabelle den Vorzug zu geben. Sie besitzt aber gravierende Nachteile. Der erste betrifft die physische Speicherung. In einer Relation sind (nach unserer Definition) alle Tupel gleich lang. Dies vereinfacht die physische Speicherung erheblich. Wir fügen immer gleich große Tupel hinzu, beziehungsweise löschen diese wieder. Dies ist erheblich einfacher

zu programmieren als in diesem Fall der nicht atomaren Relation. Verkauft nämlich Herr Schneider ein Radio für 500 DM, so ist das Attribut *Produktname* nur um ein Element zu erweitern. Hier wären dann womöglich verkettete Listen erforderlich. Unsere logische Tabellenstruktur würde sehr komplex auf die physische abgebildet. Vor allem dies wollte aber Codd mit der Vorstellung der relationalen Datenbanken vermeiden.

Tab. 15 Nicht atomare Relation VerkaeuferProdukt

Verk_Nr	Verk_Name	PLZ	Verk_Adresse	Produktname	Umsatz
V1	Meier	80075	München	Waschmaschine, Herd, Kühlschrank	17000
V2	Schneider	70038	Stuttgart	Herd, Kühlschrank	7000
V3	Müller	50083	Köln	Staubsauger	1000

Der zweite Nachteil betrifft direkt den Zugriff auf die Relation. Wir haben bereits gesehen, dass der Verkauf eines Radios durch Herrn Schneider nur zu einem Hinzufügen eines Attributs führte. Hätte hingegen Herr Schmidt, der noch nicht in der Tabelle aufgeführt ist, das Radio verkauft, so wäre ein Hinzufügen eines ganzen Tupels erforderlich gewesen. Der Verkauf eines Radios würde demnach zwei verschiedene Operationen erfordern, je nachdem ob der Verkäufer bereits in der Tabelle eingetragen ist oder nicht. Diese Unterschiede treten in normalisierten Tabellen nicht auf. In beiden Fällen erfolgt dort der Neueintrag eines Tupels.

Nachdem wir jetzt Relationen definiert haben, können wir auch relationale Datenbanken definieren:

Definition (relationale Datenbank)

☞Eine relationale Datenbank ist eine Datenbank, die der Benutzer als eine Ansammlung von zeitlich variierenden, normalisierten Relationen passender Grade erkennt.

Eine relationale Datenbank ist demnach eine Ansammlung von Relationen, die zueinander in einer gewissen Beziehung stehen, und die von einem Verwaltungssystem gemeinsam verwaltet werden. Dabei spielt es keine Rolle, ob die Datenbank auch physisch in Tabellenform aufgebaut ist. Wichtig ist allein der logische Aufbau, den der Benutzer sieht. Eine Datenbank, auf die ausschließlich mit SQL-Befehlen zugegriffen, und die ausschließlich mit SQL-Befehlen verwaltet wird, ist nach obiger Definition bereits eine relationale Datenbank. Dies liegt daran, dass SQL nur Relationen kennt. Es sei aber nicht

verschwiegen, dass es auch andere, einschränkendere Definitionen von relationalen Datenbanken gibt, die zusätzlich fordern, dass auch die physische Speicherung in Tabellenform geschehen muss.

In der Praxis müssen wir die Relationen noch untergliedern. Wir unterscheiden die „eigentlichen“ Relationen und die davon abgeleiteten Relationen. Im Einzelnen sind dies:

- **Basisrelationen**
  Dies sind alle real existierenden Relationen. Sie erschienen dem Datenbank-Designer als so wichtig, dass er ihnen einen Namen gab und sie als dauerhaften Bestandteil der Datenbank definierte (und damit abspeicherte).
- **Sichten (Views)**
  Diese Relationen heißen auch virtuelle Relationen. Sie existieren nicht real, erscheinen dem Benutzer aber wie „normale“ Relationen. Sichten werden letztlich aus Basisrelationen abgeleitet.
- **Abfrageergebnisse**
  Abfrageergebnisse sind ebenfalls Relationen. Diese existieren jedoch nur temporär im Arbeitsspeicher während der Ausgabe auf Bildschirm oder Drucker.
- **Temporäre Relationen**
  Diese Relationen existieren ähnlich den Abfrageergebnissen nur temporär. In der Regel werden sie aber nur bei bestimmten Ereignissen zerstört, etwa beim Beenden einer Transaktion.

Die für uns mit Abstand wichtigsten Relationen sind die Basisrelationen. Alle anderen Relationen werden aus diesen Basisrelationen erzeugt, sei es dauerhaft als Sichten, oder nur zur Ausgabe oder temporär. Sichten sind vor allem wichtig beim Zugriffsschutz. Mittels Sichten kann beispielsweise einem Benutzer der Einblick in Teile einer Basisrelation verwehrt werden, so dass ihm bestimmte, nicht für ihn vorgesehene Daten verborgen bleiben. Temporäre Relationen dienen vor allem zum Abspeichern von Zwischenergebnissen, wodurch beispielsweise ein komplexer Datenbankzugriff in mehrere einfachere zerlegt werden kann.

Basisrelationen und Sichten werden mittels DDL-Befehlen erzeugt, Abfrageergebnisse entstehen aus Zugriffs-, also DML-Befehlen. Temporäre Relationen werden in der Regel als DDL-Befehle (in Oracle und SQL-2) implemen-

tiert. In SQL lauten diese Befehle in der entsprechenden Reihenfolge wie folgt:

```
CREATE TABLE . . . ;                    (Erzeugen einer Basisrelation)
CREATE VIEW . . . ;                     (Erzeugen einer Sicht)
SELECT . . . ;                          (Abfrage)
CREATE TEMPORARY TABLE . . . ;          (Erzeugen einer temporären Relation)
```

Uns fehlt jetzt noch ein ganz wichtiger Begriff im Zusammenhang mit Relationen. Es handelt sich um den Primärschlüssel einer Relation. Eine exakte Definition folgt im nächsten Abschnitt. Hier sei nur erwähnt, dass der Primärschlüssel einer Relation grobgesprochen ein Attribut dieser Relation ist, durch das jedes Tupel der Relation eindeutig beschrieben wird. Der Primärschlüssel kann aus mehreren Einzelattributen zusammengesetzt sein.

Beispielsweise ist der Primärschlüssel in der Lieferantenrelation in Abb. 16 die Lieferantennummer. Bereits durch diese Nummer ist jedes Tupel in der Relation eindeutig festgelegt. Wird etwa der Lieferant mit der Lieferantennummer *L3* gesucht, so können wir sicher sein, höchstens einen Lieferanten zu finden.

In manchen Relationen besteht der Primärschlüssel aus mehr als einem Attribut. Betrachten wir nur die Relation *VerkaeuferProdukt* aus Tab. 13. Ein einzelnes Attribut legt dort noch nicht die Tupel eindeutig fest. Es sei dem Leser als Übung empfohlen, den Primärschlüssel dieser Relation zu bestimmen. Wir betrachten stattdessen ein weiteres Beispiel.

Tab. 16 Relation Lagerbestand mit zusammengesetztem Primärschlüssel

Produktname	Produkttyp	Bestand	Preis
Staubsauger	T06	25	498
Staubsauger	T17	17	219
...	...	...	...
Herd	T04	10	1598
Herd	T06	7	1998

← *Primärschlüssel* →

In der Relation *Lagerbestand* aus Tab. 16 sind die Einzelprodukte mit ihrem Bestand und dem Verkaufspreis aufgeführt. Jedes Produkt wird dabei durch seinen Namen und bei Namensgleichheit durch eine fortlaufende Nummer identifiziert. Damit besteht der Primärschlüssel aus den beiden Attributen *Produktname* und *Produkttyp*:

Primärschlüssel = (Produktname, Produkttyp)

Beachten Sie dabei, dass der Typ *T06* sowohl bei Staubsaugern als auch bei Herden vorkommt. Durch den Produkttyp allein sind hier also die Tupel noch nicht eindeutig identifiziert. In obiger Relation wäre es möglich, noch ein Attribut *Produktnummer* hinzuzufügen, in dem alle Produkte eindeutig durchnummeriert sind. Als Primärschlüssel könnte dann diese Produktnummer gewählt werden. Der Vorteil wäre ein nicht zusammengesetzter Primärschlüssel, der Nachteil eine erhöhte Redundanz. Letztlich ist es Geschmackssache, in diesem Fall bringt das zusätzliche Attribut aber kaum Vorteile.

Übrigens lässt sich leicht beweisen, dass jede Relation immer einen Primärschlüssel besitzt: In jeder Relation sind nämlich per definitionem die Tupel eindeutig. Wegen dieser Eindeutigkeit muss ein Primärschlüssel existieren. Dieser umfasst ein oder mehrere Attribute, im ungünstigsten Fall erstreckt er sich über alle Attribute.

Folglich existiert in jeder (korrekten) Relation ein Primärschlüssel. Ihn zu identifizieren, ist allerdings nicht immer ganz einfach.

3.3 Relationale Integritätsregeln

Bereits aus der Definition von Relationen ersehen wir, dass wir bestimmte Bedingungen beim Erzeugen von Relationen einhalten müssen. Um aber optimal mit Datenbanken arbeiten zu können, müssen wir weitere Einschränkungen akzeptieren. In diesem Abschnitt werden wir uns mit den zwei Regeln zur Integrität befassen. Wie bereits in Kapitel 2 erwähnt wurde, beschäftigt sich die Integrität mit der korrekten Speicherung von Daten. Wir unterscheiden im wesentlichen vier Arten der Integrität:

- **Physische Integrität**
  Diese beinhaltet die Vollständigkeit der Zugriffspfade und der physischen Speicherstrukturen. Die physische Integrität kann der Datenbankprogrammierer nicht beeinflussen. Für diese Integrität muss die Datenbank und das darunterliegende Betriebssystem sorgen.
- **Ablaufintegrität**
  Hier geht es um die Korrektheit der ablaufenden Programme, dass etwa keine Endlosschleifen vorkommen dürfen, oder die Datenkonsistenz im Mehrbenutzerbetrieb garantiert wird. Für die Ablaufintegrität ist der

Anwendungsprogrammierer, insbesondere aber auch der Datenbankdesigner verantwortlich.

- **Zugriffsberechtigung**
  Hier kommt es auf die korrekte Vergabe der Zugriffsrechte an, was im wesentlichen die Aufgabe des Datenbank-Administrators ist.
- **Semantische Integrität**
  Hier handelt es sich um die Übereinstimmung der Daten aus der nachzubildenden realen Welt mit den abgespeicherten Informationen. Semantische Integrität ist zugegebenermaßen ein Problem, da der Rechner von sich aus nur schwer entscheiden kann, ob eine Eingabe den Daten der realen Welt entspricht.

Im letzten Punkt sind der Anwendungsprogrammierer, der Datenbank-Administrator und der Datenbankhersteller gleichermaßen gefordert. Der Datenbankhersteller wird vom Sprachumfang her die Eingabeüberprüfung unterstützen, sei es über Bildschirmmasken oder Trigger (Prozeduren, die bei bestimmten auftretenden Ereignissen ablaufen). Der Datenbank-Administrator wird die zugelassenen Gebiete so weit wie möglich einschränken, und der Anwendungsprogrammierer wird die angebotenen Werkzeuge intensiv nutzen. Als Beispiel betrachten wir ein Lager, in dem 12 Kühlschränke aufbewahrt werden. Bei der Eingabe in den Rechner wird versehentlich 112 eingetippt. Der Fehler wird so schnell nicht entdeckt werden, die Integrität der Datenbank ist verletzt. Sollte es aber aus Lagerkapazitätsgründen gar nicht möglich sein, mehr als 50 Kühlschränke im Lager zu stapeln, so muss das Eingabeprogramm alle Eingaben größer als 50 von vornherein abweisen.

Die Überprüfung der Eingabe auf Korrektheit bedeutet also:

- Die Gebiete D_i sind so klein wie möglich zu wählen.
- Es gilt für alle Attributseinträge v_{ij}: $v_{ij} \in D_j$ für alle gültigen i und j.
- Soweit möglich sind die Einträge v_{ij} vom Programm automatisch auszuwählen, etwa bei der Vergabe der nächsten laufenden Nummer.

Die im Folgenden vorgestellten zwei Integritätsregeln tragen wesentlich zur Ablaufintegrität bei. Eine Verletzung auch nur einer dieser Regeln würde zu einer nur noch schwer handhabbaren Datenbank führen und ist daher grundsätzlich verboten. Beginnen wir mit der ersten Integritätsregel.

3.3.1 Entitäts-Integritätsregel

Die erste Integritätsregel oder Entitäts-Integritätsregel befasst sich mit dem Aufbau des Primärschlüssels. Wie wir bereits wissen, identifiziert der Primärschlüssel jedes Tupel eindeutig. Dies heißt umgekehrt, dass jedes Tupel einen Primärschlüssel besitzen muss; oder genauer, dass in den Primärschlüsselattributen vernünftige Werte enthalten sein müssen. Über den Begriff *vernünftig* lässt sich streiten, aber den Platz einfach leer zu lassen, ist sicherlich nicht vernünftig:

1. Integritätsregel (Entitäts-Integritätsregel)

☞Keine Komponente des Primärschlüssels einer Basisrelation darf nichts enthalten.

Die erste Integritätsregel bezieht sich nur auf Basisrelationen. Dies sind auch die einzigen real existierenden Teile einer Datenbank. Dass diese Regel etwa bei Abfrageergebnissen verletzt sein kann, erweitert nur die Abfragemöglichkeiten, ohne dadurch die Datenbanken an sich negativ zu beeinflussen. Wem es Spaß macht, darf also in der *VerkaeuferProdukt*-Relation den Befehl

```
SELECT Produktname
 FROM   VerkaeuferProdukt ;
```

eingeben. Er wird eine nicht-eindeutige Liste aller in der Relation aufgeführten Produkte erhalten.

Der Name Entitäts-Integritätsregel kommt daher, dass ein Tupel (eine Entität, ein Ding) entweder existiert oder nicht. Und diese Existenz ist mit der Existenz des Primärschlüssels verknüpft. Es hat keinen Sinn, ein Tupel einzutragen, und den Primärschlüssel erst später nachzutragen. Primärschlüssel und gesamtes Tupel sind nach dieser Regel eine untrennbare Einheit. Es existiert entweder ein Ding (eine Entität), dann ist es identifizierbar (durch den Primärschlüssel), oder es existiert nicht!

Da der Begriff *Primärschlüssel* in Datenbanken sehr wichtig ist, wollen wir ihn nun endlich exakt definieren. Wie wir bereits wissen, besitzt jede Relation einen eindeutigen Schlüssel, genauer: mindestens einen! Wenn wir nämlich als Beispiel die Tabelle der chemischen Elemente (Tab. 17) betrachten, so werden wir gleich drei mögliche Primärschlüssel entdecken. Wir können hier die Protonenanzahl, den Namen und das Symbol des chemischen Elements als

Primärschlüssel wählen. Alle drei sind mögliche Primärschlüssel, sogenannte Schlüsselkandidaten.

Tab. 17 Tabelle der chemischen Elemente

Protonen	Atomgewicht	Name	Symbol
1	1,0079	Wasserstoff	H
2	4,0026	Helium	He
3	6,941	Lithium	Li
...	...	...	...

Definition (Schlüsselkandidat)

☞Ein eventuell aus mehreren einzelnen Attributen zusammengesetztes Attribut heißt Schlüsselkandidat, falls es

- eindeutig jedes Tupel identifiziert und
- minimal ist, d.h. beim Weglassen eines einzelnen Attributs die Eindeutigkeit verloren geht.

Definition (Primärschlüssel, alternativer Schlüssel)

☞Besitzt eine Relation mehrere Schlüsselkandidaten, so wird davon einer als Primärschlüssel ausgewählt; alle anderen heißen alternative Schlüssel.

Die Minimalität bei Schlüsselkandidaten und damit auch bei Primärschlüsseln wird gefordert, um Schlüssel zu vermeiden, die aus zu vielen Attributen zusammengesetzt sind. Dies würde Abfragen komplizieren, ohne dass dadurch an anderer Stelle ein Vorteil erkennbar wäre.

In der Tabelle der chemischen Elemente existieren drei Schlüsselkandidaten. Unser Vorschlag ist, die Anzahl der Protonen als Primärschlüsel zu wählen, Name und Symbol sind dann alternative Schlüssel. Die Bedeutung der Schlüsselkandidaten liegt darin, dass Abfragen mit diesen Schlüsseln automatisch eindeutige Ergebnisse liefern. Alle anderen Abfragen bedingen eine ungewisse Anzahl von Tupeln als Antwort. Betrachten wir als Beispiel folgende Abfragen zu Tab. 16:

```
SELECT  Produktname, Preis
 FROM   Lagerbestand
 WHERE  Produktname = 'Staubsauger'
   AND  Produkttyp = 'T06' ;
```

```
SELECT  Produktname, Preis
 FROM   Lagerbestand
 WHERE  Produktname = 'Herd' ;
```

Im linken Beispiel wurde nach einem Schlüsselkandidaten ausgewählt. Als Ergebnis erhalten wir in einer Zeile das Produkt zusammen mit dessen Preis. Im zweiten Fall wird möglicherweise eine größere Anzahl von Herdgeräten zusammen mit dessen Preisen aufgelistet. Um hier die mögliche Flut von erhaltenen Daten weiterverarbeiten zu können, müssen wir diese entweder in einem entsprechend großen Feld zwischenspeichern, oder wir müssen die Daten in einer Schleife zeilenweise einlesen und bearbeiten. Wir verweisen dazu auf das Arbeiten mit Cursorn in Kapitel 9.

Am Beispiel der Relation *Lagerbestand* in Tab. 16 wollen wir die Ermittlung des Schlüsselkandidaten nachvollziehen. Wir haben bereits gesehen, dass jedes Attribut für sich das Tupel nicht eindeutig identifiziert. Erst die beiden Attribute *Produktname* und *Produkttyp* zusammen gewährleisten diese Eindeutigkeit. Bestands- oder Preisangaben führen in der Regel zu keiner eindeutigen Identifikation, da deren Inhalte zeitlichen Änderungen unterliegen. Insbesondere sind hier gleiche Werte in mehreren Tupeln jederzeit zulässig. *Produktname* und *Produkttyp* bilden zusammen den einzigen Schlüsselkandidaten. Denn das Entfernen eines der beiden Attribute verletzt die eindeutige Identifikation, ein Hinzufügen eines weiteren Attributs verletzt die Minimalität. Somit ist auch der Primärschlüssel schon festgelegt, alternative Schlüssel existieren nicht.

Es sei hier nochmals erwähnt, dass es immer empfehlenswert ist, dass Schlüsselkandidaten und insbesondere Primärschlüssel nur aus einem Attribut bestehen. Dies ist nicht immer zu erreichen, doch je kleiner die Anzahl der Schlüsselattribute ist, um so einfacher ist letztlich die Programmierung des Zugriffs auf diese Relationen.

Weiter sollte nicht übersehen werden, dass die Entitäts-Integritätsregel nur für Primärschlüssel, nicht für die Alternativschlüssel gilt. Einträge in Alternativschlüsseln dürfen demnach leer bleiben. Der Anwender hat sich ja für einen anderen Schlüsselkandidaten als eindeutigen Identifikator entschieden.

3.3.2 Referenz-Integritätsregel

Die erste Integritätsregel befasst sich mit einzelnen Relationen, die zweite Regel beeinflusst das Zusammenwirken der Relationen untereinander. Wir haben aus der Definition des Begriffs *Datenbank* bereits gelernt, dass Datenbanken zusammengehörige Dinge verwalten. Relationen an sich sind zunächst einzelne Objekte ohne Bezug zueinander. Sie haben in einer Datenbank aber nur dann ihre Daseinsberechtigung, wenn es Beziehungen zwischen ihnen

gibt. In relationalen Datenbanken spielt hier ein weiterer Schlüssel eine große Rolle, der sogenannte Fremdschlüssel. Fremdschlüssel sind der Kit, der die einzelnen Relationen zu einem gemeinsamen Datenbestand zusammenfügt.

Definition (Fremdschlüssel)

☞Ein (möglicherweise zusammengesetztes) Attribut einer Basisrelation heißt Fremdschlüssel, falls

- das ganze Attribut entweder nichts oder einen definierten Inhalt enthält,
- eine Basisrelation mit einem Primärschlüssel existiert, so dass jeder definierte Wert des Fremdschlüssels einem Wert jenes Primärschlüssels entspricht.

Der erste Punkt der Definition bezieht sich nur auf zusammengesetzte Attribute. Ein zusammengesetztes Attribut ist ein Attribut, das aus mehr als einem einzelnen Attribut aufgebaut ist. Etwa ist *Produktname+Produkttyp* ein zusammengesetztes Attribut aus Tab. 16. Dieser erste Punkt besagt, dass entweder alle Einzelattribute einen sinnvollen Inhalt besitzen, oder alle Einzelattribute gemeinsam leer sind. Eine Kombination von beiden ist für Fremdschlüssel demnach unzulässig.

Der zweite Punkt ist noch wichtiger. Er besagt mit anderen Worten, dass in einem nichtleeren Fremdschlüsselwert nur ganz bestimmte Inhalte erlaubt sind; nämlich Inhalte, die bereits im Primärschlüssel einer Relation der Datenbank abgespeichert sind, auf den sich dieser Fremdschlüssel bezieht. Dies impliziert, dass der Fremdschlüssel aus genauso vielen Attributen mit jeweils den gleichen Gebietsdefinitionen wie dieser Primärschlüssel besteht.

Wir wollen diesen Sachverhalt an einem Beispiel aufzeigen. Wir betrachten hierzu die drei Relationen *Personal*, *Kunde* und *Auftrag* aus Tab. 18 bis Tab. 20, die von der Beispieldatenbank *Radl* abgeleitet sind. Diese Datenbank *Radl* wird in Anhang A ausführlich vorgestellt. Wir werden im weiteren immer wieder auf diese Datenbank verweisen.

In den Relationen *Personal*, *Kunde* und *Auftrag* ist jeweils das erste Attribut der Primärschlüssel. Dies sind die Attribute *Persnr*, *Nr* und *AuftrNr*. Diese drei Relationen sind über Fremdschlüssel in der Relation *Auftrag* miteinander verknüpft. In dieser Relation *Auftrag* werden die eingegangenen Aufträge eingetragen, und zusammen mit dem Auftragsdatum sind dort die Personalnummer des Verkäufers und die Kundennummer des Kunden vermerkt. Möchten wir etwa zum Auftrag mit der Nummer 2 den Namen des Verkäufers wissen,

so finden wir in dieser Relation unter *Persnr* die Nummer 5 und in der Relation *Personal* unter der Personalnummer 5 den Namen *Johanna Köster*. Analoges gilt für den Namen des Kunden, in unserem Falle ist dies der Kunde *Maier Ingrid*.

Tab. 18 Relation Personal (Auszug aus Tab. 60)

Persnr	Name	Ort	Vorgesetzt	Gehalt
1	Maria Forster	Regensburg	NULL	6800.00
2	Anna Kraus	Regensburg	1	3400.00
3	Ursula Rank	Straubing	6	4200.00
4	Heinz Rolle	Nürnberg	1	4600.00
5	Johanna Köster	Nürnberg	1	3200.00
6	Marianne Lambert	Landshut	NULL	5500.00
7	Thomas Noster	Regensburg	6	3800.00
8	Renate Wolters	Kelheim	1	4700.00
9	Ernst Pach	Passau	6	800.00

Tab. 19 Relation Kunde (Auszug aus Tab. 59)

Nr	Name	Strasse	PLZ	Ort
1	Fahrrad Shop	Obere Regenstr. 4	93059	Regensburg
2	Zweirad-Center Staller	Kirschenstr. 20	01169	Dresden
3	Maier Ingrid	Universitaetsstr. 33	93055	Regensburg
4	Rafa - Seger KG	Liebigstr. 10	10247	Berlin
5	Biker Ecke	Lessingstr. 37	22087	Hamburg
6	Fahrräder Hammerl	Schindlerplatz 7	81739	München

Tab. 20 Relation Auftrag (entspricht Tab. 62)

AuftrNr	Datum	Kundnr	Persnr
1	04.08.1998	1	2
2	06.09.1998	3	5
3	07.10.1998	4	2
4	18.10.1998	6	5
5	06.11.1998	1	2

Das Attribut *Persnr* in der Relation *Auftrag* ist also ein Fremdschlüssel und bezieht sich auf den Primärschlüssel in der Relation *Personal*. Es ist jetzt nach der Definition von Fremdschlüsseln nicht erlaubt, im Attribut *Persnr* der Relation *Auftrag* einen Wert einzutragen, der im dazugehörigen Primärschlüssel der Relation *Personal* nicht existiert. Völlig analog gilt die gleiche Aussage

für den Fremdschlüssel *Kundnr* in der Relation *Auftrag*, der sich auf den Primärschlüssel *Nr* in der Relation *Kunde* bezieht.

Ein besonders interessantes Beispiel für Fremdschlüssel finden wir noch in der Relation *Personal*. Hier ist das Attribut *Vorgesetzt* ein Fremdschlüssel, der sich auf den Primärschlüssel der eigenen Relation bezieht. Dies ist sofort einsichtig, da alle Mitarbeiter, auch Vorgesetzte, in der Relation aufgeführt sind. Der Eintrag *Null* bei Frau Koster und Frau Lambert bedeutet, dass hier definitiv nichts eingetragen ist, weil beide keinem Vorgesetzten zugeordnet sind.

Da es für die Konsistenz einer Datenbank sehr wichtig ist, dass alle Fremdschlüsselinhalte korrekt sind, stellen wir dafür eine zweite Integritätsregel auf:

2. Integritätsregel (Referenz-Integritätsregel)

☞Eine relationale Datenbank enthält keinen Fremdschlüsselwert (ungleich *Null*), der im dazugehörigen Primärschlüssel nicht existiert.

Diese Regel ist nicht neu, sie ist tatsächlich schon in der Definition der Fremdschlüssel enthalten. Oder anders gesagt, diese Regel ist so wichtig, dass sie gleich in die Definition der Fremdschlüssel mit aufgenommen wurde. Die Integritätsregel ist sehr streng auszulegen, sie darf auch nicht kurzzeitig verletzt werden. Dass dies gar nicht so einfach zu erfüllen ist, wollen wir an unserer obigen Beispiel-Datenbank aufzeigen.

Nehmen wir an, Frau Köster verlässt die Firma, und aus datenschutzrechtlichen Gründen soll der Eintrag in der Relation *Personal* gelöscht werden. Ein Löschen des Tupels würde jedoch die Referenz-Integritätsregel verletzen; denn in der Relation *Auftrag* wird auf die Personalnummer 5 verwiesen, die dann gar nicht mehr existieren würde. Ein Löschen des Tupels in der Tabelle *Personal* erfordert demnach zunächst zwingende Änderungen in der Relation *Auftrag*. Bei genauer Betrachtung gibt es nur zwei Möglichkeiten: Entweder wir müssen in der Relation *Auftrag* den entsprechenden Verweis durch *Null* ersetzen oder das ganze Tupel entfernen.

Wir müssen demnach sowohl beim Löschen eines Tupels als auch beim Ändern des Primärschlüssels eines Tupels immer beachten, ob Fremdschlüssel auf dieses Tupel verweisen. Wenn ja, dann existieren genau drei Reaktionsmöglichkeiten:

- Zurückweisen des Löschens bzw. Änderns
- Löschen bzw. Ändern aller darauf verweisenden Tupel
- Nullsetzen aller darauf verweisenden Fremdschlüssel

Seit 1992 gibt es mit SQL-2 genormte Sprachkonstrukte, die diese Eigenschaften beschreiben. Entsprechend der Reihenfolge der obigen drei Punkte besitzt jeder Fremdschlüssel eine der drei Eigenschaften, getrennt nach Löschen und Ändern:

ON DELETE NO ACTION	ON UPDATE NO ACTION
ON DELETE CASCADE	ON UPDATE CASCADE
ON DELETE SET NULL	ON UPDATE SET NULL

Statt *On Delete No Action* findet man in der Literatur häufig den aussagekräftigeren, aber nicht genormten Begriff *On Delete Restrict*. Das Löschen wird hier zurückgewiesen, eine Löschaktion findet nicht statt. Analoges gilt für den Begriff *On Update No Action*.

Sollten mehrere Fremdschlüssel auf ein zu änderndes oder löschendes Tupel verweisen, so genügt es schon, wenn nur einer dieser Fremdschlüssel die Manipulation verbietet, um das Ändern oder Löschen zurückzuweisen. Ist dies nicht der Fall, so werden die Tupel all dieser Fremdschlüssel je nach Vorgabe ebenfalls geändert oder gelöscht, oder die Fremdschlüssel werden auf *Null* gesetzt.

Ein Spezialfall ist das kaskadierende Löschen (mehrfaches *On Delete Cascade*). Dieser Fall tritt ein, wenn auf die Relation, die unseren Fremdschlüssel enthält, weitere Fremdschlüssel verweisen. Auch auf diese weiteren Fremdschlüssel können wieder andere Fremdschlüssel zeigen und so fort. In diesem Fall muss überprüft werden, ob das Löschen von Tupeln das Löschen immer weiterer Tupel nach sich zieht. Ein solches fortgesetztes Löschen ist wegen der Referenz-Integritätsregel grundsätzlich nur dann erlaubt, wenn keiner der geschachtelten Fremdschlüssel das Löschen verbietet. Ansonsten darf kein einziges Tupel gelöscht werden.

Als Beispiel betrachten wir nochmals Tab. 18 und Tab. 20. Nehmen wir hier an, dass für alle Fremdschlüssel die Eigenschaft *On Delete Cascade* gelte, so würde das Entfernen von Frau Forster auch alle Mitarbeiter löschen, die Frau Forster als Vorgesetzte eingetragen haben, unter anderem Frau Köster und Frau Kraus. Dies wiederum würde kaskadierend das Löschen aller Tupel der Relation *Auftrag* bewirken. Wäre nun das Entfernen von Aufträgen untersagt (*On Delete No Action*), so würde das Löschen von Frau Köster und Frau Kraus und damit auch das Löschen von Frau Forster abgewiesen werden.

Es sei noch darauf hingewiesen, dass das Nullsetzen von Fremdschlüsseln im Falle des Löschens oder Änderns des darauf verweisenden Tupels nicht

immer erwünscht ist. Generell verhindern wir in SQL das Nullsetzen von Attributen durch die zusätzliche Angabe

```
NOT NULL
```

Diese Einschränkung ist nicht auf Fremdschlüssel beschränkt, sondern kann für jedes beliebige Attribut gesetzt werden. Gemäß Definition gilt diese Einschränkung für Primärschlüssel implizit. Zur Vertiefung der beiden Integritätsregeln sei als Übung auf die Beispieldatenbank *Radl* im Anhang verwiesen. Alle dortigen Relationen sind über Fremdschlüssel miteinander verknüpft. Es wird empfohlen, alle Fremdschlüssel und Primärschlüssel der Datenbank *Radl* selbst zu bestimmen. Zur Kontrolle kann im Anhang die Beschreibung der meisten Relationen einschließlich der Angabe der Schlüssel nachgelesen werden.

Die letzten beiden Abschnitte haben uns die Wichtigkeit der Primär- und Fremdschlüssel in einer relationalen Datenbank vor Augen geführt. Jeder Datenbankdesigner muss alle Primär- und Fremdschlüssel in allen Basisrelationen zusammen mit ihren Eigenschaften genau kennen! Wir werden uns daher beim Datenbankentwurf in Kapitel 5 noch weiter mit Fremdschlüsseln beschäftigen.

3.4 Relationale Algebra

In den bisherigen Abschnitten dieses Kapitels haben wir uns mit dem statischen Aufbau einer relationalen Datenbank beschäftigt. Doch früher oder später werden die Relationen mit Daten gefüllt, und diese Daten unterliegen dann ständigen Manipulationen.

Für alle Arten von Manipulationen, für das Anlegen von Sichten, für das Festlegen von Zugriffsrechten oder Integritätsbedingungen oder einfach für das Erzeugen von Ausgaben müssen Daten eingegrenzt, dann gelesen und schließlich logisch zusammengefasst werden. Dieses Lesen sollte mit möglichst geringem sprachlichen Aufwand möglich sein. Zum Beschreiben solcher Zugriffe benötigen wir also eine Zugriffssprache, die aus mehr oder weniger komplexen Ausdrücken bestehen kann und syntaktisch festgelegt ist. Eine solche Sprache ist zum Beispiel die Sprache SQL. Codd selbst baute seine ursprüngliche Sprachdefinition aus acht Operatoren auf und erstellte auf diesen Operatoren eine vollständige Syntax, die als relationale Algebra bezeichnet

wird. Er zeigte weiter, dass damit alle nur denkbaren Zugriffe auf beliebige Relationen der Datenbank möglich sind.

Wir werden die relationale Algebra im weiteren nicht verwenden. Zum allgemeinen Verständnis von Datenbankzugriffen ist es jedoch empfehlenswert, sich mit dieser Algebra zumindest im Überblick auseinanderzusetzen. Die folgenden Ausführungen sollen einen solchen Überblick vermitteln. Für detailliertere Informationen sei insbesondere auf [Date95] verwiesen.

3.4.1 Relationale Operatoren

Wie wir bereits erwähnt haben, ist eine Relation *R* mathematisch gesehen nichts anderes als eine Menge. E. F. Codd definierte acht Operatoren auf diesen Mengen *R*. Mathematisch gesehen sind diese Mengenoperatoren entweder auf dem Kreuzprodukt $\Re \times \Re$ oder direkt auf $\Re$ mit $\Re$ als der Menge aller Relationen definiert. Im ersten Fall liegt ein binärer, im zweiten Fall ein unärer Operator vor. Das Bildgebiet ist in beiden Fällen wieder eine Menge (Relation).

Bekannte binäre arithmetische Operatoren sind etwa die Addition oder Multiplikation, ein bekannter unärer arithmetischer Operator ist die Negation. Analog zu diesen arithmetischen besitzen unsere relationalen Operatoren *op* daher eine der beiden folgenden Formen:

binäre Operatoren: *R1* op *R2*
unäre Operatoren: op *R*

mit *R1*, *R2* und *R* als beliebigen Relationen. Diese acht Operatoren können wir in vier Mengen- und vier relationenspezifische Operatoren einteilen. Wir wollen zunächst die vier Mengenoperatoren vorstellen, bei denen es sich ausschließlich um binäre Operatoren handelt:

- **Vereinigung**: R1 UNION R2
  Aus den beiden Relationen *R1* und *R2* wird eine neue Relation erzeugt, die alle Tupel enthält, die in wenigstens einer der beiden Relationen vorkommen.
- **Schnitt**: R1 INTERSECT R2
  Aus den beiden Relationen *R1* und *R2* wird eine neue Relation erzeugt, die nur die Tupel enthält, die in beiden Relationen vorkommen.

- **Differenz**: R1 MINUS R2
  Aus den beiden Relationen *R1* und *R2* wird eine neue Relation erzeugt, die all die Tupel enthält, die in *R1*, nicht aber in *R2* vorkommen.
- **Kartesisches Produkt**: R1 TIMES R2
  Aus den beiden Relationen *R1* und *R2* wird eine neue Relation erzeugt, die aus allen möglichen Kombinationen der Tupel der beiden Relationen bestehen.

Alle diese vier Operatoren sind aus der Mathematik (Mengenlehre) wohlbekannt und dürften bis auf das Kartesische Produkt keinerlei Schwierigkeiten bereiten. Das Kartesische Produkt verknüpft einfach das erste Tupel in *R1* mit dem ersten Tupel in *R2*, dann mit dem zweiten bis hin zum letzten Tupel in *R2*. Dieses Verfahren wiederholt sich für jedes weitere Tupel in *R1*. Ist etwa die Kardinalität von *R1* gleich *n1* und die von *R2* gleich *n2*, so ist die Kardinalität von *R1 TIMES R2* gleich $n1 \cdot n2$. Bevor wir diese Operatoren nochmals in einem Überblick betrachten, wollen wir zunächst die restlichen vier Operatoren vorstellen. Hier handelt es sich um Operatoren, die speziell auf Relationen (Tabellen) zugeschnitten sind:

- **Restriktion**: R WHERE Bedingung
  Aus der Relation *R* wird durch diesen unären Operator eine neue Relation erzeugt, die alle Tupel aus R enthält, die der angegebenen Bedingung genügen.
- **Projektion**: R **[** Attributsauswahl **]**
  Aus der Relation *R* wird durch diesen unären Operator eine neue Relation erzeugt, die aus allen Tupeln von R besteht, eingeschränkt auf die angegebene Auswahl von Attributen.
- **Verbindung**: R1 JOIN R2
  Aus den beiden Relationen *R1* und *R2* wird eine neue Relation erzeugt, die aus allen möglichen Kombinationen zwischen den Tupeln der beiden Relationen besteht, wobei gemeinsame Attribute beider Relationen als Verknüpfung dienen.
- **Division**: R1 DIVIDEBY R2
  Vorausgesetzt wird hier, dass die Relation *R1* mindestens alle Attribute *Att* von *R2* enthält. Aus den beiden Relationen *R1* und *R2* wird dann eine neue Relation erzeugt, die alle Attribute von *R1* außer den Attributen *Att* enthält, und die aus allen Tupeln aus *R1* besteht, deren Werte in den Attributen *Att* mit den Werten aus *R2* übereinstimmen.

Sicherlich gibt es Probleme, insbesondere die letzten beiden Operatoren zu verstehen. Dies liegt vielleicht auch daran, dass wir nicht jeden Tag mit Operatoren auf Relationen arbeiten, sondern mehr das Arbeiten mit Zahlen oder Mengen gewöhnt sind. Die beiden unären Operatoren sind noch leicht nachvollziehbar. Im ersten Fall der Restriktion entfernen wir aus einer Relation *R* einige Zeilen (Tupel), im zweiten Fall der Projektion entfernen wir aus einer Relation *R* einige Spalten (Attribute). Um welche Zeilen oder welche Spalten es sich handelt, wird unter *Bedingung* bzw. *Attributsauswahl* angegeben.

Schwieriger wird es bei der Verbindung und der Division von Relationen. Wir wollen daher je ein Beispiel angeben. Gehen wir hierzu von den Relationen *Personal* und *Auftrag* aus Tab. 18 und Tab. 20 aus. Die Verbindung für den Join ist in diesen beiden Relationen das Attribut *Persnr*. In Tab. 21 ist der Join dieser beiden Relationen angegeben.

Tab. 21 Natürliche Verbindung der Relationen Personal und Auftrag

AuftrNr	Datum	Kundnr	Persnr	Name	Vorgesetzt	Gehalt	Ort
1	04.08.98	1	2	Anna Kraus	1	3400.00	Regensburg
2	06.09.98	3	5	Joh. Köster	1	3200.00	Nürnberg
3	07.10.98	4	2	Anna Kraus	1	3400.00	Regensburg
4	18.10.98	6	5	Joh. Köster	1	3200.00	Nürnberg
5	06.11.98	1	2	Anna Kraus	1	3400.00	Regensburg

Das verbindende Attribut *Persnr* ist hervorgehoben. Es erscheint in der Ergebnistabelle nur einmal. Da zu jedem Fremdschlüsselwert ein Primärschlüsselwert der dazugehörigen Relation existiert, besitzt das Ergebnis dieser Verbindung die gleiche Kardinalität wie die Relation *Auftrag* selbst. Zwecks Veranschaulichung der Division führen wir die folgenden drei Relationen *A*, *B* und *C* ein:

A:

L#	P#
L1	P1
L1	P2
L1	P3
L2	P1
L2	P2
L3	P2
L3	P3

B:

P#
P1

C:

P#
P2
P3

Die Operationen *A DIVIDEBY B* und *A DIVIDEBY C* liefern folgende Ergebnisrelationen:

A DIVIDEBY B:

L#
L1
L2

A DIVIDEBY C:

L#
L1
L3

Im ersten Falle besteht die Relation *B* nur aus dem Tupel *P1*. In der Relation *A* stehen genau die Tupel *L1* und *L2* mit diesem Tupel *P1* in Verbindung. Im zweiten Fall existieren Verknüpfungen zwischen *L1* und allen Tupeln von *C*, das gleiche gilt für *L3*.

Im Falle des *Join*-Beispiels gehen wir immer davon aus, dass Attribute zweier Relationen miteinander in Verbindung stehen. Dies ist in relationalen Datenbanken immer dann gegeben, wenn ein Fremdschlüssel auf den Primärschlüssel einer Relation verweist. Der Join verbindet dann alle Tupel der einen Relation mit allen Tupeln der anderen Relation, falls sie im verbindenden Attribut gleiche Werte besitzen. Natürlich darf ganz allgemein dieses Attribut aus mehreren Einzelattributen zusammengesetzt sein. Auch Verbindungen auf die gleiche Tabelle sind möglich. Es sei zur Übung empfohlen, aus der Relation *Personal* in Tab. 18 die Verbindung *Personal JOIN Personal* zu erzeugen, wobei das Fremdschlüsselattribut *Vorgesetzt* mit dem Primärschlüssel *Persnr* derselben Relation verknüpft wird.

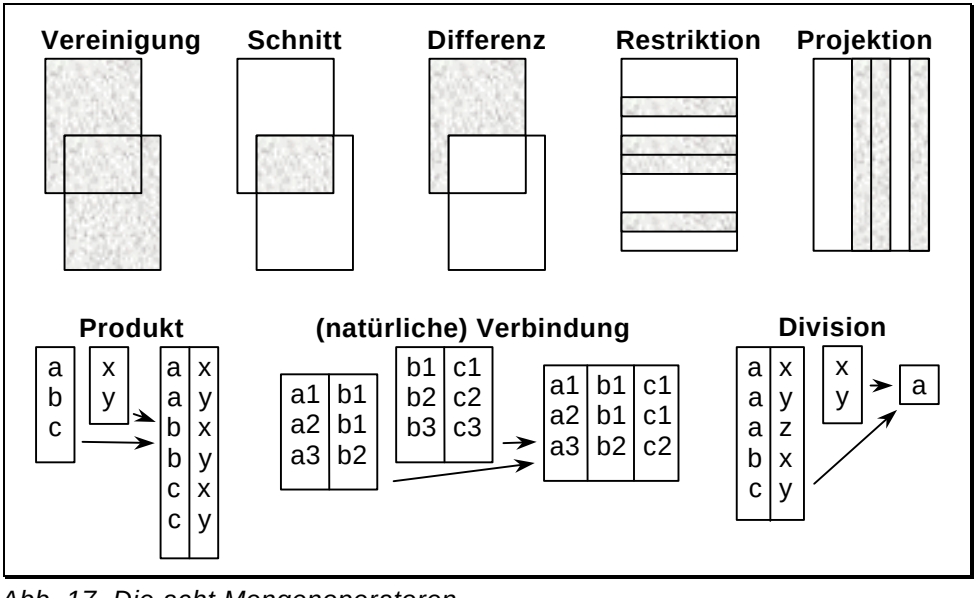


Abb. 17 Die acht Mengenoperatoren

Verbindungen hängen eng mit Fremdschlüsseln zusammen und tauchen in der Praxis auch sehr häufig auf. Wir werden im Zusammenhang mit SQL noch

ausführlich darauf eingehen. Hier sei nur erwähnt, dass diese Art der Verbindung als natürliche Verbindung oder natürlicher Join bezeichnet wird. Wir werden im nächsten Kapitel noch weitere Joins kennenlernen.

Das Beispiel der Division zeigt, dass das Attribut *P#* des Divisors auch im Dividenden *A* vorkommt. Im Ergebnis erscheinen alle Attribute des Dividenden mit Ausnahme der Attribute des Divisors. In unserem Beispiel blieb daher nur das Attribut *L#* übrig.

Zum Schluss wollen wir noch auf Abb. 17 verweisen, wo die einzelnen Operatoren und ihre Wirkungsweise grafisch dargestellt sind. Beachten Sie in dieser Abbildung den deutlich erkennbaren Unterschied zwischen kartesischem Produkt und der natürlichen Verbindung!

3.4.2 Eigenschaften der relationalen Operatoren

Wir haben im letzten Abschnitt die acht Grundoperatoren kennengelernt, die auf Relationen angewendet werden. Wir wollen hier einige Eigenschaften dieser Operatoren aufzählen. Zunächst ist wichtig, dass das Ergebnis bei allen acht Operatoren wieder Relationen sind. Wir können demnach auf diese Ergebnisse ein weiteres Mal unsere Operatoren anwenden. Durch mehrmaliges Wiederholen führt dies schließlich zu komplexen Ausdrücken, aufgebaut aus unseren acht Operatoren. Wie wir arithmetische Ausdrücke aus Termen und diese wieder aus Faktoren aufbauen, so können wir auch hier eine Syntax definieren. Diese Syntax beinhaltet Klammerregeln und die Bindungsstärke von Operatoren. Weiter wurden die Operatoren auf Kommutativität und Assoziativität untersucht. Wir wollen dies hier nicht nachvollziehen. Es sei aber angemerkt, dass die acht Operatoren tatsächlich die Basis einer kompletten, in sich geschlossenen Algebra bilden.

Bei diesem Aufbau tritt noch ein kleines Problem auf: Relationen bestehen nicht nur aus dem Rumpf, sondern auch aus dem Kopf. In einer Relation kann der gleiche Kopf nicht zweimal auftauchen. Daher sind insbesondere beim Join und beim kartesischen Produkt gelegentlich Umbenennungen des Kopfes erforderlich. Dies ist aber nicht mit unseren acht Operatoren möglich. Wir müssen hier noch einen „Operator“ *Rename* einführen.

Weiter sei erwähnt, dass die acht Operatoren nicht alle notwendig sind. Genaugenommen reichen bereits die folgenden fünf Operatoren aus: Vereinigung, Differenz, Kartesisches Produkt, Restriktion und Projektion. Die verbleibenden drei lassen sich aus diesen fünf Operatoren herleiten: Besteht die

Relation *A* aus den beiden (zusammengesetzten) Attributen *X* und Y_1 und *B* aus den beiden (zusammengesetzten) Attributen Y_1 und *Z*, so gilt:

```
A INTERSECT B  =  A MINUS (A MINUS B)
A JOIN B       =  ( (A TIMES B) WHERE A.Y1= B.Y1 ) [X, A.Y1, Z]
A DIVIDEBY B   =  A[X] MINUS ( (A[X] TIMES B) MINUS A ) [X]
```

Es sei als Übung empfohlen, diese drei Gleichungen selbst nachzuvollziehen. Bei der Division ist zu berücksichtigen, dass die Relation *B* nur aus dem Attribut Y_1 bestehen darf, *Z* demnach leer ist. Wir werden im nächsten Kapitel sehen, dass mit der Sprache SQL diese fünf und damit letztlich alle acht Operatoren nachgebildet werden können. Die Theorie zu dieser Algebra ist folglich auch für SQL gültig: Es sind alle nur denkbaren Ergebnistabellen erstellbar.

Zuletzt sei noch erwähnt, dass die relationale Algebra nicht den einzigen Zugang zum Relationenmodell ermöglicht. Bereits parallel zur relationalen Algebra wurde das Relationenkalkül entwickelt. In der relationalen Algebra werden die Zugriffe konstruiert, es wird also ein konstruktives Verfahren angewendet. Im Relationenkalkül wird beschrieben, wie mit der Datenbank und ihren Relationen gearbeitet werden kann. Hierzu wird eine Zugriffssprache (bekannt ist etwa die Zugriffssprache QUEL) definiert. Rasch stellte sich heraus, dass beide Zugänge zum Relationenmodell gleich mächtig sind, siehe [Codd72]. Auch hier sei dem interessierten Leser das Buch von Date [Date95] zur Vertiefung empfohlen.

3.5 Zusammenfassung

Fassen wir in diesem Abschnitt die wichtigsten Ergebnisse dieses Kapitels kurz zusammen. Wir haben zunächst neue Begriffe im Zusammenhang mit relationalen Datenbanken kennengelernt bzw. definiert. Die wichtigsten waren: *(normalisierte) Relation, relationale Datenbank, Primärschlüssel* und *Fremdschlüssel*.

In Relationen existiert weder eine Anordnung noch gibt es doppelte Tupel. Primärschlüssel wiederum identifizieren jedes Tupel einer Relation eindeutig und Fremdschlüssel beziehen sich immer auf den Primärschlüssel einer anderen oder sogar der gleichen Relation. Sie bilden den „Kit“ zwischen den Relationen.

Die Wichtigkeit sowohl von Primär- als auch von Fremdschlüsseln wird noch durch die beiden Integritätsregeln verdeutlicht. Die erste bezieht sich auf den Primärschlüssel und verlangt, dass dieser immer einen definierten Wert enthalten muss. Die zweite Regel fordert, dass jeder Fremdschlüssel, falls er einen definierten Wert enthält, auf ein existierendes Tupel verweisen muss. Dies bedeutet, dass jeder Fremdschlüssel nur Werte annehmen darf, die bereits im Primärschlüssel der Relation stehen, auf die der Fremdschlüssel verweist. Es bedeutet ferner, dass entsprechende Primärschlüsselwerte nur geändert werden dürfen, wenn der dazugehörige Fremdschlüsselwert gleichzeitig mit angepasst wird. Die Verletzung einer der beiden Regeln würde auf Dauer immer zu inkonsistenten Datenbanken führen und ist daher verboten. Während die erste Integritätsregel meist intuitiv eingehalten wird, finden wir in kleineren, unter Zeitdruck geschriebenen Datenbanken immer wieder Verletzungen der zweiten Regel. In allen bekannten Fällen haben die Anwender dieser Datenbanken mit Inkonsistenzen und Integritätsproblemen zu kämpfen.

3.6 Übungsaufgaben

1) Erklären Sie folgende Begriffe: Tupel, Attribut, Relation, Gebiet, Grad, Kardinalität.
2) Der Begriff „Primärschlüssel“ wird sowohl in der Datenorganisation als auch in relationalen Datenbanken verwendet. Sie sind aber in beiden Umgebungen nicht gleich definiert. Geben Sie jeweils mindestens einen Unterschied und eine Gemeinsamkeit an.
3) Geben Sie den Primärschlüssel der Relation *VerkaeuferProdukt* aus Tab. 13 an.
4) Welche Attribute eines neuen Tupels müssen immer mindestens eingetragen werden? Denken Sie dabei an die erste Integritätsregel.
5) Geben Sie die Primärschlüssel aller Relationen der Beispieldatenbank *Radl* aus Anhang A an. Suchen Sie auch alle alternativen Schlüssel.
6) Geben Sie alle Fremdschlüssel der Beispieldatenbank *Radl* aus Anhang A an.
7) Nehmen wir an, in der Beispieldatenbank *Radl* aus Anhang A gelte für alle Fremdschlüssel die Eigenschaft *on delete cascade*. Geben Sie alle Tupel an, die kaskadierend gelöscht werden, wenn
   a) der Eintrag von Fr. Köster in der Relation *Personal*
   b) der Eintrag 500001 in der Relation *Teilestamm* gelöscht wird.
8) Bilden Sie eine Verbindung der Relation *Personal* aus Tab. 18 auf sich, also *Personal JOIN Personal*. Hierbei ist das Attribut *Vorgesetzt* der einen Relation mit dem Attribut *Persnr* der anderen Relation zu verknüpfen.

9) Es seien zwei Relationen *A* und *B* mit der Kardinalität *M* respektive *N* gegeben. Geben Sie jeweils die minimale und die maximale Kardinalität der folgenden Ergebnisrelationen an (in Abhängigkeit von *M* und *N*):

A UNION B	A JOIN B	A MINUS B
A TIMES B	A INTERSECT B	

4 Die Datenbankzugriffssprache SQL

Im letzten Kapitel wurden relationale Datenbanken vorgestellt. Wir haben gesehen, dass diese Datenbanken ausschließlich aus Tabellen aufgebaut sind, die mit Daten gefüllt werden. Wir haben aber auch gesehen, dass Zusammenhänge zwischen diesen Tabellen mittels Fremdschlüssel hergestellt werden, woraus sich schließlich vielfältige Möglichkeiten der Abfrage und Manipulation ergeben. Aufgabe dieses Kapitels ist es nun zu zeigen, wie mit Hilfe einer Zugriffssprache auf diese Daten in ihrer gesamten Vielfalt zugegriffen werden kann, und wie diese Daten manipuliert werden. Die in Kapitel 3 vermittelten wichtigen theoretischen Grundlagen werden dabei in die Praxis umgesetzt.

Im ersten und umfangreichsten Abschnitt beschäftigen wir uns ausschließlich mit Abfragen, zunächst einfacheren, die sich auf eine Relation beschränken, und dann zunehmend komplexeren. Unterabfragen und Verbindungen (Joins) bilden einen wichtigen Teil, denn gerade die Verknüpfungen der Relationen untereinander ermöglichen erst die vielfältigen Abfragemöglichkeiten auf die Inhalte einer Datenbank.

Im zweiten Abschnitt behandeln wir die Mutationsbefehle. Manipulationen an sich sind relativ einfach, bedingen aber immer eine vorherige Suche des zu ändernden Feldes. Manipulationen beinhalten daher meist auch Abfragen. Doch diese sollten nach dem ersten Abschnitt keine Schwierigkeiten mehr bereiten.

Als Zugriffssprache wird SQL (Structured Query Language) verwendet. SQL war zunächst eine Sprache für den Endbenutzer. Durch die inzwischen weitverbreiteten grafischen Oberflächen ist dies heute jedoch nicht mehr gegeben. Als Schnittstellensprache und als Zugriffssprache auf andere Datenbanken hat SQL dafür eine um so größere Bedeutung erlangt. SQL hat sich, insbesondere seit der ersten Normierung im Jahre 1986, zur wichtigsten Standardsprache für Datenbanken entwickelt. Diese erste Norm (im weiteren als SQL-1 Norm bezeichnet) wurde 1989 ergänzt (SQL-1+ Norm) und 1992 erheblich erweitert (SQL-2 Norm). Diese SQL-2 Norm gliedert sich wiederum in drei Implementierungsstufen: den Entry-, den Intermediate- und den Full-Level. Während sich in der Praxis der SQL-1 Standard einer weiten Durchdringung in den relationalen Datenbanksprachen erfreut, ist der SQL-2 Standard bisher meist nur im Entry-Level realisiert. Aus diesem Grund, vor allem

aber auch wegen des immensen Umfangs der SQL-2 Norm (die ISO-Norm umfasst mehr als 600 Seiten!), beschränken wir uns hier auf die inhaltlich wichtigen Teile dieser Norm. Unser Ziel ist es also, mit SQL typische Zugriffe auf Datenbanken vorzunehmen, und nicht, SQL in seiner Gesamtheit vorzustellen. Dies schließt ein, dass die hier behandelte SQL-Syntax keinen Anspruch auf Vollständigkeit besitzt.

Seit mehreren Jahren ist eine SQL-3 Norm in Entwicklung. Diese neue Norm setzt auf dem Intermediate-Level von SQL-2 auf und beinhaltet vorwiegend objektrelationale Erweiterungen. Wiederum sind diese Erweiterungen enorm (die derzeitige Vorlage umfasst über 1300 Seiten, siehe [SQL3]). Um verschiedene objektorientierte Ansätze übernehmen zu können, mussten Kontrollstrukturen (Schleifen- und Verzweigungsstrukturen) in die Syntax mitaufgenommen werden. Auf einige dieser neuen Möglichkeiten von SQL-3 werden wir in Abschnitt 11.2 eingehen.

Eine auf dieses Buch zugeschnittene SQL-Syntax finden wir in Anhang B, die komplette Syntax von SQL-2 ist in der ISO-Norm (siehe [SQL92]) und in [MeSi93] beschrieben. Leider existieren in der Praxis viele SQL-Dialekte, die sich von der Norm, wenn auch meist nur geringfügig, unterscheiden. Es kann daher sein, dass das hier vorgestellte SQL nur nach einigen Anpassungen auf den Implementierungen einzelner Hersteller ablauffähig ist. Bei Abweichungen in Oracle und MS-Access werden entsprechende Hinweise gegeben. Es empfiehlt sich aber immer, auch auf die Handbücher der einzelnen Softwarehersteller zuzugreifen.

Im letzten Abschnitt dieses Kapitels zeigen wir noch kurz den Zusammenhang zur relationalen Algebra auf. Insbesondere werden wir nachweisen, dass alle acht Operationen der relationalen Algebra in SQL nachgebildet werden können, dass also SQL im Sinne der relationalen Algebra eine vollständige relationale Zugriffssprache ist.

4.1 Der Abfragebefehl Select

In SQL steht zur Suche und Abfrage von Daten einer Datenbank genau ein Befehl zur Verfügung, der *Select*-Befehl. Wenn wir an Kreuzprodukte, Vereinigungen oder Joins denken, dann werden wir uns zunächst fragen, ob SQL mit diesem einen Befehl all diese Abfragen überhaupt ermöglichen kann. Des Rätsels Lösung ist einfach: Der *Select*-Befehl besitzt eine umfangreiche Syntax. Wir werden diese Syntax gegenüber der SQL-2 Norm etwas einschränken.

Wir gewinnen dadurch an Übersicht, ohne merkliche Beschränkungen beim Zugriff zu spüren.

Wir gehen in diesem Kapitel immer davon aus, dass die jeweils betrachtete Datenbank existiert und zum Bearbeiten so geöffnet ist, dass wir mit Hilfe von SQL-Befehlen direkt darauf zugreifen dürfen.

Betrachten wir die Datenbank mit den Relationen *Personal*, *Kunde* und *Auftrag* aus Tab. 18 bis Tab. 20. Benötigen wir eine Liste aller Mitarbeiter, so greifen wir auf die Relation *Personal* wie folgt zu:

```
SELECT * FROM Personal;
```

Da einschränkende Bedingungen fehlen, werden mit diesem Befehl alle Tupel (Zeilen) der Relation *Personal* ausgewählt und ausgegeben. Der Stern (‚∗‘) steht für „alle Attribute (Spalten)“, so dass alle Attribute der Personaltabelle aufgelistet werden. Soll nur der Name und Wohnort aller Mitarbeiter ausgegeben werden, so lautet der Befehl

```
SELECT Name, Ort FROM Personal;
```

Die (auch im weiteren) großgeschriebenen Wörter sind reservierte Bezeichner der Sprache SQL und dürfen daher als Variablennamen nicht verwendet werden; ebenso wird jeder SQL-Befehl mit einem Semikolon abgeschlossen. Analog zu anderen höheren Programmiersprachen ist die Schreibweise wahlfrei: Es sind also beliebig viele Leerzeichen oder Zeilenwechsel zwischen einzelnen Worten erlaubt. Auch unterscheidet SQL grundsätzlich nicht zwischen Groß- und Kleinschreibung. Folgende Schreibweise ist daher äquivalent zum letzten Befehl:

```
Select   name,          ORT
From     personal  ;
```

In beiden Fällen wird eine Liste der Namen und Wohnorte aller Mitarbeiter ausgegeben. Das genaue Ausgabeformat (wie Überschrift oder Spaltenbreite) hängt dabei von der verwendeten Datenbanksoftware ab.

In der Praxis werden Abfrageergebnisse meist nicht direkt ausgegeben, sondern zur weiteren Verarbeitung in Variablen eingelesen. Diese Möglichkeit werden wir in Kapitel 9 zum eingebetteten SQL aufzeigen. Doch zunächst machen wir uns mit der Arbeitsweise der Zugriffsbefehle näher vertraut.

Zum Kennenlernen und Testen dieser Befehle bieten glücklicherweise die meisten Datenbankanbieter Möglichkeiten zur Direkteingabe obiger Befehle: In Oracle steht für die direkte Eingabe von Befehlen das Produkt SQL*Plus zur Verfügung. Nach dem Start von SQL*Plus können obige *Select*-Befehle direkt eingegeben werden. Auch in MS-Access97 kann mittels der Menüfolge *Einfügen/Abfrage/Entwurfsansicht* eine Abfrage gestartet werden. Durch Drücken der rechten Maustaste auf das neu geöffnete Fenster erscheint das zugehörige Kontextmenü. Wird der Menüpunkt *SQL* ausgewählt, so wird ein SQL-Eingabefenster geöffnet. Sowohl für Oracle ab V7 als auch für MS-Access97 können vom Autor Programme bezogen werden, um die Beispieldatenbank *Radl* aus Anhang A zu installieren, abzufragen und zu manipulieren. Zu MS-Access wird zusätzlich eine einfache SQL-Eingabemöglichkeit angeboten. Alle folgenden SQL-Befehle können damit direkt nachvollzogen werden. Weitere Hinweise können Anhang D entnommen werden.

4.1.1 Der Aufbau des Select-Befehls

Sicherlich wurde schon eine wichtige Eigenschaft der Zugriffssprache SQL bemerkt: SQL fragt nicht, wie gesuchte Daten ermittelt, sondern ausschließlich, welche Daten benötigt werden. Für SQL spielt es keine Rolle, ob die Daten sequentiell abgespeichert oder über eine komplexe Indexverwaltung geordnet sind. Der Anwender wird diese Unterschiede höchstens in der Laufzeit bemerken, nicht aber an den SQL-Zugriffsbefehlen! Dies ist sicher eines der Geheimnisse, warum sich relationale Datenbanken und insbesondere SQL so schnell verbreiteten.

Für uns hat dies den Vorteil, dass wir uns nicht mit überflüssigem Ballast abgeben müssen und gleich die ganze Mächtigkeit des *Select*-Befehls kennenlernen können. Die Syntax für den Hauptteil des *Select*-Befehls lautet:

```
SELECT [ ALL | DISTINCT ]   Spaltenauswahlliste
 FROM          Tabellenliste
 [ WHERE       Bedingung ]
 [ GROUP BY    Spaltenliste
    [ HAVING   Bedingung ] ]
```

Hier und im Folgenden verwenden wir folgende Notation: Ausdrücke in e*ck*igen Klammern können je nach Wunsch weggelassen werden. Sind mehrere Angaben durch einen senkrechten Strich (‚|‘) getrennt, so darf nur genau eine dieser Angaben verwendet werden. Die in Großbuchstaben

geschriebenen Wörter sind reservierte Bezeichner in SQL und müssen bis auf Groß- und Kleinschreibung genauso im Befehl angegeben werden. Alle anderen Wörter werden entweder noch näher erläutert, oder es handelt sich um wahlfreie Namen. Wir verweisen zwecks Einzelheiten zur Syntax auf den Anhang B am Ende des Buches. Dort finden wir eine Zusammenfassung der Syntax aller in diesem Buch behandelten SQL-Befehle und weitere Einzelheiten zur Notation. Vervollständigen wir nun den *Select*-Befehl:

```
Select-Hauptteil
[  { UNION | EXCEPT | INTERSECT }  [ ALL ]
   Select-Hauptteil
   [ { UNION | EXCEPT | INTERSECT }  [ ALL ]  ... ]    ]
[ ORDER BY Ordnungsliste ]
```

Die in geschweiften Klammern stehende Auswahl heißt, dass genau eine dieser Angaben vorkommen muss. Ein kompletter *Select*-Befehl besteht somit aus beliebig vielen Hauptteilen, die über *Union*, *Except* oder *Intersect* miteinander verknüpft sind. Die drei Punkte innerhalb der eckigen Klammern zeigen auf, dass die vorhergehende Aufzählung fortgesetzt werden darf. Bei mehr als zwei Verknüpfungen empfiehlt sich das Setzen von runden Klammern, um die Eindeutigkeit der Verknüpfungsreihenfolge zu garantieren. Nach der Verknüpfung kann das Endresultat mittels der *Order-By*-Klausel noch geordnet werden.

Tab. 22 Logische Abarbeitungsreihenfolge des Select-Befehls

1	Alle in der Tabellenliste angegebenen Relationen werden über das Kreuzprodukt oder einen Join miteinander verknüpft.
2	Aus dieser verknüpften Relation werden die Tupel (Zeilen) ausgewählt, die die angegebene WHERE-Bedingung erfüllen: Eine Restriktion liegt vor.
3	Mittels der Auswahlliste am Anfang des *Select*-Befehls findet jetzt auf das bisherige Resultat eine Projektion auf die angegebenen Spalten statt.
4	Nun wird eine Gruppierung gemäß der *Group-By*-Klausel durchgeführt. Eine Gruppierung fasst dabei mehrere Tupel (Zeilen) zu einem Tupel zusammen, so dass die Ergebnisrelation weniger Tupel enthält.
5	Eine folgende *Having*-Klausel führt jetzt auf das Ergebnis der Gruppierung nochmals eine Restriktion auf bestimmte Tupel durch.
6	Alle so erstellten Hauptteile des *Select*-Befehls werden jetzt mittels *Union*, *Intersect* oder *Except* miteinander verknüpft.
7	Die Ergebnisrelation wird nach den Vorgaben in der Ordnungsliste der *Order-By*-Klausel sortiert.

Es sei erwähnt, dass diese Syntax des Select-Befehls die SQL-1 Norm voll enthält, während weniger wichtige Teile der SQL-2 Norm weggelassen wurden. Manche Implementierungen, wie etwa MS-Access, kennen nur die *Union*-Verknüpfung. In Oracle muss lediglich der Bezeichner *Except* durch *Minus* ersetzt werden.

Die Funktionsweise des *Select*-Befehls ist zusammen mit der Abarbeitungsreihenfolge des Befehls in der Norm eindeutig festgelegt, auch wenn aus Optimierungsgründen die tatsächliche Implementierung davon abweichen kann, vorausgesetzt natürlich, dass das Endergebnis das Gleiche bleibt. Tab. 22 enthält die logische Abarbeitungsreihenfolge. Die Teile, die in dem zu betrachtenden *Select*-Befehl nicht vorkommen, werden in der Tabelle einfach übersprungen.

In den nächsten Unterabschnitten lernen wir die einzelnen Teile des *Select*-Befehls im Detail kennen. Wegen der Komplexität dieses Befehls ist es nicht verkehrt, die Abarbeitungsreihenfolge aus Tab. 22 stets vor Augen zu haben. Zusätzlich wird ein nochmaliges Studieren dieser Tabelle am Ende dieses Abschnitts zum *Select*-Befehl dringend empfohlen.

4.1.2 Die Select- und From-Klausel

Diese beiden Klauseln sind zwingende Bestandteile jedes *Select*-Befehls. Die *From*-Klausel verknüpft die in der Tabellenliste angegeben Relationen mittels des Kreuzprodukts und die *Select*-Klausel führt eine Projektion auf die in der Auswahlliste angegebenen Spalten durch. Wir stellen hier die Einzelheiten vor, ohne jedoch alle Möglichkeiten von SQL-2 voll auszuschöpfen. Beginnen wir mit der *From*-Klausel. Die Tabellenliste besitzt grob folgende Gestalt:

```
Tabellenname [ [ AS ] Aliasname ]  [ , ... ]
```

Es werden demnach die gewünschten Relationen durch Kommata getrennt angegeben. Wie wir noch sehen werden, ist es gelegentlich notwendig, den Relationen einen Aliasnamen zwecks eindeutiger Identifizierung zu geben. Dieser Aliasname darf ein beliebiger SQL-Name sein. Die SQL-Namenskonvention entspricht der PASCAL-Namensgebung: Namen beginnen mit einem Buchstaben, eventuell fortgesetzt durch weitere Buchstaben, Zahlen oder Unterstrichzeichen (‚_‘). Der Trennbezeichner *As* ist ein reines Füllwort und darf weggelassen werden (in Oracle ist er gar nicht implementiert). Die so angege-

benen Tabellen werden über das Kreuzprodukt miteinander verknüpft. Erweiterungen in SQL-2 erlauben darüber hinaus auch Verknüpfungen wie den Join und die Vereinigung. Zwischen zwei Tabellen muss dann statt des Kommas der Bezeichner

```
[ NATURAL ] [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] | UNION ]
 JOIN
```

stehen. Hier handelt es sich um den inneren Join, drei speziellen äußeren Joins und um eine Join-Vereinigung (*Union*-Join). Das Wort *Outer* ist ein reines Füllwort. Das Wort *Natural* verweist darauf, dass die an der Verknüpfung beteiligten Attribute in beiden beteiligten Relationen den gleichen Namen besitzen (natürlicher Join). Wird kein *Join*-Bezeichner angegeben, so wird standardmäßig ein *Inner-Join* angenommen.

Wo diese Bezeichner noch nicht implementiert sind, wie etwa in Oracle, müssen diese mittels Kreuzprodukt und Restriktion nachgebildet werden. MS-Access97 unterstützt genau die drei Bezeichner *Inner Join*, *Left Join* und *Right Join*, was in der Praxis meist ausreicht. Das Arbeiten mit *Join*-Verknüpfungen und Beispiele lernen wir im Unterabschnitt 4.1.6 kennen.

Die *Select*-Klausel enthält eine Liste von Spaltenausdrücken, die durch Kommata voneinander getrennt sind:

```
Spaltenausdruck [ [ AS ] Aliasname ] [ , ... ]
```

Wie wir sehen, sind nicht nur Spaltennamen erlaubt, sondern beliebige Spaltenausdrücke. Wir können somit Attribute (Spalten) im Rahmen syntaktisch korrekter Ausdrücke beliebig miteinander in Verbindung bringen. Betrachten wir dazu wieder die Relation *Personal* aus Tab. 18. Wenn wir uns für die Namen aller Mitarbeiter und deren Jahresgehalt interessieren, so erhalten wir diese Angaben durch den Befehl

```
SELECT Name, 12 * Gehalt AS Jahresgehalt
 FROM   Personal ;
```

Ausgegeben wird nun eine Tabelle mit zwei Spalten, deren Spaltenbezeichnungen *Name* und *Jahresgehalt* lauten. Hier wird die Bedeutung des Aliasnamen schnell klar: Soll auf den Ausdruck *12*Gehalt* später hingewiesen werden, so ist dies nur über einen erlaubten Namen möglich, in diesem Fall über den Aliasnamen *Jahresgehalt*.

Die Spaltenausdrücke dürfen als Variablen alle vorkommenden Spaltennamen beinhalten. Auch dürfen alle in SQL implementierten Funktionen verwendet werden. Neben numerischen Ausdrücken sind auch Zeichenketten- und Datumsausdrücke erlaubt. Der Vielfalt sind fast keine Grenzen gesetzt. Wir werden an Beispielen einige Möglichkeiten aufzeigen. Übrigens ist der Bezeichner *As* auch hier nur ein reines Füllwort und darf einfach weggelassen werden.

Sollen alle Attribute der in der *From*-Klausel angegebenen Relationen ausgegeben werden, so müssen diese nicht einzeln aufgelistet werden. Als Abkürzung dient hierfür der Stern (‚*'). Sowohl dieser Stern als auch jede Spaltenangabe kann noch qualifiziert werden, d.h. es kann noch angegeben werden, auf welche Tabelle sich eine Spalte bezieht. Bei gleichen Spaltennamen in mehreren Tabellen ist diese Qualifizierung sogar zwingend erforderlich. Hierzu wird vor die Spaltenbezeichnung der Tabellenname geschrieben, gefolgt von einem Punkt. Betrachten wir ein Beispiel. Alle drei folgenden *Select*-Befehle sind gleichwertig:

```
SELECT * FROM Personal, Auftrag ;

SELECT Personal.*, Auftrag.* FROM Personal, Auftrag ;

SELECT Personal.Persnr, Name, Ort, Vorgesetzt, Gehalt, AuftrNr, Datum,
       Kundnr, Auftrag.Persnr
 FROM Personal, Auftrag ;
```

Die drei *Select*-Befehle liefern jeweils das Kreuzprodukt der beiden Relationen *Personal* und *Auftrag* zurück, wobei alle Spalten ausgegeben werden. Beachten Sie im dritten *Select*-Befehl, dass die Personalnummer in beiden Relationen durch den gleichen Namen identifiziert wird und daher zwecks Unterscheidung qualifiziert werden muss.

In SQL sind einige skalare Funktionen definiert, die innerhalb von numerischen Ausdrücken, Zeichenketten- und Datumsausdrücken erlaubt sind. Häufige Verwendung finden die Funktionen *Upper, Lower* oder *Trim*. Die ersten beiden Funktionen wandeln Kleinbuchstaben einer Zeichenkette in Großbuchstaben um, und umgekehrt, und die dritte Funktion entfernt führende und schließende Leerzeichen einer Zeichenkette. Weitere skalare Funktionen seien hier unerwähnt, da sie in vielen Datenbanken nicht implementiert sind. So existiert beispielsweise in Oracle die Funktion *Trim* nicht. Als Ersatz dienen hier die Funktionen *LTrim* und *RTrim*, die führende respektive schließende Leerzeichen entfernen. Dafür haben praktisch alle Datenbankhersteller noch

dutzende weiterer Funktionen implementiert. Es sei auf die Handbücher dieser Hersteller verwiesen.

Zu den skalaren Funktionen kommen noch fünf Statistikfunktionen, auch Aggregatfunktionen genannt, die wegen ihrer Besonderheiten alle in Tab. 23 aufgeführt sind. Diese Funktionen ermitteln aus einem Spaltenausdruck statistische Werte, die aus allen Tupeln (Zeilen) der Relation errechnet werden.

Tab. 23 Statistikfunktionen in SQL

AVG	Average	Durchschnittswert, ermittelt über alle Zeilen
COUNT	Count	Anzahl aller Zeilen
MAX	Maximum	Maximalwert aller Zeilen
MIN	Minimum	Minimalwert aller Zeilen
SUM	Sum	Summenwert, summiert über alle Zeilen

Als Parameter kann entweder eine Spalte oder ein beliebiger Spaltenausdruck angegeben werden. Sollte sich das Gehalt eines Mitarbeiters aus seinem Monatsgehalt und aus einer jährlichen Einmalzahlung, abhängig von seiner Beurteilung, zusammensetzen, so liefert der erste der beiden folgenden *Select*-Befehle eine Auflistung aller Mitarbeiter mit Angabe ihrer Jahresgehälter. Der zweite Befehl summiert die Gehälter auf und gibt so die Jahresausgaben der Firma für Personalkosten aus:

```
SELECT  Persnr, Name, 12*Gehalt + 1000 * (6 - Beurteilung)
 FROM   Personal ;

SELECT  SUM (12 * Gehalt +1000 * ( 6- Beurteilung)) AS Personalkosten
 FROM   Personal ;
```

Mit Hilfe der Statistik-Funktionen ist es einfach möglich, Statistiken aus den vorhandenen Relationen abzuleiten. Wichtig zu wissen ist, dass sich diese Statistikfunktionen auf alle Zeilen einer Relation beziehen, und dass als Ergebnis genau ein Wert erzeugt wird. Diese Funktionen dürfen daher nur alleine oder zusammen mit weiteren Statistikfunktionen in der Auswahlliste vorkommen! Dies sollte sofort klar sein, wenn wir bedenken, dass das Ergebnis eines *Select*-Befehls wieder eine meist mehrzeilige Relation ist. Eine Statistikfunktion in der Auswahlliste liefert aber nur ein einziges Tupel zurück. Eine Spaltenangabe und eine Statistikfunktion gleichzeitig in der Spaltenliste eines *Select*-Befehls vertragen sich daher nicht. Beachten wir weiterhin die

Abarbeitungsreihenfolge des *Select*-Befehls. Die Statistikfunktionen beziehen sich immer auf das in der *From*-Klausel ermittelte Kreuzprodukt!

Die *Count*-Funktion erlaubt noch eine Abkürzung: Da diese Funktion immer alle Tupel zählt, und dieser Wert für jedes Attribut gleich groß ist, kann als Parameter einfach nur der Stern (‚∗‘) eingesetzt werden. Zum Beispiel liefert der folgende Befehl die Anzahl der in der Tabelle gespeicherten Mitarbeiter:

```
SELECT COUNT(*) AS Mitarbeiteranzahl FROM Personal ;
```

Zuletzt wollen wir nicht vergessen, dass in der *Select*-Klausel noch die beiden Bezeichner *Distinct* und *All* erlaubt sind. Wird *Distinct* angegeben, so werden gleiche Ergebniszeilen nur einmal ausgegeben, d.h. dass sich alle ausgegebenen Zeilen voneinander unterscheiden. Mit der Angabe *All* werden alle erzeugten Ergebnisse einzeln ausgegeben. Diese Angabe ist standardmäßig voreingestellt und kann daher weggelassen werden. Eine Liste der Wohnorte aller Mitarbeiter liefern beide folgenden Befehle. Allerdings wird im zweiten Befehl jeder Wohnort, auch der Ort *Regensburg*, nur einmal vorkommen:

```
SELECT Ort FROM Personal ;
SELECT DISTINCT Ort FROM Personal ;
```

Interessanterweise sind die Angaben *Distinct* und *All* auch bei allen Statistikfunktionen erlaubt. Wiederum ist *All* standardmäßig vorgegeben. Wird *Distinct* verwendet, so werden gleiche Ausdrücke nur einmal gezählt. Wenn wir beim letzten Beispiel bleiben, so liefert

```
SELECT COUNT ( DISTINCT Ort ) FROM Personal ;
```

die Anzahl aller unterschiedlichen Wohnorte der Mitarbeiter. An diesem Beispiel erkennen wir auch die Verwendung des Bezeichners *Distinct* in den Statistikfunktionen: Der Bezeichner wird, durch Leerzeichen getrennt, vor den eigentlichen Parameter geschrieben. Es ist wichtig zu wissen, dass die drei folgenden Befehle jeweils unterschiedliche Ergebnisse liefern:

```
SELECT COUNT (*) FROM Personal ;
SELECT COUNT ( Vorgesetzt ) FROM Personal ;
SELECT COUNT ( DISTINCT Vorgesetzt ) FROM Personal ;
```

Der erste Befehl zählt alle Tupel der Relation *Personal*, der zweite überliest dagegen die beiden Tupel, die im Attribut *Vorgesetzt* einen *Null*-Eintrag besit-

zen und der dritte zählt nur die unterschiedlichen Einträge im Attribut *Vorgesetzt*. Der erste *Select*-Befehl liefert daher den Wert 9, der zweite den Wert 7 und der dritte den Wert 2 zurück.

Wir hätten damit die wichtigsten Fälle der *Select*- und *From*-Klauseln behandelt. Weitere Anwendungen, insbesondere der Statistikfunktionen, ergeben sich durch die anderen Klauseln des *Select*-Befehls und werden dort behandelt.

4.1.3 Die Where-Klausel

Während die bisher behandelten *Select*- und *From*-Klauseln in jedem *Select*-Befehl vorkommen müssen, sind die *Where*- und die folgenden Klauseln wahlfrei. Wegen seiner wichtigen Bedeutung zur Einschränkung einer Relation auf bestimmte Tupel (Restriktion) kommt die *Where*-Klausel in der Praxis trotzdem in fast allen *Select*-Befehlen vor.

Die Syntax des *Select*-Befehls zeigt, dass die *Where*-Klausel nur eine Bedingung enthält. Nur diejenigen Tupel (Zeilen) werden selektiert, die diese Bedingung erfüllen. Alle anderen Tupel erscheinen in der Ergebnisrelation nicht. Zu beachten ist, dass logisch gesehen erst die *From*-, dann die *Where*- und zuletzt die *Select*-Klausel ausgewertet wird! Statistikfunktionen beziehen sich demnach nur auf die Tupel aller in der *From*-Klausel angegebenen Relationen, die durch die *Where*-Klausel ausgewählt wurden. Beispielsweise liefert der folgende Befehl als Ergebnis das kleinste Gehalt größer als 5000 DM, in unserer Beispielrelation also 5500 DM.

```
SELECT  MIN ( Gehalt )
 FROM    Personal
 WHERE Gehalt > 5000 ;
```

Die Bedingung in der *Where*-Klausel liefert einen booleschen Wert, der je nach Tupel *wahr* oder *falsch* sein wird. Beliebige boolesche Ausdrücke sind zugelassen, insbesondere Verknüpfungen mit den booleschen Operatoren *Not*, *And* und *Or*. Wie in allen Programmiersprachen besitzen diese Operatoren unterschiedliche Bindungsstärke, *Not* bindet am stärksten, *Or* am geringsten. Weitere wichtige Operatoren in SQL (neben den booleschen) sind in Tab. 24 aufgeführt.

Tab. 24 Operatoren in der WHERE-Klausel

Vergleichsoperatoren	< , <= , > , >= , = , <>
Intervalloperator	[NOT] BETWEEN ... AND
Enthaltenoperator	[NOT] IN
Auswahloperatoren	ALL , ANY , SOME
Ähnlichkeitsoperator	[NOT] LIKE
Existenzoperator	EXISTS
Eindeutigkeitsoperator	UNIQUE
Nulloperator	IS [NOT] NULL

Neu ist in SQL-2 darüberhinaus der *Match*-Operator, auf den wir aber nicht weiter eingehen wollen. Die Vergleichsoperatoren sollten keine Schwierigkeiten bereiten. Wir haben sie bereits in unserem letzten Beispiel angewendet. Auch der Intervalloperator ist relativ leicht verständlich. Hier wird angegeben, ob ein Ausdruck zwischen zwei Angaben liegt bzw. nicht liegt. Folgende beiden Bedingungen sind äquivalent:

```
A BETWEEN B AND C
A >= B AND A <= C
```

In der letzten Bedingung sind keine Klammern erforderlich, da alle in Tab. 24 genannten Operatoren stärker als die booleschen Operatoren binden. Selbstverständlich sind aber runde Klammern zugelassen, was meist die Übersichtlichkeit erhöht und Fehler vermeiden hilft. Auch folgende Bedingungen sind gleichwertig:

```
A NOT BETWEEN B AND C
NOT A BETWEEN B AND C
```

Im zweiten Fall ist *Not* der boolesche Operator, der den folgenden Ausdruck verneint. Wollen wir nun beispielsweise alle Mitarbeiter aufzählen, deren Gehalt zwischen 3000 und 4000 DM liegt, so schreiben wir kurzerhand

```
SELECT *
 FROM   Personal
 WHERE Gehalt BETWEEN 3000 AND 4000 ;
```

Während der Intervalloperator innerhalb eines Intervalls alle Werte auswählt, entnimmt der Enthaltenoperator die gültigen Werte einer Kommaliste. Eine Kommaliste ist in SQL eine durch Kommata getrennte Aufzählung, die

zwischen runden Klammern steht. Sollen beispielsweise nur diejenigen Mitarbeiter ausgegeben werden, die genau 3200, 3800 oder 4600 DM verdienen, so sollte der folgende Befehl verwendet werden:

```
SELECT *
 FROM   Personal
 WHERE Gehalt IN (3200, 3800, 4600) ;
```

Das Komplement der Ausgabe erhalten wir, wenn wir entweder zwischen dem Bezeichner *Where* und dem Namen *Gehalt* oder zwischen *Gehalt* und *In* den Bezeichner *Not* einfügen. Im ersten Fall liegt der entsprechende boolesche Operator, im zweiten Fall der erlaubte Zusatz des Enthaltenoperators vor.

Weitere ähnlich wirkende Operatoren sind die Auswahloperatoren *All*, *Some* und *Any*. Obiger Befehl hätte auch wie folgt geschrieben werden können:

```
SELECT *
 FROM   Personal
 WHERE Gehalt = ANY (3200, 3800, 4600) ;
```

Einem Auswahloperator geht immer ein Vergleichsoperator voraus, und es folgt eine Kommaliste. Obiger *Any*-Operator ist nichts weiter als eine Abkürzung für

```
WHERE Gehalt = 3200 OR Gehalt = 3800 OR Gehalt = 4600 ;
```

Dies gilt analog auch für die anderen Vergleichsoperatoren in Verbindung mit *Any*: Alle Werte in der Kommaliste werden mit dem linken Ausdruck verglichen und mit dem booleschen Operator *Or* miteinander verknüpft. *Some* ist nur eine andere Bezeichnung für *Any*, und der *All*-Operator verknüpft die einzelnen Ausdrücke mit dem booleschen Operator *And* statt mit *Or*. Beachten Sie bitte, dass die Verneinung von

```
Gehalt = ANY (3200, 3800, 4600)
```

durch folgende Bedingungen

```
NOT Gehalt = ANY (3200, 3800, 4600)
Gehalt <> ALL (3200, 3800, 4600)
Gehalt = NOT ALL (3200, 3800, 4600)
```

wiedergegeben wird, nicht jedoch durch

```
Gehalt <> ANY (3200, 3800, 4600)
Gehalt = NOT ANY (3200, 3800, 4600)
```

Die letzten beiden Bedingungen sind immer wahr! Dies wird uns klar, wenn wir *Any* durch die Langschreibweise mit der booleschen Verknüpfung *Or* ersetzen. Wir erkennen dann, dass die Bedingung wahr ist, wenn das Gehalt mindestens eines der drei angegebenen Gehälter nicht annimmt, was natürlich immer zutrifft.

In der Praxis ist es häufig erforderlich, Zeichenketten zu analysieren. Hier leistet der Ähnlichkeitsoperator *Like* wertvolle Hilfe. Wir können damit Zeichenketten vergleichen, wobei eine Wildcard-Syntax zur Verfügung steht. Wollen wir etwa alle Mitarbeiter ausgeben, die in ihrem Vornamen den Namen *Heinz* enthalten, so können wir schreiben

```
SELECT *
 FROM   Personal
 WHERE Name LIKE '%Heinz%' ;
```

Das Prozentzeichen steht anstelle beliebig viele Zeichen in der Zeichenkette. Es wird daher jedes Tupel ausgegeben, dessen Name mit beliebig vielen Zeichen beginnt, dann den Namen *Heinz* enthält und von beliebig vielen weiteren Zeichen gefolgt wird. Der Begriff *beliebig viele Zeichen* schließt die leere Zeichenkette mit ein. Das Wildcardzeichen ‚%' darf in der Zeichenkette, die dem *Like*-Operator folgt, mehrfach vorkommen. Der Zeichenkettenausdruck vor dem *Like*-Operator hingegen darf keine Wildcardzeichen enthalten. Neben dem Prozentzeichen besitzt auch das Unterstreichzeichen (‚_') eine spezielle Bedeutung. Es ersetzt genau ein beliebiges Zeichen.

Der Nulloperator dient dazu, Tupel auszuwählen, die *Null*-Werte enthalten. Mit folgendem Befehl können wir etwa alle Mitarbeiter ausgeben, die keinem Vorgesetzten zugeordnet sind:

```
SELECT *
 FROM   Personal
 WHERE Vorgesetzt IS NULL ;
```

Das Komplement davon, also alle Mitarbeiter, die einen Vorgesetzten haben, erhalten wir durch eine der beiden folgenden Bedingungen:

```
NOT Vorgesetzt IS NULL
Vorgesetzt IS NOT NULL
```

In den *Where*-Bedingungen dürfen auch alle von SQL angebotenen Funktionen verwendet werden, aber erst seit SQL-2 auch die Statistikfunktionen aus Tab. 23. Die immense Bedeutung der *Where*-Klausel ergibt sich aber erst dadurch, dass hier auch Unterabfragen erlaubt sind. Zeigen wir die Möglichkeiten der Unterabfrage gleich an einem Beispiel. Gesucht ist der Mitarbeiter, der am meisten verdient. Wir erhalten ihn mit dem folgenden *Select*-Befehl:

```
SELECT  *
 FROM    Personal
 WHERE Gehalt = ( SELECT MAX(Gehalt) FROM Personal ) ;
```

Dieser Befehl bedarf einer Erläuterung. Um ihn leichter zu verstehen, dürfen wir grundsätzlich davon ausgehen, dass logisch gesehen Tupel für Tupel (Zeile für Zeile) nacheinander einzeln daraufhin untersucht wird, ob die angegebene Bedingung erfüllt ist. Enthält die Bedingung eine Unterabfrage, so wird diese für jede Zeile einzeln ausgeführt. In unserem Beispiel wäre dies inperformant, da die Unterabfrage ein von der betrachteten Zeile unabhängiges Ergebnis liefert. SQL-Implementierungen werden hier sicherlich optimieren. Zum besseren Verständnis können und dürfen wir aber die logische Sichtweise beibehalten. Die Variable *Gehalt* bezieht sich beim ersten Vorkommen auf die Hauptabfrage und beim zweiten Auftreten auf die Unterabfrage, wo das maximale Gehalt berechnet wird.

Die Gültigkeit von variablen Bezeichnern (Spalten- und Tabellennamen) erstreckt sich immer auf die Umgebung des Vorkommens und alle darin enthaltenen Unterabfragen. Ein lokaler Bezeichner einer Unterabfrage ist in einer übergeordneten Abfrage grundsätzlich nicht sichtbar. Sollten ein lokaler Bezeichner und ein übergeordneter Bezeichner den gleichen Namen besitzen, so überdeckt der lokale immer den globalen. Gegebenenfalls muss der übergeordnete Bezeichner qualifiziert werden. Hier verhält sich SQL analog zu anderen höheren Programmiersprachen.

Die Abfragesprache SQL ist sehr mächtig. Häufig existieren daher mehrere Möglichkeiten, eine Abfrage zu formulieren. Im obigen Beispiel hätten wir den *Select*-Befehl auch mit Hilfe des Operators *All* schreiben können??:

```
SELECT  *
 FROM    Personal
 WHERE Gehalt >= ALL ( SELECT Gehalt FROM Personal ) ;
```

Die Unterabfrage liefert hier eine Kommaliste von Zahlen, nämlich alle Gehälter. Ausgewählt wird nun nur dasjenige Tupel, dessen Gehalt größer oder gleich als alle diese Gehälter ist. Dies ist aber genau die Definition des Maximums.

Natürlich darf eine Unterabfrage weitere Unterabfragen enthalten, wobei zu beachten ist, dass in SQL-1 die Unterabfrage immer auf der rechten Seite des Operators stehen muss. SQL-2 besitzt diese Einschränkung nicht mehr und erlaubt außerdem weitere Möglichkeiten in der *Where*-Klausel. Die Behandlung all dieser Erweiterungen in SQL-2 würde den Rahmen dieses Kapitels sprengen. Da diese Erweiterungen nicht die Funktionalität erhöhen, sondern meist nur die *Select*-Befehle vereinfachen oder systematisieren, wollen wir auf diese Erweiterungen verzichten.

Betrachten wir statt dessen ein etwas komplexeres Beispiel: Wir wollen alle Mitarbeiter ausgeben, die weniger als Frau Rank (*Persnr*: 3) verdienen. Diese Ausgabe erhalten wir mit Hilfe einer Unterabfrage, aber auch mit Hilfe eines Kreuzprodukts. Beide gleichwertigen Möglichkeiten sind angegeben:

```
SELECT  *
 FROM    Personal
 WHERE Gehalt <  ( SELECT Gehalt FROM Personal
                       WHERE Persnr = 3 ) ;

SELECT  P1.*
 FROM    Personal AS P1, Personal AS P2
 WHERE P1.Gehalt < P2.Gehalt  AND  P2.Persnr = 3 ;
```

Beide *Select*-Befehle liefern das gleiche Ergebnis. Im zweiten Befehl haben wir ein Kreuzprodukt einer Relation auf sich selbst verwendet. Da in diesem Fall alle Attribute zweimal mit gleichem Namen existieren, müssen wir sie qualifizieren. Dies funktioniert hier wegen der gleichen Relationennamen nur mit Hilfe von Aliasnamen.

Zuletzt lernen wir nun noch den Existenzoperator *Exists* kennen. Mit diesem Operator wird überprüft, ob eine Unterabfrage mindestens ein Tupel zurückliefert. Wir können daher den letzten *Select*-Befehl auch mit Hilfe des *Exists*-Operators formulieren:

```
SELECT  *
 FROM    Personal AS P1
 WHERE EXISTS  (  SELECT * FROM Personal
                       WHERE Persnr = 3  AND  P1.Gehalt < Gehalt ) ;
```

Wir haben bereits erwähnt, dass zumindest logisch gesehen Zeile für Zeile überprüft wird, ob die *Where*-Bedingung erfüllt ist. Dies ist jetzt das erste Beispiel, in der die Unterabfrage zu jeder Zeile ein anderes Ergebnis liefern kann. Dies hängt damit zusammen, dass die Variable *P1.Gehalt* in der Unterabfrage verwendet wird, also das Gehalt der momentan betrachteten Person. Alle anderen Variablen der Unterabfrage beziehen sich auf die Relation *Personal* der Unterabfrage. Wir sehen ferner, dass hier ein Aliasname zwingend ist, da wir uns in der Unterabfrage sonst nicht auf das Gehalt der momentan betrachteten Person beziehen könnten. Es wird somit Zeile für Zeile die Existenz eines Tupels überprüft, dessen Personalnummer gleich 3 und dessen Gehalt größer als das aktuelle Gehalt dieser Zeile (*P1.Gehalt*) ist. Nur wenn dies der Fall ist, wird diese Zeile ausgegeben.

Beachten Sie noch, dass bisher die *Select*-Klauseln von Unterabfragen nur genau einen Spaltenausdruck enthielten. Dies ist in SQL-1 (bis auf eine Ausnahme) zwingend vorgeschrieben. Es soll ja auch nur jeweils ein Attributswert weiterverarbeitet werden. SQL-2 enthält Erweiterungen, die diesen Zwang aufheben. In Anhang B sind in der Syntax zum Bezeichner *Bedingung* einige dieser Erweiterungen angegeben. Eine Ausnahme von dieser Regel finden wir beim *Exists*-Operator, da hier die Existenz von Tupeln und nicht von Attributen interessiert.

Der Operator *Unique* wird wie der Operator *Exists* verwendet. Eine Bedingung mit *Unique* ist immer dann wahr, wenn die Unterabfrage nur eindeutige Tupel liefert. Dies ist beispielsweise immer der Fall, wenn die Unterabfrage nur ein oder kein Tupel zurückliefert oder mit *Select Distinct* beginnt.

4.1.4 Die Group-By- und Having-Klausel

Die *Group-By*-Klausel ermöglicht das Zusammenfassen von Tupeln mit gleichen Eigenschaften. Diese Klausel wirkt ähnlich wie der Bezeichner *Distinct* in der *Select*-Klausel. Wir können daher auch mit *Group By* die Wohnorte aller Mitarbeiter in der Relation *Personal* aus Tab. 18 auf Seite 79 auflisten, wobei jeder Wohnort nur einmal erscheint:

```
SELECT  Ort
 FROM    Personal
 GROUP  BY Ort ;
```

Die Klausel *Group By* gruppiert nach den angegebenen Attributen. Im Unterschied zur *Select*-Klausel darf in dieser Klausel nur eine Aufzählung von Attributsnamen (Spaltennamen) stehen, komplexe Spaltenausdrücke sind nicht erlaubt. Zu beachten ist ferner, dass in der *Select*-Klausel nur Attributsnamen vorkommen dürfen, nach denen gruppiert wird. Denn stünde etwa in der *Select*-Klausel des obigen Beispiels noch das Attribut *Name*, so wäre das Gruppieren nicht möglich, weil sowohl Frau Forster als auch Frau Kraus und Herr Noster in Regensburg wohnen, andererseits aber Regensburg nur einmal in der Ergebnisrelation vorkommen wird.

Für die oben vorgestellte Gruppierung hätte allein die *Distinct*-Angabe in der *Select*-Klausel genügt. Die Mächtigkeit der *Group-By*-Klausel ergibt sich erst im Zusammenwirken mit den Statistikfunktionen. Wir könnten uns beispielsweise nicht nur für die unterschiedlichen Wohnorte der Mitarbeiter interessieren, sondern auch dafür, wie viele Mitarbeiter in den jeweiligen Orten wohnen. Das Gewünschte liefert der folgende Befehl.

```
SELECT  Ort, COUNT (*) AS Anzahl
 FROM    Personal
 GROUP  BY Ort ;
```

Das dazugehörige Ergebnis ist in Tab. 25 wiedergegeben. Im Zusammenhang mit der *Group-By*-Klausel bekommen die Statistikfunktionen eine neue Bedeutung: Sie beziehen sich nicht mehr auf die gesamte Relation, sondern nur auf die einzelnen zu gruppierenden Teile. Daher können diese Statistikfunktionen jetzt auch zusammen mit anderen Attributen in der *Select*-Klausel vorkommen, was ohne *Group-By*-Klausel grundsätzlich verboten war. Als Parameter der Statistikfunktionen sind natürlich auch Attribute erlaubt, die nicht in der *Group-By*-Klausel aufgezählt sind; sie beeinflussen ja nicht die Gruppierung, sondern nur die Statistik zu den Gruppierungen.

Tab. 25 Ausgaberelation nach der Gruppierung

Ort	Anzahl
Regensburg	3
Straubing	1
Nürnberg	2
Landshut	1
Kelheim	1
Passau	1

Interessieren uns jetzt nur die Orte, in denen mindestens zwei Mitarbeiter wohnen, so können wir diese Restriktion nicht mittels der *Where*-Klausel ausführen, denn logisch gesehen, wird die *Where*- vor der *Group-By*-Klausel ab-

gearbeitet. Wir benötigen eine weitere Restriktion nach der Gruppierung. Diese zusätzliche Gruppierung geschieht mit Hilfe der *Having*-Klausel. Diese *Having*-Klausel ist nur in Zusammenhang mit der *Group-By*-Klausel erlaubt und dient ausschließlich der Restriktion nach einer Gruppierung. Wir wollen diese Anwendung an einem Beispiel zeigen:

```
SELECT  Ort, COUNT (*), SUM (12*Gehalt), 12 * MAX (Gehalt)
 FROM    Personal
 GROUP BY Ort
 HAVING COUNT (*) > 1 ;
```

Hier werden alle Orte ausgegeben, wo mindestens zwei Mitarbeiter wohnen. Zusätzlich erhalten wir das aufsummierte Jahresgehalt und das höchste Jahresgehalt aller Mitarbeiter, die hier leben. Wir werden also ausschließlich Statistikangaben zu den Mitarbeitern aus Regensburg und Nürnberg vorfinden. Die in der *Having*-Klausel verwendeten Statistikfunktionen beziehen sich immer auf die vorher durchgeführte Gruppierung.

4.1.5 Union, Except und Intersect

Bisher haben wir den Hauptteil des *Select*-Befehls behandelt. Jeder dieser Hauptteile liefert eine Ergebnisrelation. Diese Relationen können jetzt noch miteinander verknüpft werden. In SQL-1 gibt es nur die Vereinigung (*Union*), wobei diese Vereinigung nicht in Unterabfragen erlaubt ist. Diese Einschränkung wurde in SQL-2 aufgehoben. Hier wurde auch die Differenz (*Except*) und der Schnitt (*Intersect*) zweier Relationen in die Syntax aufgenommen. Wir kennen diese Operationen bereits aus der relationalen Algebra. Sie bieten vom Verständnis erfahrungsgemäß auch keine größeren Schwierigkeiten. Allerdings müssen in der Anwendung einige Punkte beachtet werden:

- Die Spalten der einzelnen Hauptteile müssen miteinander verträglich sein. Beispielsweise darf die zweite Spalte der ersten Ergebnisrelation nicht einen Zeichenkettenwert und die gleiche Spalte der zweiten Relation einen numerischen Wert liefern. Insbesondere muss die Spaltenanzahl beider Relationen übereinstimmen.
- Bei der Verknüpfung von mehr als zwei Hauptteilen bindet *Intersect* am stärksten. Im Zweifelsfalle sollten runde Klammern gesetzt werden, um die Reihenfolge der Verknüpfung eindeutig festzulegen.

Gerade die Vereinigung kann dazu führen, dass das Endergebnis mehrere gleiche Tupel enthält. Dies widerspricht der Vereinigung in der relationalen Algebra, wo es nur eindeutige Tupel gibt. Aus diesem Grund werden bei der Anwendung der drei Operationen *Union*, *Except* und *Intersect* automatisch Mehrfachtupel entfernt. Es gilt demnach quasi ein *Distinct* als Standardvorgabe für den gesamten *Select*-Befehl! Ist diese Zusammenfassung aus bestimmten Gründen nicht erwünscht, so kann hinter den Bezeichnern *Union*, *Except* oder *Intersect* noch der Bezeichner *All* angegeben werden. In diesem Fall entfällt das Zusammenfassen gleicher Tupel.

Oracle unterstützt alle drei Verknüpfungen, statt des Bezeichners *Except* ist lediglich der Oracle-spezifische Bezeichner *Minus* zu verwenden. MS-Access unterstützt ausschließlich die wichtige *Union*-Verknüpfung.

4.1.6 Die Verbindung (Join)

Bereits im Unterabschnitt 4.1.2 wurde die Join-Verknüpfung in der From-Klausel vorgestellt. Erst jetzt können wir auf den Join näher eingehen, da zur Erklärung der Verbindung zum Einen die *Where*-Klausel und zum Anderen die Vereinigung (*Union*) benötigt werden. Betrachten wir wieder unsere Relationen *Personal* und *Auftrag* aus Tab. 18 und Tab. 20. Einen natürlichen Join dieser beiden Relationen erhalten wir mit

```
SELECT  AuftrNr, Datum, Kundnr, Personal.*
 FROM    Personal, Auftrag
 WHERE Personal.Persnr = Auftrag.Persnr ;
```

Die beiden Relationen *Personal* und *Auftrag* wurden mittels der Personalnummern der beiden Relationen verknüpft. Wir erkennen, dass Attributsnamen auch in der *Where*-Klausel durch die Namen der Relationen qualifiziert werden können, hier wegen gleichlautender Attribute sogar qualifiziert werden mussten. Mit SQL-2 können wir diesen *Select*-Befehl viel kürzer formulieren:

```
SELECT  *
 FROM    Personal NATURAL INNER JOIN Auftrag ;
```

Der *Natural-Join*-Operator bildet aus allen gemeinsamen Attributen der beiden Relationen (gleiche Attributsnamen!) einen „natürlichen“ Join. In Datenbanken, etwa Oracle, wo der *Natural-Join*-Operator noch nicht realisiert ist, sei auf die vorhergehende Ersatzform verwiesen. Diese längere Darstel-

lung zeigt deutlich das Entstehen des natürlichen Joins aus dem Kreuzprodukt auf: Es erfolgt eine Restriktion auf die Tupel, dessen Attribute *Personal.Persnr* und *Auftrag.Persnr* übereinstimmen. Anschließend wird noch eine Projektion durchgeführt, indem eines dieser beiden identischen und damit überflüssigen Attribute entfernt wird.

Häufig kann der natürliche Join nicht angewendet werden, da die Attribute, die zur Verbindung beitragen, unterschiedliche Attributsnamen besitzen. In diesem Fall bietet SQL-2 eine *On*-Bedingung im Join an. Da MS-Access den Bezeichner *Natural* grundsätzlich nicht kennt, muss Access immer auf die *On*-Bedingung zurückgreifen. Der obige Befehl lautet dann:

```
SELECT *
 FROM   Personal INNER JOIN Auftrag ON Personal.Persnr = Auftrag.Persnr;
```

Laut SQL-2 Syntax darf der Bezeichner *Inner* hier und im vorherigen Beispiel weggelassen werden. Er gilt standardmäßig, falls keine andere *Join*-Spezifikation angegeben wird. MS-Access benötigt hingegen diesen Bezeichner. Dieser Bezeichner *Inner* weist darauf hin, dass nur Tupel ausgewählt werden, die in beiden zu verknüpfenden Relationen vorkommen.

Neben dem natürlichen (inneren) Join existieren weitere Joins, sogenannte *Theta-Joins*. Man spricht von diesen Joins, wenn statt des Gleichheitszeichens in der obigen *On*-Bedingung ein beliebiger Vergleichsoperator steht. Diese allgemeinen Joins kommen selten vor, können aber gelegentlich trotzdem recht nützlich sein.

Eine sehr große Bedeutung spielt noch der äußere Join. Wir wollen dies an einem Beispiel aufzeigen und betrachten dazu wieder unsere Relationen *Personal* und *Auftrag* aus Tab. 18 und Tab. 20 auf Seite 79. Die Tabelle *Auftrag* enthält Einträge zu jedem eingegangenen Auftrag. Hier sind unter anderem der Mitarbeiter, der den Auftrag betreute, und der Kunde, der etwas kaufte, aufgeführt. Den Personalchef interessiert nun, wie viele Verkäufe jeder einzelne Mitarbeiter bisher abwickelte. Folgender *Select*-Befehl erfüllt diesen Wunsch nur unvollständig:

```
SELECT  Personal.Persnr, Name, COUNT (*) As AnzahlAuftrag
 FROM    Personal NATURAL JOIN Auftrag
 GROUP BY Personal.Persnr, Name ;
```

Wurde die Schwäche erkannt? Zunächst sieht alles korrekt aus. Durch die Gruppierung und die Statistikfunktion wird jeder Mitarbeiter nur einmal auf-

geführt, zusammen mit der Anzahl des Auftretens in der Relation *Auftrag*. Vielleicht wurden aber vor kurzem neue Mitarbeiter eingestellt, die bisher noch nichts verkauft haben. Diese werden durch den natürlichen inneren Join unter den Tisch gekehrt. Es werden ja nur diejenigen Tupel verknüpft, deren Personalnummern in beiden Relationen vorkommen!

Sollen aber auch diejenigen Tupel aufgenommen werden, die nur in einer der beiden Relationen vorkommen, so sprechen wir vom äußeren Join, zum Unterschied zum bisherigen inneren Join. In SQL-2 existiert die Verknüpfung *Natural Full Outer Join*. Der korrekte (äußere) Join lautet daher in unserem Beispiel:

```
SELECT  Personal.Persnr, Name, COUNT (AuftrNr) AS AnzahlAuftrag
 FROM    Personal NATURAL FULL OUTER JOIN Auftrag
 GROUP BY Personal.Persnr, Name ;
```

Ein äußerer Join nimmt alle Tupel der beiden involvierten Relationen bei der Verknüpfung mit auf, auch diejenigen, deren Verknüpfungsattribut nur in einer der beiden Relationen vorkommt. In diesem Fall werden die Attribute der anderen Relation mit Nullwerten aufgefüllt. In unserem Beispiel werden demnach zunächst alle Mitarbeiter und Auftragsdaten verknüpft und aufgelistet. Zu allen Mitarbeitern, die noch nichts verkauften, wird im Attribut *AuftrNr* der Wert *Null* eingetragen. Anschließend erfolgt die Gruppierung. Wie bereits in Unterabschnitt 4.1.2 erwähnt wurde, zählt *Count (Attribute)* nur diejenigen Zeilen, die in den angegebenen Attributen keine Nullwerte enthalten. Folglich wird bei neuen Mitarbeitern völlig korrekt die Zahl 0 ermittelt.

Da die *Full-Outer-Join*- ebenso wie die *Inner-Join*-Verknüpfung noch nicht in allen SQL-Dialekten implementiert ist, so auch nicht in Oracle, wollen wir diesen Befehl auch ohne diesen Operator angeben. Hier können wir auch die exakte Arbeitsweise eines äußeren Joins ablesen:

```
SELECT  Personal.Persnr, Name, COUNT (*)
 FROM    Personal, Auftrag
 WHERE Personal.Persnr = Auftrag.Persnr
 GROUP BY Personal.Persnr, Name
UNION
SELECT  Persnr, Name, 0
 FROM    Personal
 WHERE Persnr NOT IN ( SELECT Persnr FROM Auftrag );
```

Wir haben zum innerer Join noch die restlichen Personaldaten hinzugefügt. Genaugenommen liegt ein Left-Outer-Join vor, da wir nur Daten einer der an

der Verknüpfung beteiligten Relationen hinzugefügt haben (das Wort *Left* bezieht sich auf die Relation links vom Join-Bezeichner). Und tatsächlich können wir in unserem Befehl die Wörter *Full Outer Join* durch den Ausdruck *Left Outer Join* ersetzen, ohne dass sich die Ausgabe dadurch verändert. Der Grund liegt in der Eigenschaft eines Fremdschlüssels verborgen. Schließlich bezieht sich jeder Fremdschlüsselwert (ungleich Null) auf einen existierenden Wert in der korrespondierenden Relation. Und damit sind alle Tupel der Relation, die den Fremdschlüssel enthält, automatisch im inneren Join enthalten. Ein äußerer Join muss daher nur noch die restlichen Tupel der anderen Relation aufnehmen. Dies ist vermutlich der Grund, warum MS-Access nur die Bezeichner *Left Outer Join* und *Right Outer Join*, nicht jedoch *Full Outer Join* unterstützt. Der Befehl müsste in MS-Access daher lauten:

```
SELECT  Personal.Persnr, Name, COUNT (AuftrNr) AS AnzahlAuftrag
 FROM    Personal LEFT OUTER JOIN Auftrag
    ON    Personal.Persnr = Auftrag.Persnr
 GROUP BY Personal.Persnr, Name ;
```

Dem aufmerksamen Leser ist der feine Unterschied zwischen *Count(*)* und *Count(AuftrNr)* nicht entgangen. In den beiden *Select*-Befehlen, die den Outer-Join-Bezeichner enthalten, würde die Angabe *Count(*)* zu einem falschen Ergebnis führen. Da dank des äußeren Joins jeder Mitarbeiter in der Ausgaberelation mindestens einmal vorkommt, liefert die Angabe *Count(*)* immer einen positiven Wert, auch dann wenn dieser Mitarbeiter noch nichts verkauft hat. Wir nutzen jetzt das automatische Auffüllen mit NULL-Werten aus. Wir erhalten die gewünschte Lösung, wenn wir in der Statistikfunktion ein Attribut der Relation *Auftrag*, angeben, am geeignetsten ist der Primärschlüssel.

Zuletzt sei noch erwähnt, dass in SQL-2 in der *From*-Klausel auch ein *Union Join* erlaubt ist (neben *Inner Join* und *Outer Join*). Allerdings liefert dieser *Union Join* ein anderes Ergebnis als der *Union*-Operator zwischen den Hauptteilen von *Select*-Befehlen. Es werden hier die Spalten beider verknüpfter Relationen ausgegeben, wobei nicht vorkommende Werte der anderen Relation jeweils durch Nullwerte ersetzt werden. Der folgende Befehl

```
SELECT  *
 FROM    Personal UNION JOIN Kunde ;
```

ist daher äquivalent zu

```
SELECT *, NULL, NULL, NULL, NULL, NULL FROM Personal
UNION ALL
SELECT NULL, NULL, NULL, NULL, NULL * FROM Kunde ;
```

Die Syntax von SQL-2 verbietet in einem *Union-Join* den Bezeichner *Natural* und die *On*-Bedingung. Zur Vertiefung der Arbeitsweise der *Join*-Verbindung im *Select*-Befehl wird empfohlen, diesen nochmals mit den entsprechenden Ausführungen zur relationalen Algebra am Ende des vorherigen Kapitels zu vergleichen.

4.1.7 Die Order-By-Klausel

Wir wissen bereits, dass in relationalen Datenbanken alle Relationen ungeordnet sind. Trotzdem kann es für den Anwender sehr wichtig sein, Daten geordnet auszugeben. Diese Möglichkeit wird in SQL mit der *Order-By*-Klausel unterstützt. Diese Klausel sortiert das bisherige Zwischenresultat des *Select*-Befehls entsprechend den angegebenen Vorgaben. Soll etwa zunächst nach der dritten Spalte und bei Gleichheit noch nach der zweiten Spalte geordnet werden, so werden diese Vorgaben, durch Komma getrennt, nacheinander angegeben. Als Bezeichner werden entweder die Spaltennamen oder die Spaltennummern verwendet. In der Relation *Personal* würden die beiden folgenden *Order-By*-Klauseln die gewünschte Sortierung erzwingen:

```
ORDER BY 3, 2
ORDER BY Ort, Name
```

In der *Order-By*-Klausel sind nur Namen oder Spaltennummern (beginnend bei 1!) erlaubt, jedoch keine Ausdrücke. Nach einem komplexen Spaltenausdruck kann daher nur sortiert werden, wenn entweder direkt die Spaltennummer oder im Spaltenausdruck der Spaltenliste noch ein Aliasname mit angegeben wurde.

SQL unterstützt aufsteigendes wie auch absteigendes Sortieren, wobei in Zeichenketten standardmäßig nach dem zugrundeliegenden Zeichencode (ASCII, EBCDIC) sortiert wird. Die Sortierreihenfolge wird durch die Angabe der Bezeichner *Asc* (ascending, aufsteigend) und *Desc* (descending, absteigend) festgelegt. Bei fehlender Angabe wird standardmäßig eine aufsteigende Sortierung angenommen. Wollen wir beispielsweise alle Wohnorte der Mitarbeiter sortiert ausgeben, wobei uns vor allem interessiert, in welchen Orten die meisten Mitarbeiter wohnen, so könnten wir schreiben:

```
SELECT  Ort, COUNT (*) AS Anzahl
 FROM    Personal
 GROUP BY Ort
 ORDER BY Anzahl DESC, Ort ;
```

Zunächst wird absteigend nach der Anzahl sortiert. Bei Gleichheit werden auch die Wohnorte alphabetisch ausgegeben. Die letzte Zeile hätten wir unter Weglassen des Aliasnamens *Anzahl* auch schreiben können als

```
ORDER BY 2 DESC, 1 ;
```

Dies trägt aber nicht unbedingt zur Lesbarkeit des Programmcodes bei.

Zum Abschluss dieses Abschnitts weisen wir nochmals darauf hin, dass wir den *Select*-Befehl nicht vollständig behandelt haben. Wir haben auf weitere Möglichkeiten von SQL-2 bewusst verzichtet, da dadurch keine grundsätzlich neuen Erkenntnisse vermittelt werden und der Sprachumfang des *Select*-Befehls noch überschaubar bleibt. Eine ausführlichere Behandlung findet sich in [DaDa98] und [MeSi93], aber auch in [MaFr93] und [Lans94]. Die ebenfalls nicht erwähnte *Into*-Klausel, die im eingebetteten SQL das Abspeichern von Ergebnissen in Variablen ermöglicht, werden wir in Kapitel 9 zum eingebetteten SQL kennenlernen.

4.2 Manipulationsbefehle in SQL

So kompliziert und umfangreich der *Select*-Befehl ist, so einfach und übersichtlich sind die weiteren Zugriffsbefehle. Wie bereits erwähnt ist der *Select*-Befehl der einzige Abfragebefehl. Die weiteren Zugriffsbefehle verändern die gespeicherten Daten. In Tab. 26 sind diese Manipulationsbefehle aufgeführt.

Tab. 26 Mutationsbefehle in SQL

UPDATE	Ändert bestehende Einträge
INSERT	Fügt neue Tupel (Zeilen) ein
DELETE	Löscht bestehende Tupel (Zeilen)

Beginnen wir mit dem *Update*-Befehl. Die Syntax dieses Befehls lautet:

```
UPDATE   Tabellenname
 SET     Spalte = Spaltenausdruck [ , ... ]
 [ WHERE Bedingung ]
```

Ein Update bezieht sich immer auf eine zu ändernde Relation, die nach dem Bezeichner *Update* anzugeben ist. Die zu ändernden Attribute werden, voneinander durch Kommata getrennt, nach dem Bezeichner *Set* zusammen mit den neuen Werten angegeben. Sollen nicht die Spalten aller Zeilen geändert werden, so kann mittels der *Where*-Klausel eine Restriktion durchgeführt werden. Die *Where*-Klausel entspricht eins zu eins der *Where*-Klausel des *Select*-Befehls: insbesondere sind Unterabfragen erlaubt.

Wir wollen die Funktionsweise dieses Befehls an einem Beispiel aufzeigen. Um etwa allen Mitarbeitern, die weniger als 5000 DM verdienen, eine Gehaltserhöhung von 5% zukommen zu lassen, genügt ein einziger *Update*-Befehl:

```
UPDATE Personal
 SET   Gehalt = 1.05 * Gehalt
 WHERE Gehalt < 5000 ;
```

Die Mächtigkeit dieses Befehls liegt also in den vielfältigen Möglichkeiten begründet, Tupel mittels der *Where*-Klausel auswählen zu können. Ferner kann im *Update*-Befehl einem Attribut nicht nur ein neuer Werte zugewiesen werden, es dürfen vielmehr auch existierende Werte auf *Null* gesetzt werden. Hierzu wird der reservierte gleichlautende Bezeichner *Null* auf der rechten Seite einer Zuweisung verwendet.

Auch der *Delete*-Befehl ist einfach handhabbar und bedarf nur einer kurzen Erläuterung. Die Syntax lautet:

```
DELETE FROM  Tabellenname
 [ WHERE   Bedingung ]
```

Wieder entspricht die *Where*-Klausel der gleichnamigen Klausel im *Select*-Befehl, wieder sind Unterabfragen zugelassen. Der *Delete*-Befehl löscht alle Tupel der angegebenen Relation, die die Bedingung in der *Where*-Klausel erfüllen. Scheidet etwa Frau Rank aus der Firma aus, so wird der entsprechende Eintrag in der Personaltabelle durch den folgenden Befehl entfernt:

```
DELETE FROM  Personal
 WHERE Name = 'Ursula Rank' ;
```

Zuletzt betrachten wir noch den Einfügebefehl *Insert*. Damit können wir in eine bestehende Relation einen oder mehrere neue Tupel einfügen. Obwohl die Syntax in SQL-2 erweitert wurde, dürften wir in den meisten Anwendungsfällen mit der folgenden aus SQL-1 abgeleiteten Syntax auskommen:

```
INSERT INTO  Tabellenname  [  (  Spaltenliste  )  ]
  { VALUES  (  Auswahlliste  )
  |  Select-Befehl  }
```

Dieser Befehl fügt entweder eine neue Zeile mit den in der Auswahlliste angegebenen Werten in die Relation ein, oder es werden die durch einen beliebigen *Select*-Befehl (ohne *Order-By*-Klausel) ermittelten Zeilen in die Relation aufgenommen. Die Anzahl der Attribute in der Auswahlliste bzw. im *Select*-Befehl muss mit der Größe der Spaltenliste übereinstimmen. Das Gleiche gilt für die Datentypen der einzelnen Attribute. Die Spaltenliste darf weggelassen werden. In diesem Fall sind automatisch alle Spalten in der durch die Definition der Relation festgelegten Reihenfolge vorgegeben. Wir können also einen neuen Mitarbeiter wie folgt in unsere Personaltabelle (siehe Tab. 18) aufnehmen:

```
INSERT INTO  Personal (Persnr, Name, Ort)
 VALUES    (10, 'Lars Anger', 'Sinzing') ;
```

Die Reihenfolge der Angaben in der Spaltenliste und der Auswahlliste muss aufeinander abgestimmt sein. Die weiteren existierenden Attribute *Vorgesetzt* und *Gehalt* werden durch Nullwerte vorbesetzt und können durch einen späteren *Update*-Befehl jederzeit ergänzt werden.

Sind in einer Relation Werte aufzunehmen, die bereits in einer anderen Relation existieren, so empfiehlt sich die Variante mit der *Select*-Abfrage. Soll etwa der neue Mitarbeiter *Lars Anger* das gleiche Gehalt und den gleichen Vorgesetzten wie Herr Noster (Personalnummer 7) erhalten, so können diese Daten direkt aus der Relation übernommen werden:

```
INSERT INTO  Personal
 SELECT 10, 'Lars Anger', 'Passau', Vorgesetzt, Gehalt
 FROM    Personal
 WHERE Persnr = 7 ;
```

Wir erkennen, dass wir bestehende Angaben aus anderen Relationen übernehmen und fehlende entsprechend den Möglichkeiten der *Select*-Klausel di-

rekt als Konstante einsetzen können. Mit Hilfe der *Select*-Abfrage im *Insert*-Befehl können natürlich auch mehrere Tupel auf einmal eingefügt werden.

In SQL-2 ist es möglich, den *Select*- bzw. *Values*-Teil zu erweitern. Im Prinzip darf hier jetzt ein beliebiger Tabellenausdruck stehen. Zum Beispiel ist ein Join zwischen einem *Select*-Befehl und einer *Values*-Angabe erlaubt. Wir verweisen auf die Syntax in Anhang B und auf [DaDa98] und [MeSi93].

4.3 Relationale Algebra und SQL

Wir wollen noch zeigen, dass SQL mindestens die Mächtigkeit der Operatoren der relationalen Algebra besitzt. Dazu müssen wir zu allen fünf relevanten Operatoren entsprechende SQL-Befehle angeben. Die fünf relevanten Operatoren waren die Vereinigung, das Kreuzprodukt, die Restriktion, die Projektion und die Differenz (siehe Abschnitt 3.4). Als kleine Ergänzung zu diesem Kapitel haben wir daher in Tab. 27 diese Operatoren den jeweiligen *Select*-Befehlen gegenübergestellt. Dabei bedeuten *A* und *B* Relationen, die x_i stehen für Attribute und *p* für eine beliebige Bedingung.

Tab. 27 Vergleich: Relationale Algebra - SQL

Operator	Algebra	SQL-1	SQL-2
Vereinigung	A UNION B	SELECT * FROM A UNION SELECT * FROM B	SELECT * FROM A UNION SELECT * FROM B
Kreuzprodukt	A TIMES B	SELECT * FROM A,B	SELECT * FROM A,B
Restriktion	A WHERE p	SELECT * FROM A WHERE p	SELECT * FROM A WHERE p
Projektion	A [x_1 , ... , x_n]	SELECT x1, ... , xn FROM A	SELECT x1, ... , xn FROM A
Differenz	A MINUS B	SELECT * FROM A WHERE NOT EXISTS (SELECT * FROM B WHERE "alle Felder von A = alle Felder von B")	SELECT * FROM A EXCEPT SELECT * FROM B

Die Nachbildung der Differenz bereitet in SQL-1 einige Schwierigkeiten. Hier müssen in einer Unterabfrage alle Attribute der einen Relation mit den Attributen der anderen auf Gleichheit überprüft werden. In SQL-2 ist dieser

Differenz-Operator dank der *Except*-Verknüpfung wesentlich einfacher nachbildbar. Die anderen vier Operatoren können mit der Sprache SQL sehr einfach formuliert werden. Als Übung sei empfohlen, auch die drei abgeleiteten relationalen Operatoren *Schnitt*, *Join* und *Division* mit Hilfe von SQL-Befehlen nachzuvollziehen.

Wir erkennen, dass es in SQL grundsätzlich möglich ist, die Grundoperationen der relationalen Algebra nachzubilden. Dies bedeutet, dass SQL eine umfassende Zugriffssprache ist, mit der alle relevanten Daten einer relationalen Datenbank abgefragt werden können.

4.4 Zusammenfassung

Wir haben in diesem Kapitel die vier DML-Befehle von SQL kennengelernt: *Select*, *Update*, *Insert* und *Delete*. Jedes Anwendungsprogramm, das auf Datenbanken zugreift, wird mit diesen vier DML-Befehlen und ihren geringfügigen Erweiterungen zum eingebetteten SQL (siehe Kapitel 9) auskommen. Jeder, der eine Programmiersprache und diese vier Befehle beherrscht, kann auf eine bestehende Datenbank zugreifen.

Während die Manipulationsbefehle *Update*, *Insert* und *Delete* sowohl in ihrem syntaktischen Aufbau als auch in ihrer Handhabung relativ leicht verständlich sind, ist der Abfragebefehl *Select* sehr komplex. Zur Beherrschung dieses Befehls müssen wir mit folgenden Punkten vertraut sein:

- Abarbeitungsreihenfolge des *Select*-Befehls: Aus logischer Sicht ist die Abarbeitungsreihenfolge fest vorgegeben. Diese ist in Tab. 22 angegeben und trägt wesentlich zum Verständnis des *Select*-Befehls bei.
- Join: Fast jede zweite Abfrage verwendet einen inneren Join, viele auch einen äußeren. Es ist die natürlichste Art und Weise, zwei oder mehrere Relationen miteinander zu verknüpfen. Der natürliche Join beinhaltet ein Kreuzprodukt, wobei nur die Tupel mit aufgenommen werden, die in den verbindenden Attributen übereinstimmen. Die verbindenden Attribute werden nur einmal ausgegeben.
- Unterabfragen: Mit Hilfe von Unterabfragen werden Tupel ausgewählt, die bestimmte Eigenschaften erfüllen. Wieder ist die logische Abarbeitungsreihenfolge des *Select*-Befehls zu beachten: Es wird Zeile für Zeile geprüft, und für jede dieser Zeilen wird die Unterabfrage einzeln durchgeführt. Zu beachten ist ferner die Gültigkeit von Variablen.

- Gruppierung und Statistikfunktionen: Die Gruppierung fasst Gruppen mit gleichen Eigenschaften zusammen. Mit Hilfe der Statistikfunktionen können zu jeder dieser Gruppen noch zusammenfassende Statistiken wie die Anzahl, der Mittelwert, die Summe, das Maximum oder das Minimum der Gruppe ermittelt werden.

Unter Beachtung dieser wenigen Stichpunkte sollte der *Select*-Befehl keine weiteren Schwierigkeiten bereiten.

4.5 Übungsaufgaben

Die folgenden Aufgaben beziehen sich ausnahmslos auf die Relationen der Beispieldatenbank *Radl* in Anhang A.

1) Schreiben Sie einen *Select*-Befehl, der aus der Relation *Personal* die Namen aller Personen ermittelt, die mehr als 4000 DM verdienen.
2) Geben Sie mittels eines *Select*-Befehls die Gesamtanzahl der für Aufträge reservierten Teile aus (die benötigten Informationen stehen in der Relation *Teilereservierung*).
3) Geben Sie mittels eines *Select*-Befehls alle Teile der Relation *Lager* aus, deren Bestand abzüglich des Mindestbestands und der Reservierungen unter den Wert 3 gesunken ist. Als Ausgabe werden Teilenummer und Teilebezeichnung erwartet.
4) Aus wievielen Einzelteilen bestehen die zusammengesetzten Teile? Bestimmen Sie diese Stückzahlen mittels eines *Select*-Befehls. Falls ein Einzelteil wieder aus noch kleineren Einzelteilen besteht, so ist dies nicht weiter zu berücksichtigen. Ausschlaggebend zur Ermittlung der Anzahl der Einzelteile ist das Attribut *Anzahl* ohne Rücksichtnahme auf die Einheit (‚ST' oder ‚CM').
5) Geben Sie alle Teile aus, die vom Auftrag mit der Auftragsnummer 2 reserviert sind. Geben Sie dazu zu jedem Teil die Teilenummer, die Teilebezeichnung und die Anzahl der für diesen Auftrag reservierten Teile aus.
6) Modifizieren Sie die Aufgabe 5 dahingehend, dass alle Teile der Relation *Teilestamm* ausgegeben werden (Outer Join), und nicht nur die reservierten. Für die nicht für Auftrag 2 reservierten Teile ist die entsprechende Spaltenangabe zu den Reservierungen auf den Wert Null zu setzen. Schreiben Sie den *Select*-Befehl einmal ohne (SQL-1) und einmal mit Verwendung des Operators *Full Outer Join* (SQL-2).
7) Es sind 7 neue Sättel eingetroffen. Modifizieren Sie die Datenbank.

8) Es wird ein Damen-Mountainbike ins Sortiment aufgenommen. Weitere Angaben sind: 100003, 1300.00, 26 Zoll, ST, E, NULL, NULL, 6. Nehmen Sie dieses Fahrrad in die Relation *Teilestamm* auf.
9) Die Lieferantin Gerda Schmidt wird auch Kundin. Nehmen Sie die Lieferantin auch in die Kundenrelation auf. Die benötigten Daten sind direkt der Lieferantenrelation zu entnehmen.
10) Löschen Sie alle Teile aus der Relation *Lager*, deren Bestand auf Null gesunken ist.
11) Erhöhen Sie das Gehalt aller Mitarbeiter um 100 DM, bei denen die Beurteilung 1 eingetragen ist. Senken Sie gleichzeitig die Beurteilung um eine Note.
12) Bilden Sie die Operatoren *Schnitt*, *Join* und *Division* der relationalen Algebra mit Hilfe von SQL-1 und SQL-2 Befehlen nach.

5 Datenbankdesign

In Kapitel 3 haben wir Regeln zur Erstellung von Relationen kennengelernt und gesehen, dass Fremdschlüssel die Verbindung zwischen Relationen herstellen. Weiter haben wir in Kapitel 4 gelernt, auf existierende Datenbanken zuzugreifen und deren Inhalt zu verändern. In diesem Kapitel beschäftigen wir uns nun mit dem Entwurf einer Datenbank selbst. Das Relationenmodell lässt uns dazu sehr viele Freiheiten. Bis auf einige Einschränkungen zum Primär- und Fremdschlüssel hat der Datenbankdesigner ein reiches Betätigungsfeld.

Dieses Kapitel gibt wertvolle und wichtige Anregungen und Hilfestellungen zum Datenbankentwurf. Wir geben den dringenden Rat, sich als Datenbankdesigner auch an diese Anregungen zu halten. Nur so kann späterer Ärger wegen langer Zugriffszeiten, komplexer Abfragen und Redundanzen mit der Gefahr von Inkonsistenzen vermieden werden. Halten muss sich der Designer an diese Anregungen und Regeln im Gegensatz zu den Regeln in Kapitel 3 aber nicht. Doch bereits das Beispiel der Relation *VerkaeuferProdukt* in Tab. 13 auf Seite 65 zeigte eindrucksvoll die Schwächen eines schlechten Designs einer Relation auf, obwohl das Beispiel im Sinne der Definition einer Relation korrekt war. Noch schlimmer wirkt sich ein schlechtes Design einer ganzen Datenbank aus. Es gilt: einmal schlechtes Design - immer schlechte Anwendung!

Aufbauend auf Kapitel 3 werden wir in diesem Kapitel Normalformen von Relationen und die dazu benötigten Definitionen einführen. Während die Normalformenlehre immer nur einzelne Relationen für sich betrachtet, leistet das anschließend vorgestellte Entity-Relationship-Modell eine wertvolle Hilfestellung zum Gesamtdesign einer Datenbank mit all den Beziehungen zwischen den Relationen. Seine Anwendbarkeit ist universell und beschränkt sich nicht allein auf relationale Datenbanken.

Dieses und das dritte Kapitel sind die zentralen Kapitel zu relationalen Datenbanken. Jeder Datenbankprogrammierer sollte mit den Integritätsregeln aus Kapitel 3 und den hier behandelten Normalformen und dem Entity-Relationship-Modell vertraut sein. Einem optimalen Design und einer optimalen Zugriffsprogrammierung steht dann nichts mehr im Wege.

5.1 Normalformen

Die Normalformenlehre beschreibt, wie Relationen aufgebaut werden sollten, um Redundanzen und Zugriffsprobleme zu vermeiden. Bereits bei seiner Vorstellung der relationalen Datenbanken im Jahre 1970 führte E. F. Codd drei Normalformen ein. Während die ersten beiden Normalformen bis heute unverändert blieben, gibt es inzwischen zur dritten Normalform verschiedene Varianten. Darüberhinaus kamen im Laufe der Zeit noch eine vierte und fünfte Normalform hinzu. Diese Normalformen bauen direkt aufeinander auf, wobei die erste Normalform am wenigsten, die fünfte am stärksten einschränkt. Es gilt ganz allgemein:

- Befindet sich eine Relation in der n. Normalform, so befindet sie sich auch in der m. Normalform mit $m \leq n$.

Beginnen wir gleich mit der ersten Normalform:

Definition (erste Normalform)

☞Eine Relation ist in der ersten Normalform, wenn alle zugrundeliegenden Gebiete nur atomare Werte enthalten.

Diese Definition besagt, dass alle Attribute nur einfache Attributswerte enthalten. Dies haben wir aber bereits bei der Definiton einer (normalisierten) Relation gefordert. Eine Relation, wie wir sie definiert haben, ist demnach automatisch in der ersten Normalform.

Die Definition der ersten Normalform ist historisch bedingt, da in der ursprünglichen Definition einer Relation von E. F. Codd noch nicht gefordert war, dass eine Relation nur atomare Attribute enthalten darf.

Die Relation *VerkaeuferProdukt* aus Tab. 13 ist eine normalisierte Relation, und damit in der ersten Normalform. Diese Relation besitzt allerdings zwei gravierende, bereits in Kapitel 3 behandelte Schwächen. Zum Einen enthält sie Redundanzen, da zu jedem verkauften Produkt immer alle Verkäuferdaten aufgelistet sind. Zum Zweiten ist das Entfernen des Produkts *Staubsauger* aus dem Sortiment nicht möglich, da wir sonst auch alle Informationen zum Verkäufer *Müller* verlieren würden. Ein Ersetzen des Inhalts *Staubsauger* durch den *Null*-Wert ist ebenfalls nicht erlaubt, da das Attribut *Produktname* ein Teil des Primärschlüssels ist, und dies somit die erste Integritätsregel verletzen würde!

Die erste Normalform ist daher mit Sicherheit nicht ausreichend, um die angesprochenen Probleme zu beseitigen. Wir müssen weitere Normalformen einführen. Dazu benötigen wir zunächst einige Begriffe.

5.1.1 Funktionale Abhängigkeit

Die höheren Normalformen fordern bestimmte Zusammenhänge zwischen den einzelnen Attributen, mit denen wir uns eingehend beschäftigen müssen. Dabei ist folgende Definition sehr wichtig, die die Abhängigkeiten der einzelnen Attribute untereinander beschreibt.

Definition (funktionale Abhängigkeit)
☞Ein Attribut *Y* einer Relation *R* heißt funktional abhängig vom Attribut *X* derselben Relation, wenn zu jedem *X*-Wert höchstens ein *Y*-Wert möglich ist.

Die Definition beschränkt sich nicht auf einzelne Attribute, selbstverständlich können *X* und *Y* zusammengesetzte Attribute sein. Ist *Y* von *X* funktional abhängig, so schreiben wir:

$$X \rightarrow Y \quad .$$

Die Definition bedeutet: *Y* ist funktional abhängig von *X*, wenn während der gesamten Existenz der Relation zu jedem *X*-Wert höchstens ein *Y*-Wert existieren kann. Das Wort „höchstens“ kann wegen möglicher *Null*-Werte nicht durch „genau“ ersetzt werden. Wir wollen diese Definition am Beispiel der Relation *VerkaeuferProdukt* aus Tab. 13 auf Seite 65 nachvollziehen. Dort hängen die Verkäuferdaten von der Verkäufernummer ab. Es liegt eine funktionale Abhängigkeit vor. Wir können schreiben:

$$\begin{aligned} &\text{Verk\_Nr} \rightarrow \text{Verk\_Name} \\ &\text{Verk\_Nr} \rightarrow \text{PLZ} \\ &\text{Verk\_Nr} \rightarrow \text{Verk\_Adresse} \end{aligned}$$

oder kurz zusammengefasst:

$$\text{Verk\_Nr} \rightarrow (\text{Verk\_Name}, \text{PLZ}, \text{Verk\_Adresse})$$

Suchen wir nach weiteren funktionalen Abhängigkeiten in der Relation. Weder die Produkte noch die Umsätze sind eindeutig einem Verkäufer oder einer Verkäufernummer zuzuordnen, auch die Postleitzahl und die Verkäuferadresse sind nicht vom Verkäufernamen funktional abhängig. Es könnten ja im Laufe der Zeit noch weitere Verkäufer mit dem Namen *Meyer* oder *Müller* eingestellt werden. Aus dem Verkäufernamen kann daher nicht immer eindeutig auf die Adresse geschlossen werden. Interessant wird es, wenn wir die Postleitzahlen und die Wohnorte miteinander vergleichen. Im Allgemeinen existieren auch hier keine Abhängigkeiten, da es Kleingemeinden mit gleicher Postleitzahl gibt. Umgekehrt besitzen alle Großstädte mehr als eine Postleitzahl. Kommen in einer Datenbank aber nur größere Ortschaften vor (aus welchen Gründen auch immer), so wäre folgende Abhängigkeit denkbar:

PLZ → Verk_Adresse (falls die Adresse nur große Ortschaften enthält!)

Vergleichen wir die Definition des Primärschlüssels mit der Definition der funktionalen Abhängigkeit, so erkennen wir, dass in einer Relation alle Attribute vom Primärschlüssel funktional abhängen. Schließlich gibt es zu jedem Primärschlüsselwert wegen der eindeutigen Identifikation nur höchstens einen Eintrag in jedem Attribut! Um dies in unserem Beispiel der Relation *VerkaeuferProdukt* nachzuvollziehen, identifizieren wir den Primärschlüssel dieser Relation: das zusammengesetzte Attribut *Verk_Nr* + *Produktname*. Zu jedem Primärschlüsselwert existiert genau ein Umsatzwert und ein Verkäufer. Somit gilt:

(Verk_Nr, Produktname) → Umsatz
(Verk_Nr, Produktname) → (Verk_Name, PLZ, Verk_Adresse)

Es ist zu beachten, dass die Verkäuferdaten, wie schon gezeigt, bereits allein von der Verkäufernummer funktional abhängig waren. Ganz allgemein gilt, dass bei einer vorliegenden funktionalen Abhängigkeit $X \rightarrow Y$ leicht weitere konstruiert werden können, indem das Attribut X um zusätzliche Attribute erweitert wird. Um deshalb einer Flut von funktionalen Abhängigkeiten vorzubeugen, erweitern wir unsere Definition:

Definition (volle funktionale Abhängigkeit)

☞ Ein Attribut Y einer Relation R heißt voll funktional abhängig vom Attribut X derselben Relation, wenn es funktional abhängig von X ist, nicht aber funktional abhängig von beliebigen Teilattributen von X.

Ist Y voll funktional abhängig von X, so schreiben wir abkürzend $X \Rightarrow Y$. Wo zur funktionalen Abhängigkeit keine Verwechslung auftreten kann, verwenden wir weiterhin die Darstellung $X \rightarrow Y$.

Es wird sofort klar, dass bei nicht zusammengesetzten Attributen X aus der funktionalen Abhängigkeit von X automatisch die volle funktionale Abhängigkeit geschlossen werden kann. Interessant wird diese Definition bei Primärschlüsseln, die aus zusammengesetzten Attributen bestehen. Wir kommen daher auf unser Beispiel *VerkaeuferProdukt* zurück. Nach den obigen Überlegungen hängen dort die Verkäuferdaten (*Verk_Name*, *Verk_Adresse*, *PLZ*) vom Primärschlüssel funktional ab, nicht aber voll funktional. Diese Daten hingen funktional ja allein schon von der Verkäufernummer ab. Wenn wir unser Beispiel genauer betrachten, ist es genau dieses Verhalten, was unsere Relation so schwerfällig macht.

Fassen wir unsere neuen Erkenntnisse zusammen: Es gibt ganz allgemein immer Pfeile ($\rightarrow$), die vom Primärschlüssel (präziser: von den Schlüsselkandidaten) ausgehen. Dies liegt an der Definition des Primärschlüssels, und ist eine grundlegende Eigenschaft von Relationen. Probleme treten allerdings dann auf, wenn diese Pfeile keine Doppelpfeile ($\Rightarrow$) sind, wie etwa in unserer Beispielrelation *VerkaeuferProdukt*.

Bevor wir mit dem nächsten Abschnitt fortfahren, sei als Übung empfohlen, in der Beispielrelation *VerkaeuferProdukt* alle vollen funktionalen Abhängigkeiten zu ermitteln.

5.1.2 Zweite und dritte Normalform

Wir haben mit der Definition der (vollen) funktionalen Abhängigkeit bereits gesehen, dass Abhängigkeiten vom Primärschlüssel immer existieren. Diese sind auch erwünscht, nicht jedoch wenn keine vollen funktionalen Abhängigkeiten vorliegen. Wir werden aus diesem Grund nur noch Abhängigkeiten vom Primärschlüssel erlauben, die voll funktional sind:

Definition (zweite Normalform)

☞ Eine Relation ist in der zweiten Normalform, wenn sie in der ersten Normalform ist, und jedes Nichtschlüsselattribut voll funktional abhängig vom Primärschlüssel ist.

Ein Nichtschlüsselattribut ist jedes (auch zusammengesetzte) Attribut, das kein Schlüsselkandidat ist. Die Definition schließt also nicht aus, dass neben dem Primärschlüssel noch alternative Schlüssel vorkommen dürfen. Die Relation der chemischen Elemente (Tab. 17 auf Seite 76) ist beispielsweise in der zweiten Normalform. Wir wissen, dass zwischen Schlüsselkandidaten per definitionem eine eineindeutige Beziehung existiert; sie sind demnach wechselseitig voll funktional voneinander abhängig.

Die zweite Normalform bezieht sich nur auf den Zusammenhang zwischen allen Nichtschlüsselattributen und dem Primärschlüssel. Für jede Relation in zweiter Normalform gilt daher:

$$\text{Primärschlüssel} \Rightarrow \text{Nichtschlüsselattribut}\ .$$

Da für normalisierte Relationen bereits allgemein

$$\text{Primärschlüssel} \rightarrow \text{Nichtschlüsselattribut}$$

galt, sieht die zweite Normalform nur nach einer geringen Erweiterung aus. Konsequent angewendet reduziert sie jedoch deutlich Redundanzen. Wie wir bereits gesehen haben, sind in unserer Relation *VerkaeuferProdukt* die Verkäuferdaten (*Verk_Name, PLZ, Verk_Adresse*) nicht voll funktional abhängig vom Primärschlüssel. Die Relation ist daher nicht in der zweiten Normalform.

Es lässt sich sehr leicht beweisen, dass jede Relation (in der ersten Normalform) in eine oder mehrere Relationen der zweiten Normalform überführt werden kann. Einen exakten Beweis wollen wir hier nicht führen, doch können wir dies an unserem Beispiel der Relation *VerkaeuferProdukt* recht eindrucksvoll aufzeigen.

Wir nummerieren einfach die Einträge in der ursprünglichen Tabelle *VerkaeuferProdukt* durch, und fügen diese Zahlen einem neuen Attribut *Nr* hinzu. Diese Relation besitzt nun zwei Schlüsselkandidaten, nämlich *Nr* und *(Verk_Nr, Produktname)*. Legen wir den letztgenannten Schlüssel als Primärschlüssel fest, so haben wir nichts gewonnen. Wählen wir jedoch das Attribut *Nr*, so ist die Relation in der gewünschten zweiten Normalform! Von dem nicht zusammengesetzten Attribut *Nr* hängen nämlich alle anderen Nichtschlüsselattribute voll funktional ab. Als Ergebnis erhalten wir die Relation *VerkaeuferProdukt_2NF* aus Tab. 28.

Tab. 28 VerkaeuferProdukt_2NF

Nr	Verk_Nr	Verk_Name	PLZ	Verk_Adresse	Produktname	Umsatz
1	V1	Meier	80075	München	Waschmaschine	11000
2	V1	Meier	80075	München	Herd	5000
3	V1	Meier	80075	München	Kühlschrank	1000
4	V2	Schneider	70038	Stuttgart	Herd	4000
5	V2	Schneider	70038	Stuttgart	Kühlschrank	3000
6	V3	Müller	50083	Köln	Staubsauger	1000

Wir erkennen, dass wir mit dieser neuen Relation das Problem der Redundanz der ursprünglichen Relation nicht lösen. Da nun das Attribut *Produktname* aber nicht mehr Teil des Primärschlüssels ist, kann jetzt andererseits das Produkt *Staubsauger* jederzeit entfernt werden, indem die entsprechenden Einträge einfach mit *Null* überschrieben werden. Wir sehen, dass die immer mögliche Überführung einer Relation in die zweite Normalform mittels des Hinzufügens eines einzelnen Primärschlüsselattributs zumindest teilweise befriedigt. Eine optimale Transformation unserer Relation *VerkaeuferProdukt* liegt aber mit Sicherheit nicht vor!

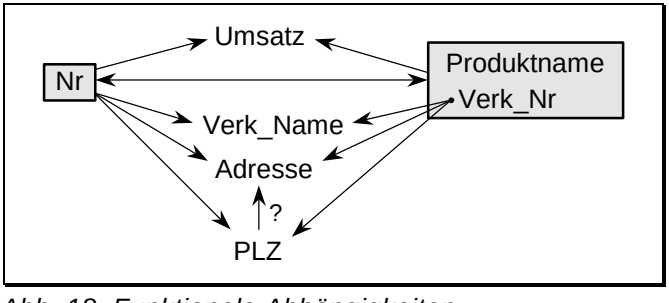


Abb. 18 Funktionale Abhängigkeiten

Abb. 18 zeigt (fast) alle vollen funktionalen Abhängigkeiten der einzelnen Attribute der Relation *VerkaeuferProdukt_2NF* untereinander und zu den umrahmten Schlüsselkandidaten auf (natürlich hängen auch die Einzelattribute des alternativen Schlüssels vom Primärschlüssel ab). Das Hauptaugenmerk sei wiederum auf das Attribut *Verk_Nr* gelenkt, von dem die Verkäuferdaten abhängen, und das gleichzeitig ein Teil eines Schlüsselkandidaten ist. Der Verkäufername ist nicht eindeutig, so dass keine funktionalen Abhängigkeiten zur Adresse und Postleitzahl des Verkäufers existieren.

Eine funktionale Abhängigkeit der Adresse (Wohnort) von der Postleitzahl liegt dann vor, wenn keine zwei Orte eine gemeinsame Postleitzahl besitzen.

Dies trifft nur dann zu, wenn wir uns auf größere Gemeinden beschränken. In der Abbildung ist der entsprechende Pfeil daher mit einem Fragezeichen versehen. In der Praxis enthält die Adresse neben dem Wohnort auch die Straße und die Hausnummer. Wir vermuten jetzt eine Abhängigkeit der Postleitzahl von der kompletten Adresse. Diese Vermutung lässt sich aber leicht am Beispiel des Ortes *Neustadt* widerlegen. Dies ist ein häufig vorkommender Ortsname, und sicherlich gibt es auch Straßennamen wie Marktplatz oder Hauptstraße in mehreren dieser Orte. Jeder Ort hat aber eine eigene Postleitzahl, so dass aus der Adresse nicht immer eindeutig die Postleitzahl folgt.

Wir können die Relation *VerkaeuferProdukt_2NF* noch durch ein Attribut *Bundesland* erweitern. Dadurch entstehen zusätzliche funktionale Abhängigkeiten des Bundeslandes von den Schlüsselkandidaten und von der Postleitzahl. Die Relation selbst wäre immer noch in der zweiten Normalform. Wir erkennen, dass die zweite Normalform zwar das Arbeiten mit Relationen erleichtert, nicht jedoch notwendigerweise Redundanzen entfernt. Die funktionale Abhängigkeit (ein Pfeil) beschreibt Abhängigkeiten. Besitzt nun ein Attribut, von dem ein Pfeil ausgeht, mehrere gleiche Einträge, so wird leider auch diese Abhängigkeit mehrfach gespeichert!

Sicherlich hat dieses Beispiel seine abschreckende Wirkung nicht verfehlt. Es zeigt uns, dass die zweite Normalform noch nicht die optimale Form unserer Relationen sein kann. Wir müssen weitere, einschränkendere Normalformen definieren. Zuvor geben wir jedoch noch einige Anmerkungen zur zweiten Normalform:

Wichtig

☞Eine normalisierte Relation mit einem nicht zusammengesetzten Primärschlüssel befindet sich immer in der zweiten Normalform. Umgekehrt besitzt eine Relation, die sich nicht in der zweiten Normalform befindet, einen zusammengesetzten Primärschlüssel.

Der Beweis dieser Aussage ist sehr einfach und sei als Übung empfohlen.

Die Definition der zweiten Normalform besagt, dass vom Primärschlüssel Doppelpfeile ausgehen. Weitere Pfeile spielen keine Rolle. Doch gerade diese sorgen für Redundanzen, denken wir nur an die Verkäufernummer in der Relation *VerkaeuferProdukt_2NF*. Die logische Folgerung ist nun, all diese weiteren Pfeile zu verbieten. Damit gelangen wir zur dritten Normalform. Wir verwenden hier nicht die ursprüngliche Definition von E. F. Codd, sondern gehen auf die erst später eingeführte, dafür aber anschaulichere Definition von

Boyce und Codd ein. Die Originaldefinition wird im folgenden Unterabschnitt vorgestellt.

Definition (Determinante)
☞Eine Determinante ist ein (eventuell zusammengesetztes) Attribut, von dem ein anderes voll funktional abhängig ist.

Diese Definition besagt nur, dass jedes Attribut, von dem ein Doppelpfeil ausgeht, als Determinante bezeichnet wird. In der zweiten Normalform sind also mindestens alle Schlüsselkandidaten Determinanten. Die dritte Normalform geht noch einen Schritt weiter:

Definition (dritte Normalform nach Boyce/Codd)
☞Eine normalisierte Relation ist in der dritten Normalform, wenn jede Determinante dieser Relation ein Schlüsselkandidat ist.

Diese Definition besagt also, dass es außer Schlüsselkandidaten keine Determinanten gibt. In einer etwas anschaulicheren Sprechweise bedeutet dies, dass alle existierenden Doppelpfeile von den Schlüsselkandidaten ausgehen!

Diese Definition greift nicht auf die Definition der zweiten Normalform zurück. Trotzdem gilt, dass sich jede Relation in der dritten Normalform auch in der zweiten Normalform befindet. Dies ist leicht einsichtig. Während die zweite Normalform nur verlangte, dass vom Primärschlüssel Doppelpfeile zu jedem Attribut existieren müssen, verlangt die dritte Normalform, dass dies auch die einzigen (Doppel-)Pfeile sind.

Es ist damit klar, dass sich unsere Relation *VerkaeuferProdukt* nicht in der dritten Normalform befinden kann. Sie ist ja nicht einmal in der zweiten. Aber auch unsere künstlich erzeugte Relation *VerkaeuferProdukt_2NF* erfüllt die Bedingungen der dritten Normalform nicht. Aus Abb. 18 können wir leicht die Determinanten ablesen. Dies sind

Nr	Verk_Nr
(Verk_Nr, Produktname)	PLZ (*eventuell*).

Auf der linken Seite stehen gleichzeitig die beiden Schlüsselkandidaten. Auch wenn wir korrekt annehmen, dass die Postleitzahl keine Determinante ist, bleibt die Verkäufernummer auf jeden Fall als Determinante erhalten. Die Verkäufernummer selbst ist kein Schlüsselkandidat, die Relation ist nicht in der dritten Normalform. Wir wollen daher unsere ursprüngliche Relation *Ver-*

kaeuferProdukt in die dritte Normalform überführen. Dazu zerlegen wir diese in drei neue Relationen. Eine solch optimale Zerlegung wird in Tab. 29 bis Tab. 31 vorgestellt. Beachten Sie bitte, dass jede dieser drei neuen Relationen in sich redundanzfrei ist.

Tab. 29 Relation Verkaeufer

Verk_Nr	Verk_Name	PLZ	Verk_Adresse
V1	Meier	80075	München
V2	Schneider	70038	Stuttgart
V3	Müller	50083	Köln

Tab. 30 Relation Produkt

Prod_Nr	Produktname
P1	Waschmaschine
P2	Herd
P3	Kühlschrank
P4	Staubsauger

Tab. 31 Relation Verknuepfung

Verk_Nummer	Produkt_Nr	Umsatz
V1	P1	11000
V1	P2	5000
V1	P3	1000
V2	P2	4000
V2	P3	3000
V3	P4	1000

Die Verbindung der beiden Relationen *Verkaeufer* und *Produkt* erfolgt über die Relation *Verknuepfung*. Wir werden im weiteren Verlauf dieses Kapitels noch sehen, dass dies das übliche und optimale Vorgehen ist, Relationen aufzubauen. Bei diesen kleinen Relationen sieht es so aus, als müssten wir hier mehr Daten speichern als in unserer ursprünglichen Relation *VerkaeuferProdukt*. Je mehr diese Relationen aber mit Daten gefüllt werden, um so deutlicher wird der Vorteil dieser drei einzelnen Relationen. Wir können jetzt jederzeit Produkte oder Verkäufer hinzufügen, ohne dass diese Produkte schon verkauft wurden, oder der Verkäufer schon etwas verkaufte. Nicht nur in Bezug auf die Normalformenlehre, auch wegen der besseren Handhabung haben damit diese drei Relationen enorme Vorteile gegenüber unserer ursprünglichen. Diese drei Relationen befinden sich tatsächlich in der dritten Normalform. Zum Beweis sei als Übung empfohlen, bei den drei Relationen jeweils die Primärschlüssel und alle Determinanten anzugeben.

Zuletzt sei noch erwähnt, dass, wie schon bei der zweiten Normalform, jede Relation in die dritte Normalform überführt werden kann, meist über eine Zerlegung der ursprünglichen Relation in mehrere neue. Diese Aussage gilt für beliebig komplexe Relationen.

Als Übung sei empfohlen, die Normalformen aller Relationen der Beispieldatenbank *Radl* in Anhang A zu bestimmen. Um nämlich auch kleinere Problemfälle aufzeigen zu können, befinden sich mit Absicht nicht alle Relationen dieser Datenbank *Radl* in der dritten Normalform. Geben Sie die Normalform jeder Relation an, und überführen Sie gegebenenfalls Relationen in die dritte Normalform.

5.1.3 Weitere Normalformen

Die dritte Normalform von Boyce/Codd reicht für fast alle Anwendungsfälle aus. Sie ist auch einfacher handhabbar als die ursprüngliche Definition von E. F. Codd. Diese Coddsche dritte Original-Normalform geht vom Begriff der Transitivität aus. Betrachten wir dazu die Relation *Verkaeufer* aus Tab. 29. Wir könnten geneigt sein, diese Relation um das Attribut *Bundesland* zu erweitern.

Tab. 32 Relation VerkaeuferLand

Verk_Nr	Verk_Name	PLZ	Verk_Adresse	Bundesland
V1	Meier	80075	München	Bayern
V2	Schneider	70038	Stuttgart	Baden-Württemberg
V3	Müller	50083	Köln	Nordrhein-Westfafen

In dieser erweiterten Relation (Tab. 32) gibt es jetzt mehrere volle funktionale Abhängigkeiten, die in Abb. 19 eingetragen sind. Hier erkennen wir, dass das Bundesland vom Schlüssel *Verk_Nr* sowohl direkt als auch über die Postleitzahl erreichbar ist. Wir sprechen allgemein von einer transitiven Abhängigkeit oder Transitivität, wenn ein Attribut von einem anderen Attribut über den Umweg eines dritten Attributs abhängig ist. Das Bundesland ist demnach zum Einen direkt, zum Anderen aber auch transitiv (über die Postleitzahl) abhängig vom Primärschlüssel.

Die Originaldefinition von Codd verbietet solche Transitivitäten. Sie lautet:

Definition (dritte Normalform nach Codd)

☞ Eine Relation ist in der dritten Normalform (nach Codd), wenn sie sich in der zweiten Normalform befindet, und jedes Nichtschlüsselattribut nicht transitiv vom Primärschlüssel abhängt.

Diese Definition der dritten Normalform ist schwächer als die Definition nach Boyce/Codd: Jede vorkommende Transitivität bedeutet, dass es Determinanten gibt, die nicht Schlüsselkandidaten sind. Genau diese Determinanten werden durch die Coddsche dritte Normalform verboten. Es kann aber darüberhinaus noch Determinanten geben, die nicht Schlüsselkandidaten sind! Als Übung sei empfohlen, Relationen zu finden, die sich in der Coddschen dritten Normalform, nicht aber in der dritten Normalform nach Boyce/Codd befinden. Als Hinweis sei angegeben, dass als Primärschlüssel ein zusammengesetztes Attribut geeignet zu wählen ist. Ist der Primärschlüssel nämlich ein einzelnes Attribut, so sind beide Definitionen der dritten Normalform gleichwertig.

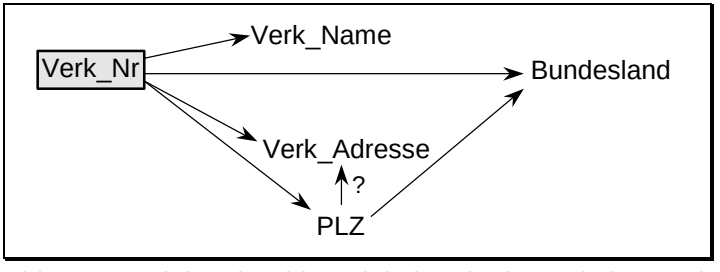


Abb. 19 Funktionale Abhängigkeiten in der Relation VerkaeuferLand

Die Boyce/Codd-sche Definition der dritten Normalform hat sich bewährt und ist der ursprünglichen Definition vorzuziehen. Wenn wir im Folgenden von der dritten Normalform sprechen, so ist immer die Definition nach Boyce/Codd gemeint.

Bis etwa 1974 war die Definition dieser Normalformen abgeschlossen. In praktischen Anwendungsfällen wurde aber bald festgestellt, dass es Relationen gibt, die zum Einen in der dritten Normalform sind, zum Anderen aber weiter redundant und schwer handhabbar bleiben. Aus diesem Grund wurden 1978 und 1979 die vierte und fünfte Normalform definiert. In der Praxis spielen diese Normalformen aber eine untergeordnete Rolle. Denn bereits die dritte Normalform reduziert die Redundanz in den Nichtschlüsselattributen. Die vierte und fünfte Normalform reduzieren nur noch etwaige Redundanzen innerhalb zusammengesetzter Schlüssel.

Ziel sollte es sein, die Primärschlüssel und weitere alternative Schlüssel nicht oder nur so weit nötig aus mehreren einzelnen Attributen zusammenzusetzen. Auch das im nächsten Abschnitt aufgezeigte Vorgehen zum Datenbankdesign mittels des Entity-Relationship-Modells lässt zusammengesetzte Primärschlüssel kaum entstehen. Die positive Folge ist, dass wir uns dann

nicht um die vierte oder gar fünfte Normalform kümmern müssen. Um die möglichen Probleme aber zumindest zu kennen, wollen wir ein Beispiel einer Relation mit einem mehrfach zusammengesetzten Primärschlüssel vorstellen.

Betrachten wir dazu wieder unsere Relation *VerkaeuferProdukt*. Gehen wir jetzt davon aus, dass ein Verkäufer durch die drei Attribute *Verkaeufername*, *PLZ* und *Adresse* eindeutig festgelegt ist. Dies können wir dann akzeptieren, wenn wir uns zur Adresse neben dem Wohnort noch die Straße und die Hausnummer als ergänzende Angaben vorstellen, und bei eventuellen Namensgleichheiten den Vornamen und gegebenenfalls den Zusatz *sen.* und *jun.* hinzufügen. Diese zusätzlichen Daten werden wir hier übersichtlichkeitshalber nicht aufnehmen. Entfernen wir nun aus dieser Relation das Attribut *Verk_Nr*, so besitzen wir eine Relation in dritter Normalform, wobei der Primärschlüssel aus den ersten vier Attributen *Verk_Name* + *PLZ* + *Verk_Adresse* + *Produktname* besteht, siehe Tab. 33.

Tab. 33 VerkaeuferProdukt_3NF

Verk_Name	PLZ	Verk_Adresse	Produktname	Umsatz
Meier	80075	München	Waschmaschine	11000
Meier	80075	München	Herd	5000
Meier	80075	München	Kühlschrank	1000
Schneider	70038	Stuttgart	Herd	4000
Schneider	70038	Stuttgart	Kühlschrank	3000
Müller	50083	Köln	Staubsauger	1000

Diese Relation kann aber nicht befriedigen. Die Redundanzen in dieser Relation sind erheblich, jedes Ändern der Adresse eines Verkäufers führt zu mehrfachen Änderungen in der Relation. Ein weiteres Problem ergibt sich aus der ersten Integritätsregel: Ist etwa zu einem neuen Verkäufer ein Datum nicht bekannt (etwa seine Postleitzahl), so kann dieser nicht, auch nicht teilweise, eingetragen werden!

An diesem Beispiel erkennen wir außerdem: Der Datenbankdesigner entscheidet über die Wahl des Primärschlüssels zu seinen Relationen. Der Designer setzt die Realität in seine Datenbank so um, dass seine Datenhaltung für ihn optimal gewählt ist. Kann es in der Datenbank nicht vorkommen, dass zwei Personen gleichen Namens die gleiche Adresse besitzen (etwa dank des Zusatzes *sen.* und *jun.*), so kann er den Namen zusammen mit der Adresse als Primärschlüssel wählen. Das Wort Design ist hier wirklich korrekt gewählt. Design ist eine Kunst und keine Naturwissenschaft. Es kommt hier sehr stark auf persönliche Erfahrungen und auch auf persönliche Vorlieben an. In unse-

rem Beispiel muss sich der Designer allerdings seiner schlechten Wahl bewusst sein, auch aus einem weiteren Grund. In einem Hochhaus könnte durchaus der Fall eintreten, dass zwei Personen den gleichen Namen besitzen, obwohl sie nicht verwandt sind. Ein Eintrag der zweiten Person in die Relation wäre dann grundsätzlich nicht möglich.

Die hier angesprochenen Probleme beseitigt erst die vierte Normalform. Betrachten wir dazu den Verkäufernamen, die Adresse und den Umsatz. Der Wert des Umsatzes hängt vom Fleiß des Verkäufers, nicht aber vom Namen des Verkäufers ab. Der Umsatz kann sich demnach beliebig ändern. Ein gänzlich anderes Verhältnis besteht zwischen dem Namen und der Adresse. Zu einer gegebenen Adresse gibt es nur eine bestimmte Anzahl von möglichen Namen, nämlich die Namen der dort Wohnenden. Wir sprechen von einer sogenannten mehrfachen Abhängigkeit. Diese liegt immer dann vor, wenn zu jedem Wert eines Attributs nur eine eng begrenzte Anzahl von Werten in einem anderen Attribut existiert. Die vierte Normalform fordert nun ausdrücklich, dass jede mehrfache Abhängigkeit auch eine funktionale Abhängigkeit nach sich zieht (die dritte Normalform wird zusätzlich vorausgesetzt). Obige Relation *VerkaeuferProdukt_3NF* ist daher nicht in der vierten Normalform.

Die fünfte Normalform oder Projektion-Join-Normalform schließlich verbietet, dass eine Relation nichttrivial in mehr als zwei Relationen zerlegt werden kann. Ohne weiter darauf einzugehen, betrachten wir nur unsere Relation *VerkaeuferProdukt*. Wir haben diese in die Relationen *Verkaeufer*, *Produkt* und *Verknuepfung* zerlegt. Da diese nichttriviale Zerlegung möglich ist, konnte sich die Relation *VerkaeuferProdukt* nicht in der fünften Normalform befinden. Tatsächlich ist die fünfte Normalform sehr einschränkend und schließt die vierte und damit alle anderen Normalformen ein.

Nachdem wir die fünf Normalformen kennengelernt haben, sei angemerkt, dass es keinen Zwang gibt, alle Relationen in die dritte oder gar fünfte Normalform zu bringen. Wir haben bereits das Problem mit der Postleitzahl angesprochen. In der Regel wird hier keine eigene Relation erzeugt, in der die Beziehungen zwischen Wohnort und Straße einerseits und Postleitzahl andererseits abgespeichert sind. Stattdessen wird fast immer die Adresse zusammen mit der Postleitzahl angegeben. Diese Relation ist dann eventuell nicht mehr in der dritten Normalform. Dies wird in der Praxis aber (fast) immer in Kauf genommen. Diese Ausnahmen sind sozusagen nur die Bestätigung der Regel, nur Relationen zu verwenden, die sich mindestens in der dritten Normalform befinden.

Wir wollen nicht verschweigen, dass eine Korrelation zwischen hohen Normalformen einerseits und der Anzahl der Relationen andererseits besteht. Eine große Anzahl von Relationen mit nur wenigen Attributen hat wiederum komplexe Zugriffe über mehrere Relationen (meist Joins) zur Folge, was letztlich längere Laufzeiten bewirkt. Ein Datenbankdesigner muss die Für und Wider genau abwägen.

Wichtig

- ☞ Primärschlüssel und alternative Schlüssel sind möglichst so zu wählen, dass sie nur aus ein oder höchstens zwei Attributen bestehen. Dies sind Aussage und Inhalt der vierten und fünften Normalform.
- ☞ Besitzt eine Relation einen nicht zusammengesetzten Primärschlüssel, so fallen die beiden Definitionen der dritten und die der vierten und fünften Normalform zusammen!

Oberstes Gebot ist daher, einfache, leicht verständliche Relationen zu entwerfen. Damit werden pathologische Fälle verhindert, wie etwa große zusammengesetzte oder überlappende Schlüssel.

In diesem Kapitel haben wir uns bisher nur mit dem Aufbau einzelner Relationen beschäftigt. Relationen in hoher Normalform sind bis auf Schlüsselkandidaten redundanzfrei. Wir dürfen aber nicht vergessen, dass eine Datenbank in der Praxis aus bis weit über 1000 Relationen bestehen kann. Unser Ziel ist es daher, ganze Datenbanken so aufzubauen, dass sie bis auf die Schlüssel (Schlüsselkandidaten und Fremdschlüssel) redundanzfrei sind.

5.2 Entity-Relationship-Modell

Im letzten Abschnitt haben wir kennengelernt, wie einzelne Relationen aufgebaut werden sollten. Mit diesem Wissen allein können wir aber noch keine Datenbank erstellen. Der Entwurf einer kompletten neuen Datenbank ist alles andere als trivial. Dies liegt weniger an der Komplexität des Problems, sondern am Umfang der zu speichernden Datenvielfalt. Ein vielfach bewährtes Werkzeug zur Erstellung einer redundanzfreien Datenbank ist das sogenannte Entity-Relationship-Modell, kurz ERM genannt. Dieses Modell wurde von P. P. Chen im Jahre 1976 vorgestellt.

Die Normalisierung der Relationen beseitigt Redundanzen in Nichtschlüsselattributen innerhalb einer Relation. Das Entity-Relationship-Modell verhin-

dert Redundanzen bis auf Fremdschlüssel auch relationenübergreifend. Dieses Modell beschränkt sich keineswegs nur auf relationale Datenbanken, es hat sich allgemein bewährt und sei wärmstens empfohlen. Dies schließt natürlich nicht aus, dass wir Datenbanken auch nach weiteren Gesichtspunkten entwerfen dürfen.

5.2.1 Entitäten

Bevor wir auf das Entity-Relationship-Modell eingehen, müssen wir einige Begriffe erklären. Es handelt sich hierbei um Begriffe, die wir zum Erstellen von relationalen Datenbanken benötigen. Schließlich müssen real vorhandene Daten und Gegebenheiten in einer Datenbank gespeichert werden, was zwangsweise zu Abstraktionen der realen Begriffswelt führen muss. Wertvolle Hilfestellungen leisten hierbei die Angaben in Tab. 34.

Tab. 34 Begriffe zu relationalen Datenbanken

Entität	ein unterscheidbares Objekt (im Sinne der betrachteten Objekte in der Datenbank), im mathematischen Sinne ein Element
Eigenschaft	ein Teil einer Entität, der die Entität beschreibt
Beziehung	eine Entität, die zwei oder mehr Entitäten miteinander verknüpft
Subtyp	eine Entität *Y*, die auch zu einer Entität *X* gehört
Supertyp	eine Entität, die Subtypen enthält

Wir wollen diese Begriffe aus Tab. 34 an einigen Beispielen erläutern. In Tab. 35 sind einige Angaben angegeben, die sich hinter diesen Begriffen verbergen können.

Tab. 35 Beispiele zu den Begriffen zu relationalen Datenbanken

Begriff	Beispiele
Entität	Person, Werkzeugteil, Produkt, Rechnung
Eigenschaft	Name, Vorname, PLZ, Ort, Adresse einer Person; Einzelteile, aus denen ein Werkzeug besteht; Rechnungsdatum
Beziehung	die Relation *Verknuepfung* verbindet die Relationen *Verkaeufer* und *Produkt* (siehe Tab. 31)
Subtyp	die Entität *Programmierer* ist ein Subtyp zur Entität *Person*
Supertyp	die Entität *Person* ist ein Supertyp der Entität *Programmierer*

In Abb. 20 finden wir schließlich eine Entität *Person*, die bestimmte Eigenschaften wie *Name*, *Gehalt*, *Adresse* oder *Personalnummer* besitzt. In diesem Beispiel ist die Adresse aus weiteren Eigenschaften (Postleitzahl, Ort, Straße und Hausnummer) zusammengesetzt (wir werden im Folgenden immer Entitäten als Rechtecke und Eigenschaften als Ellipsen zeichnen).

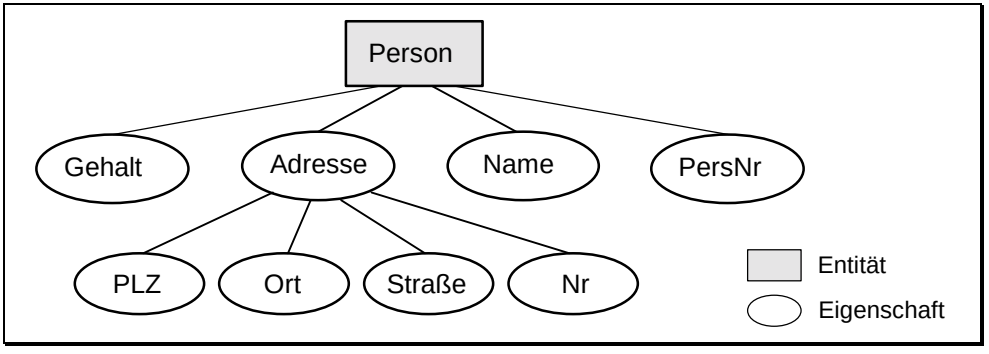


Abb. 20 Beispiel zu Entitäten und Eigenschaften

Es wird uns in der Praxis nicht schwerfallen, diese Eigenschaften *Gehalt*, *Adresse*, *Name* und *Personalnummer* einem Objekt, der Entität *Person*, zuzuordnen. In großen Datenbanken müssen wir jedoch systematisch vorgehen, um nicht die Übersicht zu verlieren. Zudem ist eine gewisse Erfahrung erforderlich, um die Entitäten und die Zuordnung der Eigenschaften zu diesen Entitäten optimal auszuwählen. Auf jeden Fall muss der Datenbankdesigner die reale Umgebung sehr genau kennenlernen, bevor er die dazugehörige Datenbank entwerfen kann.

Betrachten wir jetzt ein Beispiel zu Beziehungen: Es existiere neben der obigen Entität *Person* eine Entität (Relation) *Abteilung*, in der Daten zu den Abteilungen abgespeichert sind. Es gibt dann eine Beziehung zwischen den Personen der Entität *Person* und den einzelnen Abteilungen, da Personen jeweils einer Abteilung zugeordnet sind. Solche Beziehungen zeichnen wir in unserem Entity-Relationship-Diagramm als Raute. Abb. 21 zeigt ein solches Diagramm, wobei übersichtlichkeitshalber die einzelnen Eigenschaften der Entitäten *Abteilung* und *Person* nicht mit aufgeführt wurden. Die Zeichen 1 und *m* in der Abbildung klassifizieren die Beziehung. In diesem Fall gehören in der Regel mehrere Personen (*m*) einer Abteilung (1) an.

Subtypen sind spezielle Entitäten, die sich auf *Supertypen* beziehen. In Tab. 35 haben wir aus der Entität *Person* einen bestimmten Personenkreis (Programmierer) ausgewählt. Dieser Personenkreis besitzt in der Regel zusätzliche Eigenschaften oder Qualifikationen, die gespeichert werden sollen. Aus die-

sem Grund werden hierfür gerne eigene Entitäten als Subtypen zu den ursprünglichen Entitäten gewählt. Sollte der Programmierer noch in System- und Anwendungsprogrammierer aufgeteilt werden, so wäre *Programmierer* gleichzeitig ein Subtyp zur Entität *Person*, aber ein Supertyp zu den Entitäten *System-* und *Anwendungsprogrammierer*.

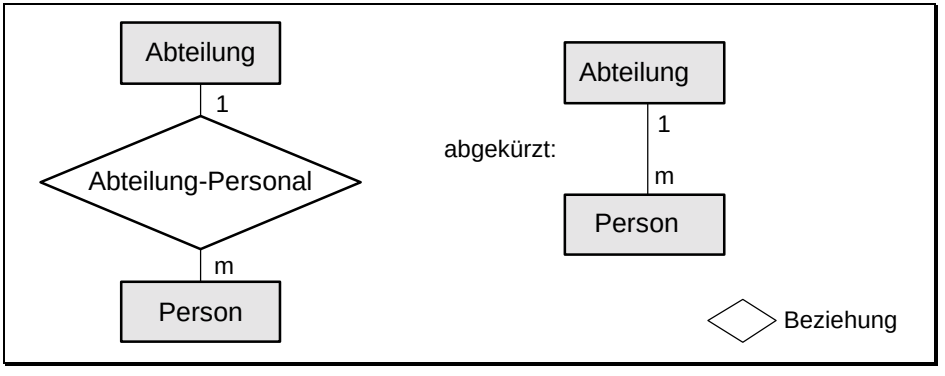


Abb. 21 Beispiel einer Beziehungsrelation

Weiter unterscheiden wir bei Entitäten noch zwischen starken und schwachen Entitäten. Wir bezeichnen eine Entität als schwach, wenn diese Entität von einer anderen Entität abhängt, und zwar in dem Sinne, dass sie ohne diese eine andere Entität nicht existieren könnte. Entitäten, die nicht schwach sind, heißen starke Entitäten. Es ist in der Praxis nicht immer einfach zu erkennen, ob eine Entität schwach ist oder nicht. Wir werden daher in diesem Abschnitt noch Hilfestellungen geben.

Ein schönes Beispiel zu schwachen und starken Entitäten liefert eine Produktions-Datenbank, die unter anderem die Entitäten *Produkt*, *Einzelteile* und *Fehler* enthält. Hier handelt es sich nacheinander um die hergestellten Produkte, um die Einzelteile der Produkte und eine Liste der bei der Produktion der einzelnen Produkte aufgetretenen Fehler. Die beiden Entitäten *Einzelteile* und *Fehler* stehen somit in einer Beziehung zur Entität *Produkt*. Sind alle denkbaren Fehler produktspezifisch, so ist die Entität *Fehler* schwach. Wird etwa ein Produkt aus der Produktion genommen, so sind die Fehlerdaten zu diesem Produkt wertlos. Einzelteile hingegen können für mehrere Produkte verwendet werden. Diese Entität kann daher nicht schwach sein.

Das Erstellen einer Datenbank mittels Entity-Relationship sollte jetzt prinzipiell klar sein. Die Hauptaufgabe wird es sein, die Entitäten der Datenbank zu ermitteln. Dazu werden wir zunächst alle Eigenschaften sammeln, die in

der zu erstellenden Datenbank abgespeichert werden sollen. Dies kann ein langwieriger und zeitaufwendiger Vorgang sein, da in der Regel mehrere Personen in der Ermittlung aller Eigenschaften involviert sind. Es kann sich hierbei auch um einen iterativen (und hoffentlich konvergenten) Vorgang handeln, bis schließlich alle gewünschten Eigenschaften exakt bestimmt sind. Anschließend werden alle Eigenschaften in zusammengehörige Objekte, die Entitäten, gruppiert.

Diese Entitäten wählen wir als die Relationen der neu zu schaffenden Datenbank. Wir sollten dabei gleich die Normalformen dieser Relationen mit berücksichtigen. Gegebenenfalls sind die Relationen in höhere Normalformen zu transformieren. Das Erzeugen dieser Relationen geschieht in SQL mit dem *Create-Table*-Befehl. Dieser könnte wie folgt aussehen:

```
CREATE TABLE  Personal
(   Persnr      INTEGER,
    Name        CHARACTER(20),
    ...
    PRIMARY KEY (Persnr)
);
```

Damit hätten wir den ersten Schritt im Entity-Relationship-Verfahren abgeschlossen, wir haben die Relationen (Entitäten) bestimmt. Im zweiten Schritt sind die Beziehungen (Relationship) zwischen den einzelnen Relationen aufzubauen.

5.2.2 Beziehungen

Wir haben bereits in Kapitel 3 kennengelernt, dass die Beziehungen zwischen den einzelnen Relationen durch Fremdschlüssel hergestellt werden. Wir haben deshalb diese Fremdschlüssel als den Kit einer Datenbank bezeichnet. Nachdem wir im letzten Unterabschnitt zeigten, wie wir Entitäten (Relationen) herleiten, ist es nun unsere Aufgabe, die Beziehungen zwischen den einzelnen Entitäten zu ermitteln und dann durch das Hinzufügen von Fremdschlüsseln zu verbinden. Um die Beziehungen zwischen den einzelnen Relationen zu bestimmen, ist es sehr vorteilhaft, sich nach der Ermittlung der Entitäten ein Entity-Relationship-Diagramm nach dem Muster von Abb. 21 aufzubauen. In diesem Diagramm werden alle Entitäten zusammen mit deren Beziehungen zueinander eingetragen.

Ein solches Diagramm ist enorm hilfreich. Es gibt in übersichtlicher Form die Struktur der Datenbank an. Ferner können leicht Entwurfsfehler erkannt und dann beseitigt werden. Ist etwa eine isolierte Entität vorhanden, die also keine Beziehungen zu anderen Entitäten besitzt, so ist zu fragen, ob diese Entität in dieser Datenbank korrekt untergebracht ist. Wir haben schließlich definiert, dass in einer Datenbank alle Daten in Beziehung zueinander stehen müssen.

Da ein Entity-Relationship-Diagramm den statischen Aufbau einer Datenbank vollständig beschreibt, wird es auch als Entity-Relationship-Modell (ERM) bezeichnet. Es ist ein Abbild (Modell) der zugrundeliegenden realen Welt.

Beziehungen zwischen den einzelnen Relationen können sehr vielfältiger Natur sein. Wir müssen daher diese Beziehungen genauer untersuchen. Dazu unterteilen wir die möglichen Arten von Beziehungen zwischen zwei Entitäten in drei Klassen: Es liegen entweder vor

- 1 zu 1 Beziehungen,
- m zu 1 (viele zu 1) Beziehungen oder
- m zu n (viele zu viele) Beziehungen.

Eine 1 zu 1 Beziehung zwischen zwei Entitäten *A* und *B* ist gegeben, wenn jeder Eintrag von *A* auf jeweils einen anderen Eintrag in *B* verweist und umgekehrt. Bei m zu 1 Beziehungen zwischen *A* und *B* können darüberhinaus mehrere Einträge von *A* mit dem gleichen Eintrag in *B* verknüpft sein. In m zu n Beziehungen gilt dies auch umgekehrt, so dass beliebige Zusammenhänge zwischen Einträgen der beiden Entitäten erlaubt sind.

1 zu 1 Beziehungen: Sie sind relativ selten. Manchmal werden etwa Zusatzinformationen zu Personen nicht direkt in der Personaltabelle, sondern in einer Zusatzrelation geführt, etwa aus Datenschutzgründen. Wenn nun zu jedem Mitarbeiter genau eine solche Zusatzinformation existiert, so liegt eine 1 zu 1 Beziehung vor. Wir schließen bei diesen Beziehungen den Spezialfall mit ein, dass zu einigen Mitarbeitern auch keine Zusatzinformationen gehalten werden, was genaugenommen einer 1 zu *c* Beziehung entspricht mit *c* gleich 0 oder 1.

m zu 1 Beziehungen: Wir haben eine solche Beziehung in Abb. 21 kennengelernt. Dort existiert zu jeder Abteilung mindestens ein Mitarbeiter aus der Entität *Person*. Umgekehrt ist jeder Mitarbeiter dieser Entität höchstens einer Abteilung zugeordnet. Wie schon in der 1 zu 1 Beziehung wollen wir auch

hier den Fall mit einschließen, dass eine Person keiner Abteilung zugeordnet ist (genaugenommen liegt dann eine m zu *c* Beziehung vor mit *c* gleich 0 oder 1). Ansonsten könnte etwa der Firmenchef, der keiner Abteilung zugeordnet ist, nicht der Entität *Person* angehören.

m zu n Beziehungen: Wir haben diese ebenfalls schon kennengelernt. In Tab. 29 und Tab. 30 sind die Relationen *Verkaeufer* und *Produkt* aufgeführt. Jeder Verkäufer kann mehr als ein Produkt verkaufen, gleichzeitig kann auch jedes Produkt von mehreren Verkäufern angeboten werden. Auch hier schließen wir ein, dass eventuell ein Produkt überhaupt nicht verkauft wurde, oder dass ein Verkäufer erst neu angefangen hat. Die Zahlen *m* und *n* dürfen im Spezialfall also auch Null werden.

Beginnen wir gleich mit einer ausführlichen Betrachtung der überaus wichtigen m zu n Beziehungen:

Wir können hier die erforderliche Beziehung nicht durch Hinzufügen von Fremdschlüsseln in den beiden betroffenen Entitäten (Relationen) erreichen. Wir müssen vielmehr eine eigene Entität (Relation) erzeugen, die diese Beziehung herstellt. Aus diesem Grund ist auch der Begriff *Beziehung* bzw. *Beziehungsentität* im Entity-Relationship-Modell so wichtig.

Eine Beziehungsentität zwischen zwei Entitäten enthält zwei Fremdschlüssel, die auf die Primärschlüssel der beiden Entitäten verweisen. Es liegen also Beziehungen wie zwischen *Verkaeufer* und *Produkt*, oder wie zwischen *Teilelieferant* und *Teile* vor. Sehr selten finden wir auch Beziehungen zwischen mehr als zwei Entitäten. Diese enthalten dann entsprechend mehr Fremdschlüssel. Da das Vorgehen bei Mehrfachbeziehungen analog zu dem bei Zweierbeziehungen ist, werden wir uns im Folgenden auf Zweierbeziehungen beschränken.

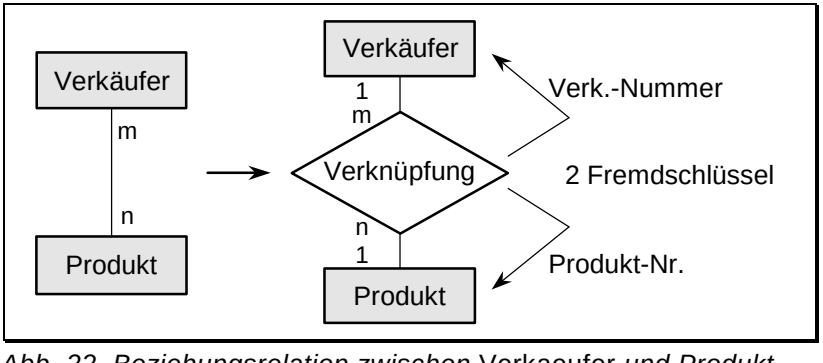


Abb. 22 Beziehungsrelation zwischen Verkaeufer *und Produkt*

Charakteristisch für Beziehungsrelationen ist, dass die beiden Fremdschlüssel zusammen immer ein Schlüsselkandidat sind. In der Relation *Verknuepfung* aus Tab. 31 sind dies die beiden Attribute *Verk_Nummer* und *Produkt_Nr*. Da in dieser Relation keine weiteren Schlüsselkandidaten existieren, ist dies gleichzeitig der Primärschlüssel. Wir hätten allerdings auch ein weiteres Attribut *Beziehungsnummer* hinzufügen können und diese fortlaufende Nummer als Primärschlüssel wählen können. Dies ist meist eine Geschmackssache und bleibt letztendlich dem Datenbankdesigner überlassen. Die Relation *Verknuepfung* wird in der Datenbanksprache SQL wie folgt erzeugt, wobei alle Schlüsselinformationen bereits mit angegeben sind:

```
CREATE TABLE  Verknuepfung
(   Verk_Nummer     CHARACTER(4)      REFERENCES Verkaeufer,
    Produkt_Nr      CHARACTER(4)      REFERENCES Produkt,
    Umsatz          SMALLINT,
    PRIMARY KEY  (Verk_Nummer, Produkt_Nr)
);
```

Wir erkennen an diesem SQL-Befehl, dass sich das Attribut *Verk_Nummer* auf die Relation *Verkaeufer* (genauer auf den Primärschlüssel dieser Relation) und *Produkt_Nr* auf die Relation *Produkt* beziehen. Ebenso ist der zusammengesetzte Primärschlüssel angegeben.

Die Relationen von Tab. 29 bis Tab. 31 sind in Abb. 22 in einem Entity-Relationship-Modell zusammengefasst worden. Jetzt, wo wir dieses Modell und den Begriff Entität kennen, fällt es uns leichter, eine nicht normalisierte Relation in eine höhere Normalform überzuführen. Beispielsweise enthielt die ursprüngliche *VerkaeuferProdukt*-Relation zwei Entitäten (Verkäufer und Produkt). Wir müssen solche Relationen nach ihren Entitäten zerlegen und mit einer Beziehungsrelation verknüpfen.

Unser Vorgehen zum Entwurf einer Datenbank ist also vorgezeichnet. Nachdem wir die Entitäten und damit die Relationen festgelegt haben, überlegen wir uns die Beziehungen zwischen den einzelnen Relationen und erstellen hierzu Beziehungsentitäten. Der Aufbau dieser Beziehungsrelationen (zumindest im Falle einer m zu n Beziehung) wurde in den letzten Absätzen erklärt. In der Praxis geschehen die Herleitung der Entitäten und der Beziehungen meist parallel.

Allerdings ist damit der Aufbau einer Datenbank noch nicht vollständig abgeschlossen. Der Datenbankdesigner muss sich mit den Fremdschlüsseln noch eingehender beschäftigen, wenn der spätere Zugriff auf die Datenbank nicht

zum Alptraum werden soll. Im wesentlichen müssen zu jedem Fremdschlüssel drei Fragen beantwortet werden:

- Darf ein Fremdschlüsselwert leer bleiben (mit Inhalt *Null*)?
- Darf ein Entitätseintrag gelöscht werden, auf den sich ein Fremdschlüssel bezieht?
- Darf ein Entitätseintrag geändert werden, auf den sich ein Fremdschlüssel bezieht?

Die Antwort auf die erste Frage beeinflusst die beiden anderen und sollte deshalb genau überlegt werden. In einigen Fällen kann es sehr wohl sinnvoll sein, den Fremdschlüsselwert auf *Null* zu setzen. Es kann etwa sein, dass in einer Firma eine Person keiner Abteilung angehört, z. B. der Leiter der Firma oder eine Person mit abteilungsübergreifenden Aufgaben. Andererseits wird eine Rechnung immer an einen Kunden geschickt, in einer Relation (Entität) *Rechnung* ist im entsprechenden Fremdschlüssel der Eintrag *Null* nicht sinnvoll und daher nicht zuzulassen.

Die anderen beiden Fragen hängen mit der Referenz-Integritätsregel zusammen, siehe Unterabschnitt 3.3.2. Wir erinnern uns, dass sich nach dieser Regel jeder Fremdschlüsselwert (ungleich *Null*) auf einen existierenden Eintrag beziehen muss. Wird jener Eintrag jedoch gelöscht oder der Primärschlüssel jenes Eintrags geändert, so wäre diese Integritätsregel verletzt!

Befassen wir uns daher eingehend mit der obigen zweiten Frage, dem gewünschten Löschen eines Eintrags. Das Löschen eines Eintrags hat wegen der Referenz-Integritätsregel auch einen Einfluss auf alle sich auf diesen Eintrag beziehenden Fremdschlüssel. Bereits beim Erstellen eines Fremdschlüssels (und damit beim Erzeugen einer Relation) muss daher festgelegt werden, wie im Falle des Löschens eines Eintrags, auf den sich der Fremdschlüssel bezieht, zu reagieren ist. Es gibt wegen der Integritätsregel nur drei Möglichkeiten:

① NO ACTION (Restrict): Das Löschen des Eintrags wird zurückgewiesen, wenn mindestens ein Fremdschlüsselwert auf diesen Eintrag verweist.
② CASCADE: Mit dem Löschen des Eintrags werden auch alle Tupel gelöscht, die sich auf diesen zu löschenden Eintrag beziehen.
③ SET NULL: Mit dem Löschen des Eintrags werden alle Fremdschlüsselwerte auf *Null* gesetzt, die sich auf diesen zu löschenden Eintrag beziehen.

Die Operationen *No Action* und *Set Null* bedürfen kaum einer weiteren Erläuterung. Wir erkennen, dass ein Eintrag einer Relation nicht gelöscht werden darf, wenn mindestens ein Fremdschlüsselwert auf diesen Eintrag verweist, dessen Fremdschlüssel die Eigenschaft *On Delete No Action* besitzt. Die Eigenschaft *Set Null* darf natürlich nur dann vergeben werden, wenn der Fremdschlüssel den Wert *Null* annehmen darf.

Etwas komplexer verhält es sich mit der Eigenschaft *Cascade*. Ein Löschen eines Eintrags führt hier automatisch auch zum Löschen von Tupeln, die sich auf diesen Eintrag beziehen. Nun können aber auch auf diese letzteren Tupel Fremdschlüssel mit der Eigenschaft *Cascade* verweisen und so fort. Dies kann zu einem kaskadierenden Löschen vieler Tupel in vielen Relationen führen. Es ist aber zu beachten, dass das Löschen all dieser Tupel nur dann möglich ist, wenn in der zu löschenden Kette kein *No Action* ein Löschen verbietet. Andernfalls wird das Löschen aller Tupel zurückgeweisen. Weiter sei vermerkt, dass natürlich das erste Auftreten von *Set Null* die kaskadierende Kette abbricht, ebenso wenn kein weiterer Verweis zu einem Eintrag existiert.

Die gleichen drei Möglichkeiten existieren auch im Falle einer gewünschten Änderung des Primärschlüssels eines Eintrags. Auch hier sind *No Action* und *Set Null* selbsterklärend. Die Eigenschaft *On Update Cascade* führt zu einer Anpassung des entsprechenden Fremdschlüsselwertes. Hier taucht ein kaskadierender Update nur in den seltenen Fällen auf, dass der Fremdschlüssel Teil des Primärschlüssels ist und sich ein weiterer Fremdschlüssel auf diese Relation bezieht. In diesem Spezialfall gilt das Analogon zum besprochenen Fall des kaskadierenden Löschens.

Der Datenbankdesigner muss sich dieser Eigenschaften der Fremdschlüssel unbedingt bewusst sein und sich bei jeder der obigen Fragen für genau eine Lösung entscheiden. Er stellt damit die Weiche für die erlaubten Zugriffe auf die Datenbank. In SQL-2 wurden diese Möglichkeiten aufgenommen, nur sind diese wichtigen Eigenschaften der Fremdschlüssel bisher erst in wenigen Datenbanken implementiert. In SQL-2 könnte unsere Beziehungsrelation *Verknuepfung* einschließlich der referentiellen Integrität wie folgt aussehen:

```
CREATE TABLE  Verknuepfung
(   Verk_Nummer CHAR(4) REFERENCES  Verkaeufer
                        ON DELETE NO ACTION  ON UPDATE CASCADE,
    Produkt_Nr  CHAR(4) REFERENCES  Produkt
                        ON DELETE NO ACTION  ON UPDATE CASCADE,
    Umsatz      SMALLINT,
    PRIMARY KEY  (Verk_Nummer, Produkt_Nr)
) ;
```

Werden in SQL-2 die referentiellen Integritätsregeln nicht mittels *On Delete* und *On Update* spezifiziert, so wird automatisch das Löschen und Ändern abgewiesen, es wird demnach standardmäßig die SQL-Klausel *No Action* gesetzt. In vielen modernen Datenbanken werden grafische Unterstützungen zum Datenbankdesign angeboten. Meist ist hier ebenfalls *Restrict* sowohl für das Ändern als auch das Löschen voreingestellt.

Die Relation *Verknuepfung* enthält damit alle für uns wesentlichen Eigenschaften. Ideal wäre es nun, wenn bei jedem Zugriff auf eine Datenbank alle diese im *Create-Table*-Befehl angegebenen Bedingungen automatisch überprüft und gegebenenfalls mit einer Fehlermeldung abgewiesen würden. Dieser Forderung kommen immer mehr Datenbankhersteller gemäß der SQL-2 Spezifikation nach. Beispielsweise wird von Oracle und MS-Access die Fremdschlüsselbedingung *No Action* immer überprüft.

Unsere Überlegungen zu den Fremdschlüsseln gelten für alle Beziehungen zwischen den einzelnen Entitäten, wenn wir bis jetzt auch nur die m zu n Beziehungen näher kennengelernt haben. Wir werden aber gleich sehen, dass alle anderen Beziehungen nur Sonderfälle dieser m zu n Beziehungen sind, und meist sogar eine einfachere Handhabung erlauben.

Für m zu 1 Beziehungen gelten die gleichen Aussagen wie für m zu n Beziehungen. Allerdings benötigen wir keine eigene Beziehungsrelation. Es genügt, in der Viele-Beziehung (Angabe: *m*) einen Verweis (ein Fremdschlüsselattribut) auf die 1-Beziehung hinzuzufügen. Betrachten wir das Beispiel der Beziehung zwischen den Entitäten *Abteilung* und *Person* aus Abb. 21. Wir erweitern die Relation *Person* durch ein Attribut *Abteilungsnr*, das Auskunft darüber gibt, zu welcher Abteilung ein Mitarbeiter gehört:

```
CREATE TABLE Person
(   PersNr          SMALLINT  PRIMARY KEY,
    Name            Character (20),
    . . .
    Abteilungsnr    SMALLINT  REFERENCES Abteilung
                              ON DELETE SET NULL   ON UPDATE CASCADE
);
```

Wir erkennen, dass damit die m zu 1 Beziehung schon komplett beschrieben ist, eine eigene Beziehungsrelation ist überflüssig. Genaugenommen hatten wir bei m zu n Beziehungen eine Beziehungsrelation nur deshalb benötigt, um diese Beziehung in zwei m zu 1 Beziehungen zerlegen zu können. Dies können wir deutlich in Abb. 22 erkennen.

Wie bereits erwähnt, sind 1 zu 1 Beziehungen recht selten. Sie werden analog dem m zu 1 Fall behandelt. Häufig ist hier der Primärschlüssel einer der beiden Relationen auch der Fremdschlüssel, so dass kein zusätzliches Attribut erforderlich wird. Da die Referenz-Integritätsregel erfüllt sein muss, tritt der Fremdschlüssel daher immer in derjenigen Relation auf, die weniger Tupel enthält.

Wir hätten damit das Erstellen einer Datenbank nach dem Entity-Relationship-Modell komplett vorgestellt. Wir möchten an dieser Stelle jedoch nochmals auf zwei bereits erwähnte Begriffe zurückkommen, auf schwache Entitäten und auf Subtypen. Wir wissen bereits, dass eine schwache Entität von einer anderen Entität abhängt. Es liegt folglich eine m zu 1 oder eine 1 zu 1 Beziehung vor.

Betrachten wir das Beispiel einer Firma, die zu jedem Mitarbeiter die Arbeitszeiten elektronisch erfasst. In der Datenbank existiert eine Relation *Arbeitszeit*, in der diese Zeiten zu jedem Mitarbeiter für das letzte Quartal abgespeichert sind. Die Daten eines Mitarbeiters in der Relation *Arbeitszeit* sind wertlos, falls dieser aus der Firma ausscheidet. Es handelt sich bei der Relation *Arbeitszeit* tatsächlich um eine schwache Entität. Das Entity-Relationship-Diagramm hierzu finden wir in Abb. 23. Wir haben dort die schwache Entität mit einem Doppelrechteck gekennzeichnet.

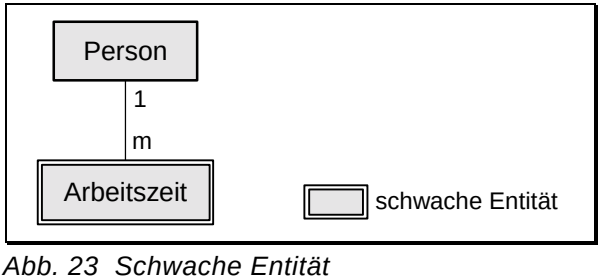


Abb. 23 Schwache Entität

Die schwache Entität *Arbeitszeit* besitzt einen Fremdschlüssel auf die Entität *Person*. Die Eigenschaften dieses Fremdschlüssels sind:

```
NOT NULL
ON UPDATE CASCADE
ON DELETE CASCADE
```

Dies ist leicht einsichtig. Natürlich muss sich jedes Arbeitszeitdatum auf einen Mitarbeiter beziehen, so dass ein *Null*-Wert im Fremdschlüssel nicht mög-

lich ist. Erhält ein Mitarbeiter eine andere Personalnummer, so muss diese Änderung auch in der Entität *Arbeitszeit* nachvollzogen werden. Wird ein Mitarbeiter in der Relation *Person* gelöscht, so sind auch alle Daten in der Entität *Arbeitszeit* wertlos und somit ebenfalls zu entfernen.

In schwachen Entitäten besitzt der Fremdschlüssel immer die drei obigen Eigenschaften. Es gilt aber auch die Umkehrung: Jede Entität, die sich nur auf eine einzige andere Entität bezieht und diese drei Eigenschaften ihres Fremdschlüssels besitzt, ist eine schwache Entität! Auf diese Art und Weise kann eine schwache Entität leicht identifiziert werden.

Zuletzt gehen wir noch auf Subtypen ein. Wir betrachten dazu nochmals das Beispiel aus Tab. 35. Gegeben sei also unsere bereits bekannte Entität *Person*, wobei für das technische Personal (Programmierer, Ingenieure) noch Zusatzinformationen abzuspeichern sind. Geschieht dies direkt in der Relation *Person*, so blieben für das nichttechnische Personal viele Attribute leer. Aus diesem Grund ist es naheliegend,eine Relation *TechPerson* einzuführen, die diese Zusatzinformation für das technische Personal aufnimmt. Es liegt eine 1 zu 1 Beziehung vor (genauer eine 1 zu *c* Beziehung mit *c* gleich 0 oder 1). Die Relation *TechPerson* besitzt den gleichen Primärschlüssel wie die Relation *Person*. Dieser Primärschlüssel ist gleichzeitig der Fremdschlüssel auf die Relation *Person*. Dieses Zusammenfallen von Primär- und Fremdschlüssel ist eine typische Eigenschaft eines Subtyps. Weiter ist leicht zu erkennen, dass ein Subtyp eine schwache Entität darstellt, wir können daher die Eigenschaften des Fremdschlüssels leicht angeben. Wir erhalten:

```
CREATE TABLE TechPerson
(   PersNr   SMALLINT  PRIMARY KEY
                       REFERENCES Person
                       ON DELETE CASCADE   ON UPDATE CASCADE,
    . . .
) ;
```

In dieser Relation muss für *PersNr* nicht explizit *Not Null* angegeben werden, da dies für Primärschlüssel ja automatisch gilt.

5.3 Zusammenfassung

Wir haben in diesem Kapitel kennengelernt, Relationen in Normalformen einzuordnen, und wir haben gesehen, wie wir mit Hilfe des Entity-Relation-

ship-Modells ganze Datenbanken erstellen können. Die Normalformenlehre und das Entity-Relationship-Modell haben sich in der Praxis bewährt, sie werden weltweit eingesetzt, und auch wir können beide Entwicklungsmethoden nur wärmstens weiterempfehlen.

In der Praxis dürfen wir beim Datenbankdesign die Normalformenlehre und das Entity-Relationship-Modell nicht voneinander trennen. Die beiden Werkzeuge verzahnen sich ineinander. Schon bei der Wahl der Entitäten werden wir beispielsweise darauf achten, möglichst hohe Normalformen zu erreichen. Fassen wir nochmals kurz zusammen: Das Design geschieht in mehreren Schritten, wobei auch Iterationen zugelassen sind, weil etwa während des Entwurfs noch Zusatzanforderungen mit aufgenommen werden müssen. Die einzelnen Schritte sind:

- Auswahl aller Eigenschaften, die in die Datenbank aufgenommen werden sollen (meist ein langwieriger und zeitaufwendiger Prozess). Eine gute Auswahl schon in der Anfangsphase verhindert ein mehrmaliges und zeitraubendes Anpassen der Datenbank.
- Zusammenfassen der Eigenschaften zu Entitäten. Hier ist die Normalformenlehre zu berücksichtigen. Bei einer guten Auswahl der Entitäten werden die Relationen meist automatisch in dritter oder noch höherer Normalform sein.
- Ermittlung der Beziehungen zwischen den einzelnen Entitäten.
- Erzeugen der Relationen, Erzeugen der Beziehungsrelationen bei m zu n Beziehungen, Hinzufügen von Fremdschlüsseln bei 1 zu m und 1 zu 1 Beziehungen.
- Ermittlung der Eigenschaften der Fremdschlüssel. Hier helfen auch Stichworte wie *schwache Entität* und *Subtyp* weiter.

Dieses relativ kurze Kochrezept zum Erstellen einer Datenbank ist immer anwendbar, sei es bei kleinen oder großen Datenbanken. Es sei allerdings nicht verschwiegen, dass bei größeren Datenbanken leicht die Übersicht verloren werden kann. Hier ergeben sich oft 100 und mehr Entitäten, allein das ERM erstreckt sich über mehrere Seiten. In solch großen Datenbanken bilden wir zusammengehörige Untergruppen (Schemata, wir kommen im nächsten Kapitel darauf zurück) wie Rechnungswesen, Personalwesen oder Einkauf, deren Beziehungen zu anderen Untergruppen relativ gering sind. Auch hier ist wieder der Designer gefragt, eine feste technische Anleitung gibt es nicht. Sehr große Datenbanken ab etwa 100 Entitäten sind nur noch schwer zu überblicken. Hier wird in der Regel auf spezielle Case-Tools zurückgegriffen.

Und vergessen wir nicht: Ein Design zu einer Datenbank wird nur einmal durchgeführt. Die Folgen eines schlechten Entwurfs bekommen wir aber während der gesamten Lebensdauer dieser Datenbank zu spüren. Komplexe Zugriffe und schlechte Performance sind noch die harmlosen Auswirkungen. Bei Datenverlusten und Inkonsistenzen wird es ernst.

5.4 Übungsaufgaben

1) Beweisen Sie, dass sich jede Relation mit nicht zusammengesetztem Primärschlüssel in der zweiten Normalform befindet.
2) Geben Sie alle vollen funktionalen Abhängigkeiten
   a) in den Relationen *VerkaeuferProdukt* (siehe Tab. 13), *Chemische Elemente* (siehe Tab. 17) *und VerkaeuferProdukt_2NF* (siehe Tab. 28),
   b) in allen Relationen der *Radl*-Datenbank aus Anhang A an.
3) Geben Sie zu allen Relationen aus Aufgabe 2 die Determinanten an.
4) Geben Sie zu den Relationen *Verkaeufer*, *Produkt* und *Verknuepfung* aus Tab. 29 bis Tab. 31 alle Determinanten an.
5) Bestimmen Sie die Normalformen aller Relationen der *Radl*-Datenbank aus Anhang A.
6) Geben Sie eine Relation an, die in der Coddschen dritten Normalform, jedoch nicht in der dritten Normalform nach Boyce/Codd ist. Hinweis: Diese Relation muss einen aus mindestens zwei Attributen bestehenden Primärschlüssel besitzen. Betrachten Sie ferner eine Relation mit mehreren Schlüsselkandidaten, deren Attribute sich überlappen!
7) Geben Sie alle Fremdschlüssel der Relationen *Personal*, *Kunde* und *Auftrag* aus Tab. 18 bis Tab. 20 auf Seite 79 an. Bestimmen Sie auch die Eigenschaften dieser Fremdschlüssel.
8) Geben Sie zur Beispieldatenbank *Radl* an, welche Entitäten schwach sind, bei welchen Entitäten es sich um Beziehungsrelationen handelt und ob Sub- und Supertypen vorliegen.
9) Eine Schwäche der Datenbank *Radl* ist das Auftragswesen. Erweitern Sie daher die Datenbank durch ein Rechnungswesen. Die neue Relation *Rechnung* sollte mindestens ein Rechnungsdatum, den Rechnungsbetrag und Informationen über einen Rabatt, erfolgte Mahnung und erfolgte Bezahlung enthalten. Ergänzen Sie das Entity-Relationship-Modell aus Abb. 54 auf Seite 298. Geben Sie alle neuen Fremd- und Primärschlüssel und die Eigenschaften der Fremdschlüssel an.

6 Die Datenbankbeschreibungssprache SQL

Wir haben in Kapitel 4 kennengelernt, mit Hilfe von SQL auf bestehende Datenbanken zuzugreifen. Wir benötigen dazu in SQL nur vier Befehle, den *Select*-, *Insert*-, *Delete*- und *Update*-Befehl. Allerdings war insbesondere der *Select*-Befehl sehr umfangreich und entsprechend mächtig.

Die Beschreibungssprache in SQL verhält sich fast konträr dazu: Hier gibt es deutlich mehr Befehle, die dann allerdings im Aufbau relativ einfach sind. Es existieren Befehle zum Erzeugen, Ändern und Löschen von Tabellen und Sichten; wir können Indizes auf Attribute setzen und wieder entfernen, um dann schneller nach diesen Begriffen suchen zu können. Darüberhinaus existieren Befehle, um Zugriffsrechte auf einzelne oder mehrere Relationen einer Datenbank zu vergeben und bei Bedarf wieder zu entziehen. Weiter gibt es Befehle, um die Integrität einer Datenbank zu sichern, und schließlich bleiben noch die Befehle, um eine Datenbank selbst einzurichten.

Beim Erzeugen und Ändern von Relationen können Integritätsbedingungen angegeben werden, die später bei jedem Zugriff automatisch abgeprüft werden. Die möglichen Bedingungen zur referentiellen Integrität wurden im wesentlichen schon im letzten Kapitel angesprochen und werden hier ergänzt, während Einzelheiten und Beispiele zur semantischen Integrität erst in Kapitel 8 im Abschnitt über Integrität vorgestellt werden, etwa die *Check*-Bedingung und die Befehle *Create Domain und Create Assertion*. Auch die Befehle *Grant* und *Revoke* zum Vergeben und Entziehen von Zugriffsrechten werden erst in Kapitel 8 im Abschnitt über Sicherheit besprochen.

Was in SQL schon für die Zugriffssprache galt, trifft noch mehr auf die Beschreibungssprache zu: Die Erweiterungen von SQL-1+ und insbesondere SQL-2 gegenüber SQL-1 sind enorm. Vor allem die Möglichkeiten der Angabe von Integritätsbedingungen in den neuen Normen sind wichtig, und wir wollen und müssen sie behandeln. Wieder sei auch auf Anhang B verwiesen, wo die Syntax aller hier verwendeten SQL-Befehle zusammengefasst ist.

Besonderheiten in Oracle und MS-Access werden am Ende dieses Kapitels in einem eigenen Abschnitt behandelt. Auch die erst in SQL-2 definierten Systemtabellen werden in einem weiteren Abschnitt angesprochen, ebenso die Systemtabellen in Oracle.

6.1 Relationen erzeugen, ändern und löschen

Bereits im letzten Kapitel haben wir an Hand von Beispielen kennengelernt, Relationen (Tabellen) mit Hilfe des *Create-Table*-Befehls zu erzeugen. In diesem Abschnitt werden wir jetzt alle Möglichkeiten des Einrichtens von Relationen aufzeigen. Die Syntax des hierzu benötigten *Create-Table*-Befehls lautet:

```
CREATE TABLE   Tabellenname
(   {   Spalte   {  Datentyp  |  Gebietsname  }   [  Spaltenbedingung [ ... ] ]
    |   Tabellenbedingung        }
    [ ,  ...  ]
)
```

Die verwendete Notation wurde bereits zu Beginn des Kapitels 4 angesprochen und ist ausführlich im Anhang B beschrieben. Hier sei nur kurz erwähnt, dass die in Großbuchstaben angegeben Namen reservierte Bezeichner in SQL sind. Alle anderen Angaben sind entweder Namen oder werden im weiteren noch näher spezifiziert. Die in eckigen Klammern angegebenen Werte sind wahlfrei, und bei Angabe des senkrechten Strichs zusammen mit geschweiften Klammern ist nur eine der Möglichkeiten zu verwenden. Obiger Syntax entnehmen wir also, dass ein *Create-Table*-Befehl mit der Eingabe des Tabellennamens beginnt, gefolgt von einer Attributsliste. Jedes Attribut wird durch einen Spaltennamen identifiziert, besitzt einen anzugebenden Datentyp und kann wahlweise durch Spaltenbedingungen noch genauer spezifiziert werden. Wahlweise können in der Attributsliste auch Tabellenbedingungen angegeben werden. Sowohl die Spalten- als auch die Tabellenbedingungen (im englischen: *column constraint* bzw. *table constraint*) legen zusätzliche Eigenschaften fest, etwa Primär- und Fremdschlüsselangaben.

Die wichtigsten der im SQL-Standard vordefinierten Datentypen sind in Tab. 36 aufgelistet. Darüberhinaus lassen sich eigene Datentypen mit Hilfe von *Gebieten* definieren. Wir kommen in Kapitel 8 im Absatz zur Integrität darauf zurück.

Die Größe der Datentypen *Integer* und *Smallint* ist implementierungsabhängig, wobei sichergestellt ist, dass *Integer*-Zahlen mindestens so groß wie *Smallint*-Zahlen sind. In MS-Access werden *Integer*-Zahlen als 4-Byte-Zahlen und *Smallint*-Zahlen als 2-Byte-Zahlen abgespeichert. In Oracle sind beide 38-stellig. Die y-Werte in der Tabelle dürfen auch weggelassen werden. Es wird dann die Zahl Null angenommen. Das zusätzliche Weglassen der x-Werte ist

ebenfalls möglich, wobei dann implementierungsbedingte Werte angenommen werden. Darüberhinaus entspricht die Angabe *Character* ohne Parameter dem Vorgabewert *Character(1)*, ebenso besitzt *Bit* ohne Parameter die Bedeutung *Bit(1)*. Aus Kompatibilitätsgründen werden in SQL-2 auch die Datentypen *Real* und *Double Precision* unterstützt. Sie entsprechen beide dem Typ *Float* mit einer implementierungsabhängigen Anzahl von Nachkommastellen.

Tab. 36 Liste der Datentypen in SQL

INTEGER	Ganzzahl
INT	Kurzform von INTEGER
SMALLINT	Ganzzahl
NUMERIC (x,y)	x stellige Dezimalzahl mit y Nachkommastellen
DECIMAL (x,y)	mindestens x stellige Dezimalzahl mit y Nachkommastellen
FLOAT (x)	Gleitpunktzahl mit x Nachkommastellen
CHARACTER (n)	Zeichenkette der festen Länge n
CHAR (n)	Kurzform von CHARACTER (n)
CHARACTER VARYING (n)	Variable Zeichenkette mit bis zu n Zeichen
VARCHAR (n)	Kurzform von CHARACTER VARYING (n)
BIT (n)	Bitleiste der festen Länge n
DATE	Datum (Jahr, Monat, Tag)
TIME	Uhrzeit (Stunde, Minute, Sekunde)

Der Datentyp *Date* ist notwendig, um etwa Geburtsdaten oder andere Termine speichern zu können. Wichtig ist, dass Variablen vom Typ *Date* miteinander verglichen werden können. Ebenso können diese Variablen voneinander subtrahiert werden. Das Ergebnis ist dann eine Integerzahl, die die Anzahl der Tage zwischen diesen Daten enthält. Auch die Addition oder Subtraktion einer Datumsvariable mit einer Integerzahl ist erlaubt. Das Datum wird entsprechend um die Anzahl Tage erhöht bzw. erniedrigt. Insgesamt ist das Arbeiten mit dieser Variable sehr praktisch. Wegen der unterschiedlichen nationalen Schreibweisen und der Zeitzonenangaben treten allerdings sowohl beim Datentyp *Date* als auch bei *Time* (Uhrzeit) gewisse Probleme beim Umgang mit diesen Variablen auf. Eine ausführliche Beschreibung zu den Datentypen *Date* und *Time* finden wir etwa in [DaDa98] und [MeSi93].

Häufig wird in Abfragen das aktuelle Datum benötigt. Deshalb wurde in SQL-2 die Konstante *Current_Date* vom Datentyp *Date* definiert. Diese Kon-

stante enthält das aktuelle Datum. Gestern war also *Current_Date – 1* und morgen ist *Current_Date + 1*, um zwei einfache Beispiele zu nennen. Oracle und MS-Access kennen diese Konstante nicht und verwenden stattdessen die Konstante *Sysdate* (Oracle) beziehungsweise die Funktion *Date* (MS-Access).

Es sei noch erwähnt, dass mit Hilfe der Funktion *Cast* ein Datentyp in einen anderen explizit umgewandelt werden kann. Dies ist insbesondere beim Umwandeln von Datums- in Zeichenkettenfelder und umgekehrt wichtig. Die folgende Funktion erzeugt ein Datumsfeld aus der angegebenen Zeichenkette:

```
CAST ( '1995-12-24' AS DATE )
```

Diese SQL-2-Funktion ist in MS-Access nicht und in Oracle nur sehr eingeschränkt ab Version 8.0 implementiert. MS-Access und Oracle besitzen eigene Implementierungen, um Zeichenketten, die ein gültiges Datum enthalten, in den Datentyp *Date* zu konvertieren: *to_date* (in Oracle) und *CDate* (in MS-Access). In MS-Access gibt es auch den Wahrheitswert *Boolean*. Dieser Typ wird von SQL nicht unterstützt und muss gegebenenfalls durch den Datentyp *Character(1)* oder *Bit(1)* nachgebildet werden.

Wir wollen vor allem auf die für den Aufbau einer Datenbank enorm wichtigen *Spalten*- und *Tabellenbedingungen* genauer eingehen, wobei wir uns besonders auf die Möglichkeiten zur Unterstützung der referentiellen Integrität in SQL-1+ und SQL-2 konzentrieren.

In vielen Fällen reicht es, einzelne Attribute mittels *Spaltenbedingungen* einzuschränken. Mögliche Einschränkungen sind:

- [CONSTRAINT Bedingungsname] NOT NULL
- [CONSTRAINT Bedingungsname] { PRIMARY KEY | UNIQUE }
- [CONSTRAINT Bedingungsname]
  REFERENCES Tabellenname [(Spalte [, ...])]
  [ON DELETE { NO ACTION | CASCADE | SET NULL }]
  [ON UPDATE { NO ACTION | CASCADE | SET NULL }]
- [CONSTRAINT Bedingungsname] CHECK (Bedingung)

Jedes einzelne Attribut kann wahlweise durch eine oder mehrere aufeinanderfolgende Bedingungen eingeschränkt werden. Wir finden hier die Angaben zum Primärschlüssel wieder, die Angaben zu Fremdschlüsseln einschließlich der referentiellen Integritätsregel und die Angaben zur Überprüfung von Werten mittels der *Check*-Bedingung. Die Angabe *Not Null* erlaubt keine Nullwerte in diesem Attribut und *Unique* erzwingt die Eindeutigkeit des Attributs.

Beides ist natürlich automatisch für den Primärschlüssel gegeben. Bei alternativen Schlüsseln muss das Attribut *Unique* allerdings explizit gesetzt werden.

Eine *Check*-Bedingung entspricht eins zu eins der Bedingung in der *Where*-Klausel des *Select*-Befehls. Mittels der *Check*-Bedingung wird festgelegt, welche Bedingungen ein Attribut erfüllen muss. Soll ein Attribut *attr* etwa immer zwischen 10 und 90 liegen, so wird dies mittels der Spaltenbedingung

```
CHECK ( attr BETWEEN 10 AND 90 )
```

sichergestellt. Weitere Beispiele zur *Check*-Bedingung werden in Kapitel 8 zur semantischen Integrität aufgeführt.

Mit der wahlfreien Angabe von Bedingungsnamen zusammen mit dem reservierten Wort *Constraint* erhalten die einzelnen Bedingungen einen eindeutigen Namen zwecks späterer Identifikation. Dies ist dann wichtig, wenn eine Bedingung eventuell wieder gelöscht werden soll. Wir kommen beim *Alter-Table*-Befehl darauf zurück.

Es sei darauf hingewiesen, dass im *Check*-Ausdruck nur auf das jeweilige Attribut selbst verwiesen werden darf. Auch können zusammengesetzte Schlüssel nicht als Spaltenbedingung definiert werden, da eine Spalte durch die Angaben *Primary Key* oder *References* für sich alleine bereits als Schlüssel festgelegt wird. Bei „spaltenübergreifenden" Bedingungen und Schlüsseln sind daher Spaltenbedingungen überfordert, wir müssen auf Tabellenbedingungen zugreifen. Diese sind ähnlich wie Spaltenbedingungen aufgebaut, nur dass hier zusätzlich anzugeben ist, auf welche Attribute sich die Bedingungen beziehen. Die Syntax der drei möglichen Tabellenbedingungen ist im Folgenden aufgelistet:

- [CONSTRAINT Bedingungsname]
  { PRIMARY KEY | UNIQUE } (Spalte [, ...])
- [CONSTRAINT Bedingungsname]
  FOREIGN KEY (Spalte [, ...])
  REFERENCES Tabellenname [(Spalte [, ...])]
  [ON DELETE { NO ACTION | CASCADE | SET NULL }]
  [ON UPDATE { NO ACTION | CASCADE | SET NULL }]
- [CONSTRAINT Bedingungsname]
  CHECK (Bedingung)

Die Angaben *Primary Key*, *Unique* und *Foreign Key* müssen in der Tabellen- statt in der Spaltenbedingung erfolgen, wenn aus mehreren Attributen zusammengesetzte Schlüssel vorliegen. Wird *On Delete* oder *On Update* nicht

spezifiziert, so wird standardmäßig *No Action* gesetzt, das Ändern bzw. Löschen also gegebenenfalls unterbunden. Im *Check*-Ausdruck dürfen alle Attribute der Relation verwendet werden, in Unterabfragen sogar beliebige Attribute anderer bereits existierender Relationen. Analog zu Spaltenbedingungen können wir auch hier mittels des Bezeichners *Constraint* einen Bedingungsnamen angeben. Eine spätere Modifikation mittels eines *Alter-Table*-Befehls ist damit jederzeit möglich (siehe weiter unten).

Wir hätten damit die Syntax zum Erzeugen von Relationen abgeschlossen und geben als Beispiel das Erzeugen der bereits früher verwendeten Relation *Personal* (siehe Tab. 18 auf Seite 79) an:

```
CREATE TABLE Personal
(       Persnr      INTEGER             PRIMARY KEY,
        Name        CHARACTER (25)      NOT NULL,
        Ort         CHARACTER (15),
        Vorgesetzt  SMALLINT            REFERENCES Personal
                                ON DELETE SET NULL   ON UPDATE CASCADE,
        Gehalt      NUMERIC (8,2)
                                CHECK ( Gehalt BETWEEN 800.00 AND 9000.00 ),
        CONSTRAINT MaxVerdienst
                        CHECK ( Gehalt >= ( SELECT MIN(Gehalt) FROM Personal ) )
) ;
```

Wir haben hier die Primär- und Fremdschlüssel identifiziert, ebenso wurde das Gehalt auf ein Intervall eingeschränkt und mittels einer Tabellenbedingung so nach unten eingeschränkt, dass ein neu einzutragendes Gehalt das bisherige minimale Gehalt nicht unterschreiten darf.

Beachten Sie bei der Erzeugung von Relationen dringend, dass ein späteres Ändern der Struktur dieser Relationen immer mit Aufwand verbunden ist. Alle Zugriffe auf diese Relationen in allen Programmen müssen angepasst werden. Ein optimales Datenbankdesign ist daher die beste Vorbeugung. Hier gilt die bereits erwähnte Devise: einmal schlechtes Design – immer schlechte Anwendung. Nach dem etwas zeitaufwendigen Design der Datenbank ist dann das eigentliche Erzeugen der Relationen relativ einfach, die erforderlichen *Create-Table*-Befehle sind schnell programmiert. Es sollte der Aufwand nicht gescheut werden, alle bekannten Einschränkungen mittels Spalten- und Tabellenbedingungen anzugeben. Nur so kann das Datenbankverwaltungssystem von sich aus Integritätsverletzungen erkennen. Dieser einmalige Aufwand sichert auf einfache Weise die Integrität der Datenbank.

An dieser Stelle sei noch erwähnt, dass Verletzungen der angegebenen Spalten- und Tabellenbedingungen im laufenden Betrieb zur Zurückweisung

des entsprechenden Zugriffsbefehls führen. Es wird ein entsprechender Returncode zurückgeliefert. Entsprechend dieses Codes muss dann das Programm die geeigneten Maßnahmen treffen. Wir verweisen hier auf Kapitel 9, wo SQL-Befehle in Programme eingebettet werden und die Technik der Abfrage des Returncodes geübt wird.

Haben wir eine für unsere Verhältnisse optimale Datenbank entworfen und realisiert, so müssen wir mit der Zeit doch damit rechnen, dass gelegentlich Änderungen an der Datenbankstruktur vorzunehmen sind. Dies ist immer dann erforderlich, wenn Änderungen in der realen Welt nachzuvollziehen sind. Für solche Änderungen, meist Erweiterungen, steht der SQL-Befehl *Alter Table* zur Verfügung. Die Syntax lautet:

```
ALTER TABLE Tabellenname
   {  ADD [ COLUMN ]    Spalte  { Datentyp | Gebietsname }
                                                [ Spaltenbedingung  [ … ] ]
   |  DROP [ COLUMN ]  Spalte  { RESTRICT | CASCADE }
   |  ADD  Tabellenbedingung
   |  DROP CONSTRAINT  Bedingungsname  { RESTRICT | CASCADE }
```

Wir können demnach mit einem *Alter-Table*-Befehl entweder ein neues Attribut (Spalte) hinzufügen oder ein bestehendes Attribut entfernen oder eine weitere Tabellenbedingung angeben oder eine existierende Tabellen- oder Spaltenbedingung verwerfen. Wir erkennen jetzt, wie wichtig es sein kann, die Angabe einer Bedingung mittels eines eindeutigen Bezeichners zu identifizieren. Nur so kann diese Bedingung auch wieder entfernt werden. Beim Löschen von Attributen oder Tabellen- und Spaltenbedingungen muss eine der Bedingungen *Restrict* oder *Cascade* angegeben werden. Bei *Restrict* wird das Löschen verhindert, wenn noch Sichten existieren, die dieses Attribut verwenden, oder wenn noch Integritätsbedingungen zu diesem Attribut vorliegen. Bei *Cascade* wird das entsprechende Attribut auch in diesen Sichten gelöscht, ebenso darauf verweisende Integritätsbedingungen. Ist das zu löschende Element eine Tabellenbedingung, so hat *Restrict* nur bei Fremdschlüsselbedingungen eine Bedeutung. Gibt es noch Verweise, so wird das Löschen unterbunden. Bei *Cascade* werden die Verweise ebenfalls gelöscht.

Soll, aus welchen Gründen auch immer, eine Relation samt der darin gespeicherten Tupel entfernt werden, so ist dies mit dem *Drop-Table*-Befehl möglich. Die Syntax lautet:

```
DROP TABLE   Tabellenname   { RESTRICT | CASCADE }
```

Seit SQL-2 ist einer der beiden Bezeichner (*Restrict* oder *Cascade*) zwingend. Wird *Restrict* gesetzt, wird ein Löschen der Relation verweigert, falls noch Verweise auf diese Relation existieren. Im Falle von *Cascade* werden entsprechende Verweise (Tupel) ebenfalls gelöscht. Im praktischen Einsatz ist das Löschen ganzer Relationen oder auch nur das Löschen einzelner Attribute oder Tabellenbedingungen selten. Nur beim Entstehungsprozess einer neuen Datenbank noch während der Testphase besitzt ein *Drop*-Befehl eine Existenzberechtigung.

6.2 Erzeugen und Entfernen eines Index

Die im letzten Abschnitt behandelten Basisrelationen bilden den Grundstock einer jeden Datenbank. Doch diese noch etwas grobe Basis bedarf eines Feinschliffs. Ein besonders wichtiger Schliff ist die Performance der Datenbank. Um die Suche nach Informationen auch bei sehr großen Datenbeständen schnell durchführen zu können, sind Sekundärschlüssel (siehe Abschnitt 1.8) erforderlich. In SQL-1 werden diese Sekundärschlüssel über sogenannte Indexe realisiert. Diese Indexe werden nicht in der Relation selbst, sondern extern gespeichert. Dies gestattet, diese Indexe auch wieder zu entfernen. Insbesondere ermöglicht dieses Vorgehen ein jederzeitiges Hinzufügen weiterer Indexe, falls dies erforderlich sein sollte.

Dies ist auch der Grund, warum wir uns beim Erzeugen der Relationen noch keine Gedanken über diese Indexe machen mussten. Zum Hinzufügen und Entfernen solcher Indexe stehen eigene Befehle zur Verfügung. Diese lauten:

```
CREATE [UNIQUE] INDEX   Name   ON
                Tabellenname  (  { Spalte  [ ASC | DESC ] }  [ , ... ]  )

DROP INDEX  Name
```

Wie wir erkennen, kann sich ein Index auch über mehrere Attribute erstrecken, wobei noch angegeben werden kann, ob ein Index aufsteigend (*Asc*) oder absteigend (*Desc*) sortiert werden soll. Standardmäßig ist aufsteigende Sortierung eingestellt.

In SQL-2 wurde dieser Befehl nicht aufgenommen. Es bleibt dem Systemverwalter vorbehalten, mit mächtigen herstellerabhängigen Befehlen die Leistung der Datenbank zu optimieren. So legt etwa Oracle automatisch zu jedem Primärschlüssel einen Index an. Und nicht nur in MS-Access können wir mit

folgendem Befehl einen Index auf den Primärschlüssel der Relation *Personal* setzen:

```
CREATE UNIQUE INDEX IPers ON Personal (Persnr);
```

Besonderes Augenmerk sei hier auf das Wort *Unique* gelegt. In diesem Fall wird die Eindeutigkeit des Attributs erwartet und auch überprüft. Es sollte bei Schlüsselkandidaten immer mit angegeben werden.

6.3 Sichten (Views)

Die Möglichkeiten des Erzeugens von Basisrelationen mit dem *Create-Table*-Befehl haben wir im vorletzten Abschnitt behandelt. Wir haben aber bereits in Kapitel 3 kennengelernt, dass es neben Basisrelationen noch weitere Relationen gibt: Sichten oder virtuelle Relationen (im englischen: *View*) sind von Basisrelationen abgeleitete Relationen, wobei in der Datenbank nicht deren Inhalt, sondern nur die Ableitungsregeln abgespeichert werden!

Der Sinn von Sichten liegt darin, den Benutzern auch andere Betrachtungsweisen von Relationen zu geben, ohne dass diese Relationen real existieren, wodurch Redundanzen vermieden werden. Auch im Falle des Zugriffsschutzes gibt es wichtige Anwendungen, wenn zum Beispiel nur auf Teile einer Relation von Dritten zugegriffen werden darf. Schließlich existieren noch Möglichkeiten zur Überprüfung von Integritätsbedingungen mit Hilfe von Sichten.

Wir wollen an Hand von Beispielen die Anwendungsmöglichkeiten von Sichten aufzeigen, zunächst geben wir aber die Syntax zum Erzeugen und Löschen von Sichten an:

```
CREATE VIEW  Sichtname  [ (  Spalte  [ , ... ]  ) ]    AS   Select-Befehl
       [ WITH CHECK OPTION ]

DROP VIEW  Sichtname   { RESTRICT  |  CASCADE }
```

Wir erkennen, dass eine Sicht mittels einer beliebigen *Select*-Abfrage (ohne *Order-By*-Klausel) aufgebaut wird. Fehlen die Spaltennamen im *Create-View*-Befehl, so werden die Attributsnamen aus der Spaltenliste des *Select*-Befehls übernommen. Stehen in diesem *Select*-Befehl in der Spaltenliste jedoch nicht nur einzelne Attributsnamen, sondern beliebige Spaltenausdrücke, so müssen alle Spaltennamen der Sicht explizit aufgeführt werden.

Analog zum *Drop-Table*-Befehl ist ab SQL-2 beim *Drop-View*-Befehl eine der beiden Angaben *Restrict* oder *Cascade* zwingend. Bei der Angabe von *Restrict* wird das Löschen der Sicht zurückgewiesen, wenn andere Sichten oder Tabellen- und Spaltenbedingungen existieren, die auf diese Sicht verweisen. Wird *Cascade* angegeben, so werden auch diese Sichten und Bedingungen mit entfernt.

Betrachten wir ein erstes Beispiel zur Anwendung von Sichten. Die Relation *Auftrag* aus Tab. 19 ist in dieser Form für den Anwender kaum lesbar, da Personal- und Kundennummern in der Regel nicht auswendig bekannt sind. Hier könnte folgende Sicht weiterhelfen:

```
CREATE VIEW VAuftrag (AuftrNr, Datum, Kundenname, Persname) AS
    SELECT AuftrNr, Datum, Kunde.Name, Personal.Name
    FROM    Auftrag, Kunde, Personal
    WHERE Kunde.Nr = Auftrag.Kundnr AND Personal.Persnr = Auftrag.Persnr;
```

Das Ergebnis dieser Sicht *VAuftrag* ist in Tab. 37 zu sehen. Jeder Auftrag erscheint zusammen mit dem Verkäufer- und Käufernamen.

Tab. 37 Sicht VAuftrag

AuftrNr	Datum	Kundenname	Persname
1	04.08.1998	Fahrrad Shop	Anna Kraus
2	06.09.1998	Maier Ingrid	Johanna Köster
3	07.10.1998	Rafa – Seger KG	Anna Kraus
4	18.10.1998	Fahrräder Hammerl	Johanna Köster
5	06.11.1998	Fahrrad Shop	Anna Kraus

Mit Hilfe dieser Sicht *VAuftrag* genügt jetzt folgender Befehl, um alle Aufträge anzusehen, die Frau Köster tätigte:

```
SELECT  * FROM VAuftrag
 WHERE Persname = 'Johanna Köster' ;
```

Wir erkennen, dass damit die Zugriffsbefehle wesentlich vereinfacht werden, was zur Übersichtlichkeit in den Programmen beiträgt. Es bleibt Aufgabe der Datenbank, unter Zuhilfename der Sichtdefinition den kompletten Suchbefehl auf die zugrundeliegenden Basisrelationen selbst zu erzeugen und auszuführen. Vermutlich wird intern folgender umfangreiche Befehl für die Ausgabe generiert:

```
SELECT  AuftrNr, Datum, Kunde.Name AS Kundenname,
        Personal.Name AS Persname
FROM    Auftrag, Kunde, Personal
WHERE Kunde.Nr = Auftrag.Kundnr  AND  Personal.Persnr = Auftrag.Persnr
   AND  Personal.Name = 'Johanna Köster' ;
```

Wir sehen bereits an diesem Beispiel, dass immer dann Sichten zu empfehlen sind, wenn eine andere Sichtweise auf Teile der Datenbank gewünscht und öfter verwendet wird. Sichten verbergen die zugrundeliegende komplexe Datenstruktur und erlauben somit dem Anwender einfache Zugriffsbefehle. Bei der Verwendung von Sichten müssen wir aber immer bedenken, dass es sich um keine echten Relationen handelt, sondern dass eine Abfrage, definiert in der Sichtdeklaration (*Create View*), dahintersteckt. Das Datenbankverwaltungssystem baut, wie wir oben bereits gezeigt haben, aus einer Abfrage auf eine Sicht eine Abfrage auf die entsprechenden Basisrelationen auf. Das System nimmt dazu die Sichtdefinition zu Hilfe.

Änderungen in den zugrundeliegenden Basisrelationen sind damit in allen Sichten sofort abfragbar, mit dem Vorteil, dass keine Inkonsistenzen wegen doppelter Datenhaltung entstehen können. Erinnern wir uns an ein Beispiel in Kapitel 2. Der dortige Getränkehändler führt die Relationen *Eingang*, *Ausgang* und *Lager*. Um Inkonsistenzen von vornherein zu verhindern, sollte die Relation *Lager* als Sicht definiert werden, da aus Ein- und Ausgang der momentane Bestand eindeutig berechnet werden kann.

Grundsätzlich dürfen auf Sichten beliebige DML-Befehle angewendet werden. Allerdings darf in Sichten weder geändert, gelöscht oder eingefügt werden, wenn der *Select*-Befehl in der Sichtdefinition

- auf mehr als eine Relation zugreift,
- eine *Group-By*-Klausel enthält,
- eine *Distinct*-Angabe enthält,
- die Spaltenliste nicht nur aus einzelnen Spaltennamen besteht,
- Mindestens einen der Operatoren *Union*, *Intersect* oder *Except* enthält.

Ebenso darf eine Sicht nicht verändert werden, wenn diese auf einer Sicht basiert, die eine der fünf obigen Bedingungen erfüllt. Diese Einschränkungen sind nicht willkürlich. Bei jeder der obigen Bedingungen kann passieren, dass Tupel der zugrundeliegenden Relation in der Sicht zusammengefasst wurden. Dies hat zur Folge, dass ein einzelnes Tupel in der Sicht mehreren realen Tupeln entsprechen kann. Ein Löschen eines einzelnen Tupels der Sicht könnte

dann das Löschen mehrerer Tupel der Basisrelation zur Folge haben. Ähnlich verhält es sich mit Änderungen oder dem Einfügen. Im letzten Fall wüsste die Datenbank nicht, wie viele Tupel einzufügen sind.

Betrachten wir als Beispiel unsere oben definierte Sicht *VAuftrag*. Diese Sicht enthält eine Liste von Aufträgen, unter anderem gefolgt vom Gesamtauftragsvolumen. Sicherlich wäre es jetzt gefährlich, hier ein Tupel zu löschen, da das System nicht wüsste, ob nur Auftragsdaten oder auch Personal- und Kundendaten zu löschen wären. Genauso problematisch wäre das Einfügen eines neuen Tupels: Würde ein neuer Kundenname angegeben, so wüsste das System nicht, ob neben dem neuen Auftrag auch neue Kundendaten eingetragen werden sollen.

Zuletzt wollen wir noch eine weitere wichtige Anwendung von Sichten kennenlernen. Betrachten wir dazu eine Basisrelation *Vereinsmitglieder*, die die Namen und die dazugehörigen Daten der einzelnen Mitglieder eines Vereins enthält. Da in diesem Verein eine intensive Jugendarbeit betrieben wird, könnte eine Sicht *Jugend* von Interesse sein. Diese würde dann einfach aus der Basisrelation abgeleitet:

```
CREATE VIEW Jugend AS
   SELECT * FROM Vereinsmitglieder WHERE Alter < 21 ;
```

Hier wurde vorausgesetzt, dass ein Attribut *Alter* in der Relation existiert. Dem Leiter der Jugendarbeit ist der Zugriff auf die Relation *Jugend* gestattet, jedoch nicht auf die Relation *Vereinsmitglieder*. Wir sehen hier einen weiteren wichtigen Sinn von Sichten: Zugriffssicherheit. Der Jugendleiter hat keinen Zugriff auf die umfassende Relation *Vereinsmitglieder*, sondern nur auf die für ihn relevanten Teileinträge.

Diese Sicht *Jugend* ist entsprechend den obigen Ausführungen beliebig veränderbar, alle DML-Befehle sind erlaubt. Der Jugendleiter führt auch regelmäßig Änderungen durch: An jedem Geburtstag eines Mitglieds wird das Alter um eins erhöht. Dieses Erhöhen funktioniert einwandfrei. Wird das Jugendmitglied jedoch 21 Jahre alt, so kann der Jugendleiter diese Änderung nach dem Update nicht mehr sehen, da der Jugendliche nun nicht mehr in der Sicht enthalten ist (aber natürlich noch in der Basisrelation!). Versehentliche Änderungen können daher von ihm nicht mehr rückgängig gemacht werden.

Die Option *With Check Option* weist alle Manipulationen zurück, die dazu führen würden, dass Tupel nicht mehr der Sicht angehören würden. In unserem Beispiel versehen wir daher den *Create-View*-Befehl mit dieser Option:

```
CREATE VIEW Jugend AS
   SELECT * FROM Vereinsmitglieder WHERE Alter < 21
   WITH CHECK OPTION ;
```

Jetzt würde der Versuch, das Alter eines zwanzigjährigen Mitglieds von der Sicht *Jugend* aus anstatt von der Basisrelation *Vereinsmitglieder* zu erhöhen, mit einer Fehlermeldung abgewiesen werden. Wir haben damit eine weitere Möglichkeit kennengelernt, Integritätsbedingungen anzugeben. In SQL-1 war dies die einzige. In Kapitel 8 finden wir im Abschnitt Sicherheit weitere Beispiele zum Zugriffsschutz mit Hilfe von Sichten und im Abschnitt Integrität weitere Beispiele zu der Option *With Check Option*.

6.4 Kataloge und Schemata

In der Praxis besitzen größere Datenbanken eine hohe Anzahl von Relationen. Um hier nicht den Überblick zu verlieren, wurde SQL-2 dahingehend erweitert, dass Relationen, die untereinander eng verknüpft sind, in einem Verbund (Schema) zusammengefasst werden können. Dies verbessert die Übersicht auch bei sehr großen Datenbeständen. Die Sprache SQL lässt dabei genug Freiheiten, um diese Gruppierungsmöglichkeiten den eigenen Bedürfnissen optimal anzupassen.

Genaugenommen kennt die Sprache SQL das Wort *Datenbank* nicht. SQL besitzt daher auch keine Sprachkonstrukte wie *Create Database*. In der Praxis haben daher die einzelnen Datenbankhersteller ihr eigenes SQL-ähnliches Gerüst darübergestülpt. Wir gehen darauf im nächsten Abschnitt näher ein.

SQL-2 besitzt folgende Philosophie: Jeder Benutzer kann, vorausgesetzt sein Rechner ist vernetzt, auf fast unbeschränkte Daten zugreifen. Der Benutzer schließt sich dazu zunächst an einen SQL-Server an. Dies geschieht mit Hilfe des SQL-Befehls *Connect*. Die Syntax dieses Befehls lautet etwas vereinfacht (zu weiteren Details siehe Anhang B):

```
CONNECT TO { DEFAULT | SQL-Servername }
```

Wird der *Default*-Wert benutzt, so ist damit zu rechnen, dass wir mit dem lokalen SQL-Server verbunden werden. Dies ist allerdings abhängig vom Datenbankhersteller und im Standard nicht näher spezifiziert. Im Einbenutzerbetrieb ist dieser Befehl überflüssig, da SQL auch eine Standardverbindung erlaubt. Diese Standardverbindung wird gewählt, falls dieser Befehl nicht ange-

geben wird. Heißt ein Server beispielsweise *Server13*, so baut folgender Befehl eine entsprechende Verbindung auf:

```
CONNECT TO 'Server13' ;
```

Zum Beenden der Verbindung steht der Befehl

```
DISCONNECT { DEFAULT | CURRENT | SQL-Servername }
```

zur Verfügung. Die häufigste Angabe ist *Current*. Damit wird die aktuelle SQL-Server-Verbindung beendet. Sind mehrere Verbindungen gleichzeitig geöffnet, so ist die zu schließende Verbindung explizit anzugeben.

Um alle nur denkbaren Möglichkeiten abzudecken, unterteilt SQL-2 alle Daten auf einem Server in Kataloge, diese wiederum in Schemata, und diese Schemata enthalten dann Basisrelationen, Sichten, Zugriffs- und Integritätsregeln. In der Praxis wird es daher sinnvoll sein, ein Schema als einen zusammengehörigen Teil einer komplexen Datenbank anzusehen, etwa das Personal- oder Rechnungswesen, oder Ein- und Verkauf. Mehrere solcher Teile (Schemata) ergeben eine Datenbank und können in SQL in einem Katalog verwaltet werden. Doch auch andere Konfigurationen sind denkbar, SQL erlaubt hier freie Hand. Allerdings ist in SQL der Katalog die Einheit für die Verwaltung der Systemtabellen, siehe hierzu Abschnitt 6.6.

Der Sinn dieser zweistufigen Datenverwaltung liegt darin, dass zum Einen auf einem Rechensystem mehrere Datenbanken gleichzeitig verwaltet werden können, und zum Anderen viele Benutzer nur auf bestimmte benutzerspezifische Teile dieser Datenbanken zugreifen. Genaugenommen greift ein Benutzer nach einem *Connect* meist nur auf eine einzige Datenbank zu. Weitere Konstrukte in SQL (*Cluster*) ermöglichen nun dem Systemverwalter, Benutzer auf bestimmte Kataloge einzuschränken. Ohne auf weitere Details einzugehen, sei nur soviel erwähnt, dass Kataloge in SQL die Zugriffseinheit für den Benutzer sind. Inwieweit er zugreifen darf, legt der Systemverwalter fest. Die Schemata in SQL sind wiederum eine logische Zusammenfassung einer Datenbankstruktur.

Nur innerhalb einer solchen logischen Einheit (*Schema*) müssen Namen von Relationen oder Integritätsregeln eindeutig sein. Greift ein Benutzer schemaübergreifend zu, so muss er im Zweifelsfall den gewünschten Namen qualifizieren: Jedem Namen einer Basisrelation, Sicht, Zugriffs- oder Integritätsregel kann dazu der Schemaname, gefolgt von einem Punkt, vorangestellt werden. Gehört die Relation *Auftrag* etwa zum Schema *Rechnungswesen*, so kann die

Relation vollständig als *Rechnungswesen.Auftrag* qualifiziert werden. Damit lässt sich diese Relation von einer weiteren gleichen Namens in einem anderen Schema eindeutig unterscheiden.

Zum Festlegen eines Schemas einschließlich des Erzeugens aller Relationen und Sichten und inklusive Zugriffs- und Integritätsregeln dient folgender SQL-2 Befehl:

```
CREATE SCHEMA [ Schemaname ] [ Schemaelement [ , ... ] ]
```

Der Schemaname kann wieder qualifiziert werden, indem der Katalogname, gefolgt von einem Punkt, vorangestellt wird. Ansonsten wird der standardmäßig eingestellte Katalog verwendet (kann mit dem *Set-Catalog*-Befehl geändert werden). Die Schemaelementliste enthält eine Liste aller Elementdefinitionen (z.B. *Create-Table*-Definitionen zum Erzeugen von Relationen), die zum Schema gehören. Gleichzeitig wird dieses Schema dem Benutzer zugeordnet, der diesen Befehl absetzte.

Ein Schema kann auch wieder gelöscht werden. Hierzu existiert der folgende Befehl:

```
DROP SCHEMA Schemaname { CASCADE | RESTRICT }
```

Mit der Angabe von *Restrict* wird das Löschen des Schemas zurückgewiesen, falls noch Schemaelemente existieren. Die Angabe von *Cascade* ist sehr vorsichtig einzusetzen. Hier wird das Schema immer entfernt, zusammen mit all seinen Elementen, also auch allen darin enthaltenen Relationen und deren Inhalten.

Betrachten wir ein einfaches Beispiel eines Schemas. Unsere Basisrelationen *Personal*, *Kunde* und *Auftrag* aus Tab. 18 bis Tab. 20 auf Seite 79 und unsere im vorherigen Abschnitt definierte Sicht *VAuftrag* könnten wir in einem Schema mittels folgenden Befehls zusammenfassen:

```
CREATE SCHEMA Radl
        CREATE TABLE Personal  (   ...        ),
        CREATE TABLE Kunde     (   ...        ),
        CREATE TABLE Auftrag   (   ...        ),
        CREATE VIEW VAuftrag   (   ...        ),   ...     ;
```

Natürlich sollten in diesem Schema auch alle Zugriffsrechte (*Grant*-Befehle, siehe Kapitel 8), alle Indexe (*Create-Index*-Befehle) und weitere Integritätsbefehle (*Create-Assertion*- und *Create-Domain*-Befehle, siehe Kapitel 8) mit

aufgeführt werden. Würden in unserem Beispiel auf unserem SQL-Server bereits mehrere Kataloge existieren, so hätten wir unser Schema *Radl* noch einem Katalog explizit zuordnen können, etwa dem Katalog *Beispieldatenbank* mittels des qualifizierten Schemanamens *Beispieldatenbank.Radl*. Dem Schema *Radl* können auch noch zu einem späteren Zeitpunkt Elemente hinzugefügt werden. Im entsprechenden *Create*-Befehl muss der Name des neuen Elements nur durch den Schemanamen qualifiziert werden. Kataloge selbst können nicht mit Hilfe von SQL-Befehlen erzeugt oder gelöscht werden. Dazu sind implementierungsabhängige Befehle erforderlich.

6.5 Besonderheiten in Oracle und MS-Access

Wie bereits im letzten Abschnitt erwähnt wurde, beschreibt erst SQL-2 den logischen hierarchischen Aufbau von Datenbeständen mittels Katalogen und Schemata. Ganz allgemein war der Sprachumfang von SQL-1 nicht hinreichend, um alle in der Praxis auftretenden Fälle abdecken zu können. Zwangsläufig entstand so eine Vielzahl von SQL-Dialekten. Erst mit SQL-2, und damit erst seit 1992, steht ein auch in der Praxis ausreichender Sprachumfang zu Verfügung. Die meisten Datenbankhersteller arbeiten zur Zeit an einer Anpassung ihrer Dialekte an die neue SQL-Norm. Dies wird wegen Kompatibilitätsproblemen zwangsläufig noch einige Zeit in Anspruch nehmen. Wir wollen daher hier die bisherige Entwicklung am Beispiel von Oracle und MS-Access im Bereich der Beschreibungssprache herausgreifen.

Oracle unterstützt SQL in einem sehr weiten Bereich. Bereits ab der Version 6.0 erfüllt Oracle die SQL-1+ Norm syntaxmäßig vollständig. Ab Version 7.2 wird SQL-2 voll unterstützt, wenn auch nur im Entry-Level. Dies beinhaltet Tabellen- und Spaltenbedingungen und die Fremdschlüsselbeziehungen. Es fehlt allerdings die Möglichkeit zur Angabe aller Fremdschlüsseleigenschaften (nur *On Delete Cascade* wird unterstützt), ebenso die Definition von Gebieten, Assertions (siehe Kapitel 8) und Schemata. Dafür bietet Oracle zahlreiche SQL-ähnliche Sprachkonstrukte an. Ein Teil dieser Befehle dient dabei auch der Verbesserung der Performance der Datenbank. Einige sind in Tab. 38 aufgelistet.

Mit diesen Befehlen werden Datenbanken der erforderlichen Größe zusammen mit Logdateien und gegebenenfalls Archivlogs (siehe Kapitel 7) angelegt. Jede Datenbank kann aus einer beliebigen Anzahl *Cluster* bestehen. Dies sind physische Einheiten, um die Performance der Datenbank zu erhöhen. Ein *Clu-*

ster enthält wiederum Relationen, die dann alle physisch auf Platte nebeneinander angelegt werden. Dies beschleunigt die interne Suche auf Daten, wenn sehr häufig nur innerhalb eines *Clusters* zugegriffen wird. Der *Befehl Create Schema Authorization* entspricht dem SQL-Befehl *Create Schema* und dient zur Definition logischer Einheiten innerhalb einer Datenbank. Darüberhinaus bietet Oracle weitere interne Einheiten, etwa *Tablespaces* an. Sie sind die Einheit für Recovery und Backup. Sie stellen einen physischen Bezug zu den gespeicherten Daten auf Magnetplatte her. Mit entsprechenden Einstellungen kann dadurch das Datenbankzugriffssystem in seiner Leistungsfähigkeit optimiert werden.

Tab. 38 Befehle in Oracle zum Arbeiten mit Datenbanken

CREATE DATABASE [<Dbname>] ...	Legt eine neue Datenbank inklusive Logdateien an
ALTER DATABASE [<DBname>] ...	Ändert Datenbankeinstellungen
CREATE CLUSTER <Clustername> ...	Erzeugt einen neuen Cluster
DROP CLUSTER <Clustername>	Löscht einen Cluster
CREATE SCHEMA AUTHORIZATION <Schemaname> ...	Erzeugt ein Schema
CREATE TABLESPACE <Tablespace-Name> ...	Erzeugt einen physischen Bereich zum Speichern der Tabellen

MS-Access besitzt eine leistungsfähige, fensterorientierte Datenbank und ermöglicht die Erstellung einer Datenbank völlig ohne SQL-Befehle. Es stehen Assistenten zur Verfügung, die den kompletten Aufbau einer Datenbank übernehmen. Natürlich können wir in MS-Access aber auch mit SQL-Befehlen arbeiten, denn auch MS-Access deckt die Norm SQL-1+ vollständig ab. Tabellen- und Spaltenbedingungen im *Create-Table*-Befehl sind definiert. Die Angabe von Fremdschlüsseleigenschaften wird allerdings nur in der Oberfläche selbst unterstützt, nicht in SQL-Befehlen. Außerdem müssen Bedingungsnamen in SQL-Befehlen immer angegeben werden.

Das Erzeugen einer Datenbank, oder auch nur das Erzeugen eines Schemas mit Hilfe von SQL-Befehlen wird in MS-Access nicht unterstützt. Dafür bietet MS-Access wertvolle grafische Werkzeuge. Beispielsweise werden mittels des Menübefehls Extras/Beziehungen alle Relationen zusammen mit ihren Beziehungen (also ein Entity-Relationship-Modell) automatisch angezeigt (siehe Abb. 24). In diesem ERM lassen sich beispielsweise die Fremdschlüsseleigenschaften weiter festlegen. Wenn diese Eigenschaften auch nicht mittels SQL-

Befehlen spezifiziert werden können, so sind zumindestens in diesem ERM die Einstellungen *On Delete Cascade* und *On Update Cascade* möglich.

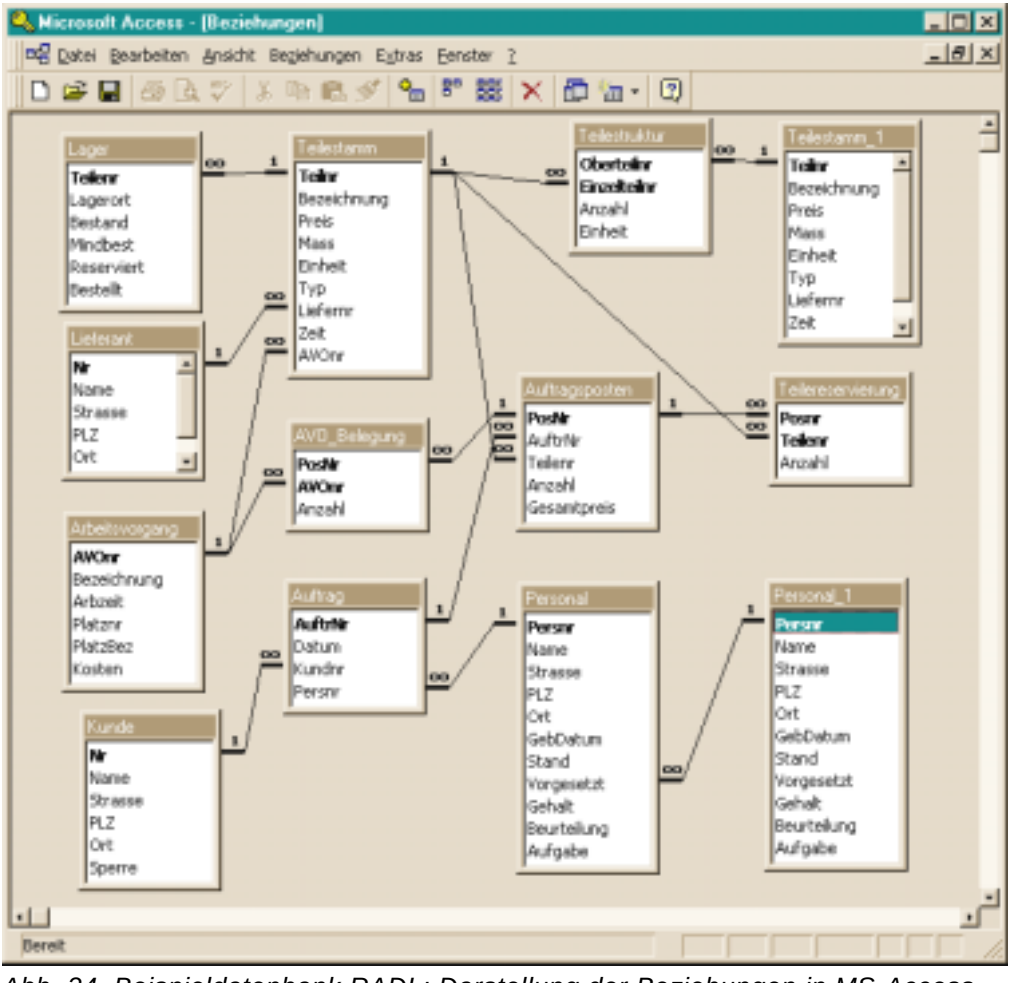


Abb. 24 Beispieldatenbank RADL: Darstellung der Beziehungen in MS-Access

6.6 Systemtabellen in SQL und Oracle

Datenbanken enthalten viele Relationen, diese besitzen Attribute, Primär- und Fremdschlüssel, weitere Tabellen- und Spaltenbedingungen, Gebietsdefinitionen und Zugriffsrechte. All diese Informationen muss das Datenbanksystem verwalten. Es ist nur naheliegend, dass das Datenbankverwaltungssystem

diese und weitere Informationen in ihm vertrauten Strukturen, also Relationen, ablegt.

SQL-1 überließ die Verwaltung der Datenbank noch den einzelnen Herstellern. Ab SQL-2 wird in jedem Katalog automatisch ein Schema namens *Information_Schema* angelegt. In diesem Schema werden die gesamten Verwaltungsinformationen eines Katalogs in Form von Relationen angelegt. Einige dieser Relationen zählt Tab. 39 auf.

Tab. 39 Relationen aus INFORMATION_SCHEMA

Relation	Enthält
SCHEMATA	alle Schemata
DOMAINS	alle Gebiete
TABLES	alle Basisrelationen
VIEWS	alle Sichten
VIEW_TABLE_USAGE	alle Abhängigkeiten der Sichten von Relationen
VIEW_COLUMN_USAGE	alle Abhängigkeiten der Sichten von Spalten
COLUMNS	alle Spaltennamen aller Basisrelationen
TABLE_PRIVILEGES	alle Zugriffsrechte auf Relationen
COLUMN_PRIVILEGES	alle Zugriffsrechte auf Spalten aller Relationen
DOMAIN_CONSTRAINTS	alle Gebietsbedingungen für alle Gebiete
TABLE_CONSTRAINTS	alle Tabellenbedingungen aller Relationen
REFERENTIAL_CONSTRAINTS	alle referentiellen Bedingungen
CHECK_CONSTRAINTS	alle Check-Bedingungen aller Relationen
ASSERTIONS	alle allgemeinen Bedingungen

Die Liste in der Tabelle ist nicht vollständig. Sie soll nur einen Überblick geben, dass SQL die Datenbankstrukturen komplett und übersichtlich verwaltet. Übrigens handelt es sich bei den in Tab. 39 angegebenen Relationen durchwegs um Sichten. Diese sorgen dafür, dass der Benutzer nur die Informationen erhält, für die er oder der allgemeine Benutzer *Public* Zugriffsrechte besitzt. Weiter werden nur Daten des jeweiligen Katalogs ausgegeben.

Auch Oracle speichert seine Systemdaten in Relationen. Neben einigen Basisrelationen existieren eine Fülle von Sichten und Synonymen (insgesamt über 100!). Eine Kurzbeschreibung all dieser Relationen finden wir in der Relation *Dictionary*. Wir können diese mittels eines *Select*-Befehls abfragen und werden dann feststellen, dass viele Sichten mit „USER_“, andere wiederum mit „DBA_“, „SYSTEM_“ oder „ALL_“ beginnen. Bei Ersteren handelt es sich um Relationen und Sichten, die Informationen zu von Benutzern angelegten Datenbankstrukturen enthalten (siehe Tab. 40). Die anderen drei

Gruppen beinhalten Strukturdefinitionen, die den Datenbankadministrator bzw. das System bzw. alle Angaben betreffen. Sie sind in der Regel nur für den Administrator zugreifbar.

Tab. 40 Ausgewählte Systemtabellen in Oracle

Relation	enthält
DICTIONARY	Zusammenfassung über alle Systemrelationen
USER_TABLES	alle Relationen des Benutzers
USER_TAB_COLUMNS	alle Attribute aller Relationen des Benutzers
USER_VIEWS	alle Sichten des Benutzers
USER_CONSTRAINTS	alle Spalten- und Tabellenbedingungen
USER_CONS_COLUMNS	alle Attribute, die Spalten- und Tabellenbedingungen enthalten
USER_INDEXES	alle Indexe in Relationen des Benutzers
USER_IND_COLUMNS	alle Attribute, die Indexe besitzen
USER_TAB_PRIVS	alle Privilegien in Bezug auf Relationen
USER_COL_PRIVS	alle Privilegien in Bezug auf Attribute
USER_TABLESPACES	alle Tablespaces des Benutzers

Beispielsweise finden wir Informationen zu allen definierten Benutzerrelationen in der Relation *User_Tables*. Weiter gibt es Relationen, die Informationen zu allen definierten Spalten, Indexen, Sichten, Zugriffsrechten, Check- und Integritätsbedingungen enthalten. Kurz gesagt: Alle Beschreibungssprachbefehle führen zu Einträgen in Systemrelationen, und diese Einträge sind lesbar und damit jederzeit mittels eines *Select*-Befehls abfragbar.

Zuletzt sei noch darauf hingewiesen, dass MS-Access seinen Benutzern eine übersichtliche grafische Oberfläche anbietet. Systemtabellen, auf die mittels SQL-Befehlen zugegriffen werden könnte, existieren daher in MS-Access nicht.

6.7 Zusammenfassung

Wir haben in diesem Kapitel einige wichtige DDL-Befehle vorgestellt. In der Praxis werden diese Befehle recht selten verwendet; denn wer implementiert schon alle paar Tage eine neue Datenbank? Im Bedarfsfall empfiehlt sich allerdings das genaue Studieren der Syntax und der Arbeitsweise dieser Befehle im Handbuch des Datenbankherstellers. Es sollten alle Möglichkeiten

bezüglich Integrität ausgereizt werden. Nur so sind später optimale Anwendungen möglich.

Wichtig ist bei großen Datenbanken die Aufteilung in zusammengehörige Gruppen (in SQL: *Schema*), und innerhalb dieser Gruppen die Erstellung der Basisrelationen. Ergänzt werden diese Definitionen durch das Hinzufügen geeigneter Indexe und Sichten. Nochmals sei dringend darauf hingewiesen, Integritätsbedingungen bereits beim Erzeugen der Relationen mit anzugeben. Nur so kann das Datenbankverwaltungssystem im Betrieb eventuelle Integritätsverletzungen automatisch erkennen.

Das Erzeugen einer Datenbank wird durch das Gewähren von Zugriffsrechten und das Hinzufügen von semantischen Integritätsbedingungen abgerundet. Wir verweisen diesbezüglich auf Kapitel 8.

6.8 Übungsaufgaben

1) Schreiben Sie alle *Create-Table*-Befehle zum Erzeugen der Beispieldatenbank *Radl* aus Anhang A. Geben Sie alle Entitäts- und Referenz-Entitäten in Form von Tabellen- und Spaltenbedingungen an. Ergänzen Sie diese Angaben nach Belieben durch weitere sinnvolle Integritätsbedingungen (*Unique*, *Not Null*, *Check*-Bedingung).
2) Schreiben Sie einen Befehl, der zur Relation *Auftragsposten* aus der Beispieldatenbank *Radl* das Attribut *Einzelpreis* hinzufügt. Füllen Sie dieses Attribut mit Daten, ermittelt aus den Attributen *Anzahl* und *Gesamtpreis* automatisch auf. In welcher Normalform befindet sich die Relation Auftragsposten jetzt?
3) Versehen Sie das Attribut *Bezeichnung* der Relation *Teilestamm* aus der Beispieldatenbank *Radl* mit einem Index. Ist ein eindeutiger Index mittels des Bezeichners *Unique* sinnvoll?
4) In einem großen Dienstleistungsunternehmen werden zentral Aufträge entgegengenommen und an Sachbearbeiter weitergeleitet, deren Einsatzort in der Nähe des Kunden liegt. Dazu existiere in der Datenbank eine Relation *Personal*, die die benötigten Daten der Sachbearbeiter speichert, insbesondere auch den Namen und den Einsatzort. Wegen einer großen Anzahl von Sachbearbeitern ist die Suche nach möglichen Sachbearbeitern in der Nähe des Kunden zeitaufwendig. Schreiben Sie einen Index, um die alphabetische Suche nach Sachbearbeitern an bestimmten Einsatzorten zu beschleunigen.
5) Erzeugen Sie in der Beispieldatenbank *Radl* eine Sicht *VPers*, die der Relation *Personal* ohne die Attribute *Gehalt* und *Beurteilung* entspricht. Weiter sind in dieser Sicht nur die Personen aufzunehmen, denen ein Vorgesetzter zugeordnet ist. Dürfen in dieser Sicht auch Daten geändert werden?

6) In SQL-1 sind Tabellenbedingungen im *Create-Table*-Befehl nicht vorgesehen. Schreiben Sie deshalb als Ersatz eine Sicht *VTeilestamm*, die der Relation *Teilestamm* der Beispieldatenbank *Radl* entspricht, aber gleichzeitig im Attribut *Einheit* nur die Angaben „ST“ und „CM“ und im Attribut *Typ* nur die Angaben „E“, „Z“ und „F“ erlaubt. Andere Angaben werden abgewiesen.
7) Die Relation *Auftragsposten* enthält aus Redundanzgründen nur den Gesamtpreis jedes einzelnen Auftragspostens. Schreiben Sie daher eine Sicht *VAuftragsposten*, die alle Daten der Relation *Auftragsposten* enthält und zusätzlich ein Attribut Einzelpreis. Ist diese Sicht änderbar?
8) In SQL gibt es keinen *Alter-Schema*-Befehl. Überlegen Sie, wie in ein existierendes Schema weitere Schemaelemente aufgenommen werden können. Zeigen Sie dies an einem Beispiel.

7 Concurrency und Recovery

Der Aufbau einer Datenbank, insbesondere der von relationalen Datenbanken, wurde in den letzten Kapiteln ausführlich behandelt. Auch der Zugriff auf diese Datenbanken wurde besprochen und am Beispiel von SQL praktisch geübt. Holen wir uns jetzt aber ins Gedächtnis zurück, dass unser Anforderungskatalog in Kapitel 2 nicht nur aus dem Erstellen und Zugreifen oder dem Verbergen der physischen Struktur bestand. In der Praxis spielen auch die Forderung der Datensicherheit und (im Zeichen der Vernetzung) der Parallelbetrieb eine eminent wichtige Rolle. Über einen Teil dieser Forderungen, die Recovery und die Concurrency, wollen wir in diesem Kapitel und über die Sicherheit und Integrität im nächsten Kapitel sprechen.

Unter Recovery verstehen wir die Rekonstruktion einer Datenbank im Fehlerfall. Aufgabe der Recovery ist es, nach aufgetretenen Problemen verlorengegangene Datenbankinformationen wiederzubeschaffen. Die Konsistenz der Daten muss gewährleistet werden. Die Art der Probleme, wie etwa plötzlicher Rechnerausfall oder Feuer, darf keinen Einfluss auf die Wiederherstellbarkeit der Daten besitzen.

Concurrency ist der englische Begriff für Parallelbetrieb. Gefordert ist hier das Zulassen gleichzeitiger Zugriffe von mehreren Benutzern auf eine Datenbank. Ohne entsprechende Maßnahmen können durch solch gleichzeitige Zugriffe Konsistenzverletzungen und Datenverlust auftreten. Dies ist natürlich zu verhindern, was Synchronisationsmechanismen erfordert.

Zu beiden Themen (Recovery und Concurrency) werden in diesem Kapitel die auftretenden Probleme eingehend geschildert und Lösungen vorgestellt. Wir ziehen im Weiteren die englischen Bezeichnungen vor, da sie sich auch in der deutschsprachigen Literatur durchgesetzt haben.

Vor allem von Kleindatenbankanwendern wird die Bedeutung der Recovery und Concurrency gern unterschätzt. In solch kleinen Datenbanken wird meist eine tägliche Sicherung durchgeführt, und bei Störungen werden die auftretenden Fehler zu Fuß korrigiert, was aber auch hier schon sehr mühsam, aufwendig und fehleranfällig sein kann. Parallelbetrieb wird in solchen Systemen in der Regel überhaupt nicht oder nur unzureichend unterstützt.

Ein ganz anderes Bild ergibt sich auf Großrechnern und mittlerweile auch auf allen mittleren Systemen. Betrachten wir nur Banken- oder Versicherungs-

datenbanken. Die Daten müssen ständig und von überall verfügbar sein, Datenverluste und Inkonsistenzen müssen unter allen nur denkbaren Umständen verhindert werden. Dabei soll dem Endbenutzer und dem Anwendungsprogrammierer der hohe Aufwand verborgen bleiben, den das Datenbankverwaltungssystem leisten muss, um Datenverluste und Inkonsistenzen zu unterbinden und einen reibungslosen Parallelbetrieb zu ermöglichen.

Trotz der scheinbar verschiedenen Zielsetzungen verzahnen sich Recovery und Concurrency sehr. Der Aufwand für die Recovery steigt erheblich im Parallelbetrieb gegenüber der Einplatzlösung, und Concurrency ist ohne eine minimale Recovery nicht vorstellbar. Insbesondere sind beide Begriffe sehr eng mit der Transaktionsverarbeitung verbunden.

7.1 Recovery

Wie bereits erwähnt, befasst sich die Recovery mit dem Wiederherstellen von Daten nach aufgetretenen Fehlern. Um Daten nach einem Fehlerfall (z.B. nach Stromausfall) reproduzieren zu können, müssen wir bereits vorher Vorsorge treffen. Die geeignete Vorsorge hängt von den möglichen Fehlerquellen ab. Wir müssen daher zunächst die möglichen auftretenden Fehler erfassen und klassifizieren. Ganz grob können wir die Fehler in zwei Klassen einteilen, in Hardware- und Softwarefehler. Eine kleine Liste dieser Fehler ist im Folgenden angegeben:

HW-Fehler: Stromausfall, Wackelkontakte, Plattenausfall, Arbeitsspeicherausfall, Netzausfall, Brand (Feuer, Löschwasser).

SW-Fehler: Fehler in der Datenbanksoftware, im Betriebssystem, im Anwendungsprogramm, in der Übertragungssoftware.

Eine gewisse Vorsorge wird ein Datenbankadministrator dadurch treffen, dass er die Hard- und Software auch nach dem Gesichtspunkt der Zuverlässigkeit auswählt. Dies minimiert zwar die Ausfallrate, ganz verhindert wird sie dadurch jedoch nicht.

Die Forderung der Anwender und des Administrators wird im Fehlerfall sein, dass möglichst keine Daten verlorengehen, dass eine Rekonstruktion des letzten Datenbestandes möglich ist, und dass dabei die Datenbank-Integrität erhalten bleibt. Zwecks Rekonstruierbarkeit der Daten wird deshalb in der Praxis mindestens einmal pro Woche eine Komplettsicherung und täglich eine

Differenzsicherung des Datenbestandes durchgeführt. Außerdem wird ein elektronisches Logbuch geführt. Dieses Logbuch vermerkt alle Änderungen in der Datenbank seit der letzten Sicherung.

Die Forderung der Konsistenz bereitet zusätzliche Schwierigkeiten. Ein konsistenzerhaltender Datenbankzugriff (Transaktion) besteht in der Regel aus mehreren Einzelzugriffen auf den Datenbestand. Geschieht ein Fehler gerade inmitten einer solchen Transaktion, so ist normalerweise die Konsistenz des Datenbestandes nicht mehr gegeben. Bei der Rekonstruktion einer zerstörten oder inkonsistenten Datenbank muss daher unbedingt auf nicht korrekt abgeschlossene Transaktionen geachtet werden.

Trotz obiger Forderungen für den Fehlerfall soll die Datenbank performant ablaufen. Dies hat zur Folge, dass möglichst viele Daten im Arbeitsspeicher gehalten werden, um die im Vergleich zur Rechnergeschwindigkeit sehr lange dauernden Ein- und Ausgabezugriffe zu minimieren. Ein Stromausfall zeigt aber gerade dann besonders unangenehme Folgen. Bevor wir uns aber diesen komplizierten Fall genauer ansehen, beginnen wir mit den Transaktionen:

Betrachten wir dazu das Beispiel einer Buchung eines Fluges von München nach San Francisco. Womöglich fliegt der Reisende über Frankfurt und New York. Er bucht dann zusammen drei Einzelflüge. Natürlich soll jede einzelne Buchung nur erfolgen, wenn auch die beiden anderen gelingen. Die drei Buchungen müssen demnach atomar erfolgen.

Eine Datenbank unterstützt die Gesamtbuchung einer solch komplexen zusammengesetzten Buchung mittels der Unterstützung konsistenzerhaltender Manipulationen, den Transaktionen. In SQL lässt sich beispielsweise jedes Anwendungsprogramm in einzelne Transaktionen zerlegen. Diese Zerlegung geschieht mit dem Befehl *Commit Work* oder abkürzend einfach *Commit*. Jedes solche *Commit Work* beendet die gerade laufende Transaktion, folgende Aktionen gehören bereits zur nächsten Transaktion. Unsere Buchung besteht demnach aus drei einzelnen Update-Befehlen, die die Buchungen vornehmen, gefolgt von einem *Commit Work*:

```
UPDATE Flug  SET ...   ;     (Flug München - Frankfurt)
UPDATE Flug  SET ...   ;     (Flug Frankfurt - New York)
UPDATE Flug  SET ...   ;     (Flug New York - San Francisco)
COMMIT WORK ;                (Transaktion beendet, Buchung endgültig erfolgt)
```

Ist nun im Fehlerfall eine Transaktion noch nicht abgeschlossen, so muss diese komplett zurückgesetzt werden. Ein Rechnerabsturz während des Ablaufs obigen Programmausschnitts erfordert während der Rekonstruktion die

Rücknahme aller bereits durchgeführten Änderungen. Umgekehrt muss bei einer Rekonstruktion jedes Datum erhalten bleiben, das durch eine (mit *Commit Work*) beendete Transaktion geändert wurde. Erst dann ist die Zuverlässigkeit des Datenbestandes einer Datenbank gewährleistet.

Nun kommt es auch im Normalfall gelegentlich vor, zum Beispiel bei der obigen Buchung eines Fluges, dass dem Reisenden die angebotenen Flug- und Wartezeiten missfallen. Sicherheitshalber wurden die Flüge aber bereits vorreserviert. Doch auch hier gibt es in Datenbanken keine Probleme, denn alle Aktionen innerhalb von nicht abgeschlossenen Transaktionen gelten als noch nicht endgültig festgelegt. Sie können jederzeit widerrufen werden. Obiger Programmausschnitt sieht dann in der Programmiersprache C++ unter Verwendung von eingebetteten SQL-Befehlen (Voranstellen des Bezeichners *Exec Sql* vor SQL-Befehlen, siehe auch Kapitel 9) wie folgt aus:

```
EXEC SQL UPDATE Flug  SET ...   ;          // Flug München - Frankfurt
EXEC SQL UPDATE Flug  SET ...   ;          // Flug Frankfurt - New York
EXEC SQL UPDATE Flug  SET ...   ;          // Flug New York - San Francisco
cout << " Wollen Sie den Gesamtflug fest buchen? (j/n): " ;
getchar (zeichen);
if ( zeichen == 'j' )
{       EXEC SQL COMMIT WORK;              // Transaktion erfolgreich beendet
        cout << "Der Flug ist gebucht!\n" ;
}
else
{       EXEC SQL ROLLBACK WORK;            // Transaktion wird zurueckgesetzt
        cout << "Der Flug wurde storniert!\n" ;
}
```

Neu in diesem Programmausschnitt ist der SQL-Befehl *Rollback Work* oder kurz *Rollback*. Mit diesem Befehl werden alle seit dem letzten *Commit Work* durchgeführten Änderungen in der Datenbank automatisch zurückgesetzt. Um dieses Rücksetzen immer zu gewährleisten, muss das Datenbankverwaltungssystem deshalb alle Änderungen innerhalb einer Transaktion in einem Logbuch protokollieren.

Das Mitführen eines solchen Logbuchs, kurz Log genannt, ist für die Sicherheit der Daten einer Datenbank extrem wichtig. Wir müssen beispielsweise beachten, dass ein Rechnerabsturz an jeder beliebigen Stelle eines Programms erfolgen kann, wobei im Mehrbenutzerbetrieb sogar viele Programme gleichzeitig auf eine Datenbank zugreifen. Das Datenbankverwaltungssystem wird mit Hilfe der Logeinträge dafür sorgen, dass im Fehlerfall alle verlorengegangenen Daten wiederhergestellt werden, für die bereits ein *Commit Work*

erfolgte. Andererseits sind Änderungen, verursacht durch noch nicht abgeschlossene Transaktionen, zurückzusetzen.

7.1.1 Recovery und Logdatei

Wir interessieren uns nun für Einzelheiten zum Ablauf der Transaktionen im Zusammenhang mit der Wiederherstellung von Datenbeständen. Um eine Wiederherstellung garantieren zu können, müssen geänderte Daten sowohl in der Datenbank als auch im Log gespeichert werden. Eines dürfte schon klar sein: Die für die Recovery erforderlichen Logdateien zum Aufnehmen der Logeinträge stehen aus Sicherheitsgründen ausschließlich auf nichtflüchtigen Medien. Weiter dürfte offensichtlich sein, dass das Abspeichern der Logdaten nicht auf dem selben Medium erfolgen sollte, auf dem bereits die Datenbank gespeichert ist. Ein Ausfall dieses Mediums, in der Regel eine Magnetplatte, würde die Logdatei und die Datenbankdaten gleichzeitig zerstören, eine Recovery wäre nicht mehr gewährleistet.

Wir wollen nun ausführlich kennenlernen, wann welche Daten in der Logdatei gespeichert werden müssen. Betrachten wir zunächst die Voraussetzungen, die ganz allgemein in einem Datenbankbetrieb gegeben sind:

- Die Datenbank ist auf externen nichtflüchtigen Medien angelegt und direkt vom Server aus zugreifbar.
- Die Logdatei steht auf einem weiteren externen nichtflüchtigen Medium und ist ebenfalls vom Server aus direkt zugreifbar.
- Zur Entlastung des Ein-/Ausgabeverkehrs und zur Beschleunigung werden Daten der Datenbank in einem internen Arbeitsspeicherbereich (Datenbank-Puffer, englisch: Cache) zwischengespeichert.
- Beim Lesezugriff wird grundsätzlich vom Datenbank-Puffer gelesen. Ist das gesuchte Datum noch nicht im Puffer, so wird es zunächst von der Datenbank in den Puffer geholt.
- Auch das Schreiben erfolgt grundsätzlich in den Datenbank-Puffer. Zu einem günstigen späteren Zeitpunkt erfolgt die Aktualisierung der Datenbank. Ein forciertes Schreiben in die Datenbank bei Transaktionsende findet nicht statt.
- Ist der Datenbank-Puffer voll gefüllt, so werden die am längsten nicht mehr benutzten Daten freigegeben. Geänderte Daten werden auf jeden Fall vorher in die Datenbank zurückgeschrieben.

Die Informationen, welche Daten der Datenbank im Puffer stehen, und welche geändert wurden, heißen Metadaten und werden in einem eigenen Bereich im Arbeitsspeicher verwaltet. Zu den Metadaten zählen auch die Daten über alle offenen Transaktionen und über die Synchronisation beim Puffer- und Logdateizugriff.

Die Datenbankpufferung im Arbeitsspeicher reduziert sowohl die Laufzeiten als auch den Datentransfer zwischen Datenbank und Arbeitsspeicher erheblich. Die Größe eines Datenbank-Puffers kann dabei mehrere Gigabytes betragen. Die Verwaltung eines solch komplexen Datenbankbetriebs übernimmt das Datenbankverwaltungssystem. Eine kleine Übersicht über die Arbeitsweise einer Datenbank mit Datenbankpufferung und Sicherung in die Logdatei vermittelt Abb. 25.

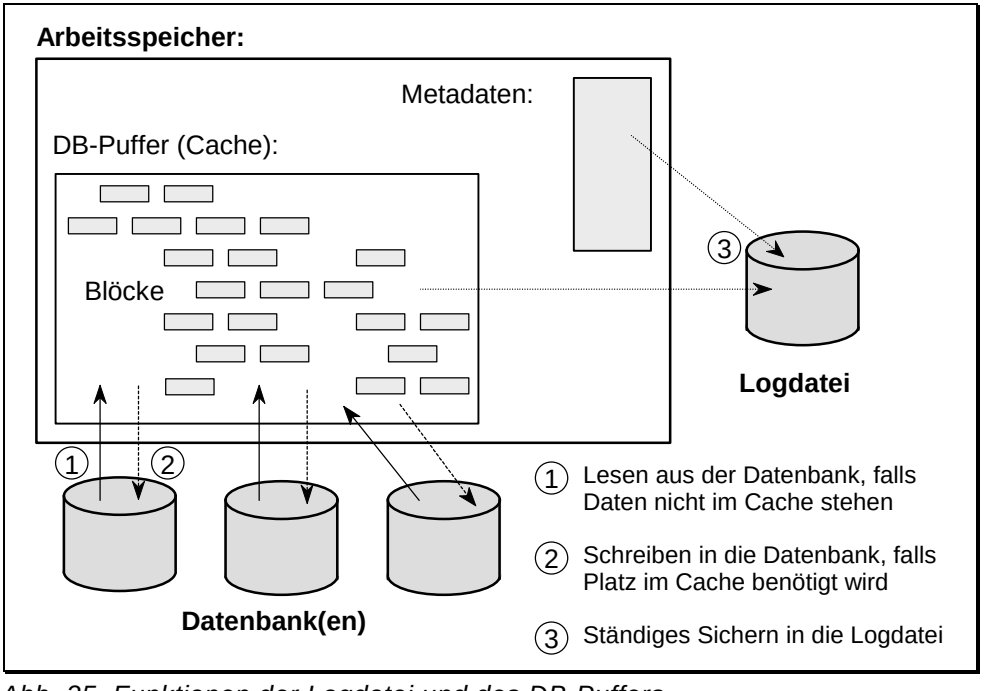


Abb. 25 Funktionen der Logdatei und des DB-Puffers

Die zugrundeliegende Idee ist, Daten direkt und ausschließlich im Datenbankpuffer zu bearbeiten. Um ein Rücksetzen einer Transaktion jederzeit zu garantieren, werden die zu ändernden Daten zunächst in der Logdatei persistent, d. h. nicht flüchtig, gespeichert. Dann erfolgt der Update der Daten, und auch diese neuen Daten werden wieder in die Logdatei geschrieben. Die genaue Abarbeitungsreihenfolge können wir dabei aus Tab. 41 entnehmen.

Tab. 41 Transaktionsbetrieb mit Log und Arbeitsspeicherpufferung

Lesen der Daten	Die Daten werden von Platte eingelesen, falls sie sich nicht bereits im Puffer des Arbeitsspeichers befinden.
Merken der bisherigen Daten	Die zu ändernden Daten werden in die Logdatei geschrieben (Before Image).
Ändern der Daten	Ändern (*Update*, *Delete*, *Insert*) der Daten im Arbeitsspeicher, Sperren dieser Einträge für andere Benutzer.
Merken der geänderten Daten	Die geänderten Daten werden in die Logdatei geschrieben (After-Image).
. . .	Obige vier Schritte können sich innerhalb einer Transaktion mehrmals wiederholen.
Transaktionsende mit COMMIT bzw. ROLLBACK	COMMIT: Schreiben eventuell noch nicht geschriebener Before- und After-Images und benötigter Metadaten zu dieser Transaktion in die Logdatei. Transaktionsende in der Logdatei vermerken. Sperren freigeben. ROLLBACK: Rücksetzen der Metadaten der Transaktion. Geänderte Daten im Arbeitsspeicher mittels der Before-Images restaurieren. Alle geänderten Daten, die bereits in die Datenbank geschrieben wurden, werden für ungültig erklärt. Sperren freigeben.
Änderungen speichern	Die geänderten Daten werden asynchron (unabhängig vom Transaktionsbetrieb) in die Datenbank geschrieben.

Mit dieser Abarbeitungsreihenfolge ist zu jeder Zeit gewährleistet, dass alle Datenänderungen nachvollzogen werden können. Wir wollen dies detailliert nachprüfen:

Beginnen wir mit dem Fall, dass eine Transaktion mittels eines *Rollback Work* zurückgesetzt wird. Mittels des Before-Images, dem Merken des Inhalts vor einer Änderung in der Datenbank, können jederzeit die ursprünglichen Daten reproduziert werden. Sollten die Daten bereits vorher in die Datenbank geschrieben und eventuell sogar aus dem Arbeitsspeicherpuffer entfernt worden sein, so genügt ein erneutes Lesen der geänderten Daten in den Puffer, ein Rücksetzen mittels des Before-Images und ein für ungültig Erklären des betreffenden Datenbankinhalts.

Der zweite Fall betrifft das Verlorengehen von Datenbankdaten. Hier hilft das Merken der geänderten Daten mittels des After-Image weiter. Da beim Ende einer Transaktion ein entsprechender Vermerk in die Logdatei geschrieben wird, kann bei einem Rechnerabsturz oder einem Plattenausfall jederzeit erkannt werden, welche Änderungen zu einer bereits beendeten Transaktion gehören. Im schlimmsten Fall, einem totalen Datenverlust auf der Magnet-

platte, muss die letzte Sicherung eingespielt werden. Und anschließend sind alle Transaktionen mittels der Logeinträge nachzuvollziehen. Änderungen, die zu keiner bereits beendeten Transaktion gehören, werden natürlich nicht mit ausgeführt. Wir erhalten somit den neuesten konsistenten Stand der Datenbank.

Wird die Logdatei nicht auf demselben Medium angelegt, auf dem sich auch die Datenbank befindet, so ist es äußerst unwahrscheinlich, dass beide Datenbestände, die Logdatei und die Datenbank, gleichzeitig zerstört werden. Wollen wir uns aber auch dagegen absichern, so können wir zusätzlich die Datenbank oder die Logdatei doppelt anlegen und verwalten, natürlich wieder auf verschiedenen Medien, wenn möglich auch in verschiedenen Räumen.

Genaugenommen müssen neben den Before- und After-Images auch viele Metadaten in die Logdatei geschrieben werden. Obwohl der Datenverkehr zwischen Arbeitsspeicher und Datenbank dank Pufferung erheblich reduziert werden kann, geht dies zu Lasten des Datentransfers in die Logdatei. In der Logdatei müssen aber nur die tatsächlichen Änderungen, meist wenige Bytes, vermerkt werden. Da auch die Metadaten meist nicht umfangreich sind, genügt häufig pro Transaktion das Schreiben eines einzigen Blocks (mit allen Änderungsinformationen) in die Logdatei! Das Schreiben in die Logdatei erfolgt daher meist verzögert erst zum Transaktionsende. Dies ist legitim, da ja erst dann die Daten gesichert sein müssen. Es muss allerdings sichergestellt sein, dass vor dem Schreiben von Änderungen in die Datenbank das entsprechende Before-Image in der Logdatei gespeichert wurde. Zur Performancesteigerung trägt darüberhinaus bei, dass in die Logdatei sequentiell geschrieben wird. Positionierzeiten des Schreibkopfes fallen daher nicht an.

Natürlich gibt es weitere Optimierungen, die jeder Datenbankhersteller je nach Philosophie und Bedarf einsetzt. Das Grundprinzip aus Tab. 41 ist aber nie in Frage gestellt.

Die Logdatei wird mit der Zeit sehr umfangreich, eine Sicherung der Datenbank, zumindest jedoch eine Differenzsicherung, sollte daher täglich durchgeführt werden. Anschließend kann der Inhalt der Logdatei komplett gelöscht werden. Bei sehr großen Datenbanken wird aus Sicherheitsgründen und wegen der immensen Datenmengen die Logdatei in zwei Teile zerlegt, in eine „normale“ Logdatei und ein Archivlog. Es werden dabei laufend ältere Einträge der Logdatei in das Archivlog, meist eine Banddatei, geschrieben. Beschriebene Magnetbänder oder Magnetbandkassetten werden sofort sicher aufbewahrt. Dass dies zusätzlichen Software-Aufwand erfordert ist klar, wir wollen auf

diese Besonderheiten aber nicht weiter eingehen. Einen noch nicht behandelten wichtigen Punkt dürfen wir aber nicht übergehen, das Anlegen von Checkpoints:

Aus dem Datenbankpuffer werden nur diejenigen Blöcke der Datenbank verdrängt, auf die seit längerem nicht mehr zugegriffen wurde. Einige Daten der Datenbank werden jedoch laufend gelesen oder geändert, so dass diese nie in die Datenbank zurückgeschrieben werden. Diese Blö*ck*e im Datenbankpuffer heißen Hot Spots. Aus zwei Gründen sind solche Hot Spots nicht erwünscht. Erstens enthält die Datenbank auf dem externen sicheren Medium einen eventuell sehr alten Stand, zweitens werden aus Performance-Gründen die Metadaten gerne sequentiell geschrieben und zyklisch wieder überschrieben. Da aber Metadaten zu allen Blöcken des Puffers geführt und gemerkt werden müssen, solange diese nicht den aktuellen Stand auf der externen Datenbank wiedergeben, können Metadaten zu Hot Spots nicht einfach überschrieben werden. Dies ließe sich zwar durch komplexere Algorithmen ändern, doch der erste Grund führt dazu, dass im Fehlerfall die gesamte Datenbanksitzung seit der letzten Sicherung nachvollzogen werden müsste. Schließlich wissen wir ja im Allgemeinen nicht, welche Daten von welchen Transaktionen noch nicht auf Platte gesichert wurden. Dieses Nachvollziehen der Datenbanksitzung seit der letzten Sicherung kann Stunden dauern. Im Fehlerfall wäre die Datenbank für diese Zeit blockiert, was in der Praxis nicht akzeptabel ist.

Der einzige vernünftige Ausweg aus diesem Dilemma ist, gelegentlich alle Blöcke des Datenbankpuffers zwangsweise in die Datenbank zu schreiben. Diese Zeitpunkte heißen Checkpoints, und werden in regelmäßigen Abständen, meist im Minutenbereich, vom Datenbankverwaltungssystem angestoßen. Diese Checkpoints verursachen einen sehr hohen Ausgabeverkehr und bremsen dadurch alle aktuell ablaufenden Transaktionen merkbar. Dies wird jedoch in Kauf genommen, da dies eine eventuell erforderliche Recovery erheblich erleichtert und beschleunigt. Wir kommen gleich darauf zurück. Es sei noch darauf hingewiesen, dass die Datenbank nach einem Checkpoint nicht notwendigerweise konsistent ist, da beim Checkpoint alle geänderten Daten gespeichert werden, also auch solche, die von noch nicht abgeschlossenen Transaktionen stammen. Natürlich wird jeder Checkpoint auch in der Logdatei vermerkt. Ebenso werden zunächst alle Before-Images in die Logdatei geschrieben, bevor mit dem Schreiben in die Datenbank begonnen wird.

7.1.2 Recovery und Checkpoints

Vollziehen wir jetzt die Recovery unter Einbeziehung von Checkpoints nach. Gehen wir dazu davon aus, dass hinreichend Vorsorge im Transaktionsbetrieb getroffen wurde: Wir arbeiten aus Performancegründen mit Datenbank-Pufferung, haben eine Logdatei im Einsatz und leeren regelmäßig unsere Puffer mittels Checkpoints. Selbstverständlich wird auch eine tägliche Differenzsicherung des Datenbestandes und eine wöchentliche Gesamtsicherung vorgenommen. Wir sind demnach gegen Fehlerfälle gewappnet. Die erforderlichen Reaktionen auf die verschiedenen auftretenden Fehler wollen wir jetzt etwas genauer betrachten. Tab. 42 zeigt die Einteilung der möglichen Fehler in drei Klassen, den lokalen Fehlern, dem Softcrash und dem Hardcrash.

Tab. 42 Mögliche Fehler im Transaktionsbetrieb

Lokaler Fehler	Nur eine einzelne Transaktion ist betroffen. Meist liegt ein Softwarefehler vor, etwa Division durch Null oder Bereichsgrenzenüberschreitungen.
Softcrash	Ausfall der Software im größeren Umfang bis hin zum Stromausfall, so dass Daten im Arbeitsspeicher ausgefallen oder ungültig sind: Viele oder sogar alle Transaktionen sind betroffen.
Hardcrash	Ein Hardwareteil fällt aus, etwa eine Magnetplatte.

Auf jeden dieser Fehler reagiert das Datenbankverwaltungssystem angemessen. Bei größeren Fehlern entscheidet der Datenbankadministrator über die einzuleitenden Maßnahmen:

- **Lokaler Fehler:** Das Datenbankverwaltungssystem setzt alle Änderungen der Transaktion, die den Fehler verursachte, zurück. Dies geschieht gegebenenfalls unter Zuhilfenahme der Einträge in der Logdatei. In der Regel werden die gültigen Before-Images sogar noch im Arbeitsspeicher stehen. Alle Sperren der Transaktion werden zurückgesetzt. Es erfolgt eine Meldung an den Datenbankadministrator und, soweit bekannt, auch eine Mitteilung an den Benutzer, der die Transaktion startete.
- **Hardcrash**: Bei einem Plattenausfall stoppt der Datenbankadministrator den Datenbankbetrieb. Alle noch nicht beendeten Transaktionen werden vom Datenbankverwaltungssystem zurückgesetzt. Der Datenbankpuffer wird geleert, sofern Daten von nicht ausgefallenen Platten

betroffen sind. Die Logdatei wird aktualisiert, Sperren werden freigegeben. Der Datenbankadministrator sorgt für eine neue Hardware. Der letzte Sicherungsstand wird auf die neue Platte eingespielt. Die Änderungen seit diesem Sicherungsstand werden auf dieser Hardware mittels der Logdatei nachvollzogen.

Ist „nur“ die Logdatei vom Hardcrash betroffen, so kann der Datenbankadministrator entscheiden, im unsicheren Betrieb weiterzuarbeiten. Die bessere Entscheidung ist, den Datenbankbetrieb sofort anzuhalten, eine Differenzsicherung vorzunehmen und mit neuer Hardware den Betrieb fortzusetzen. Das Anhalten des Datenbankbetriebes wird erreicht, indem neue Transaktionen nicht mehr zugelassen werden, da das Zurücksetzen noch offener Transaktionen wegen der mangelhaften Logdatei eventuell nicht mehr möglich ist. Anschließend werden alle Sperren freigegeben, und der Datenbankpuffer wird geleert.

Sobald der Arbeitsspeicher vom Hardcrash betroffen ist, ist eine direkte Rekonstruktion nicht mehr möglich. Es muss wie im folgenden Softcrash-Fall vorgegangen werden.

- **Softcrash**: Sobald Daten im Arbeitsspeicher verloren gehen, kann die gesamte Datenbank inkonsistent sein. Denken wir nur an Verwaltungsdaten des Datenbankpuffers oder an Metadaten (Sperren) oder an noch nicht gesicherte Before- oder After-Images. Ist der Datenverlust lokalisierbar, und sind nur wenige Transaktionen davon betroffen, so liegt ein lokaler Fehler vor. Jede einzelne Transaktion kann entsprechend zurückgesetzt werden. Andernfalls, etwa bei Stromausfall, ist ein Rücksetzen noch offener Transaktionen nicht mehr möglich. Da die Datenbank auf der Magnetplatte vom Softwareausfall zwar nicht betroffen ist, diese jedoch weder aktuell noch notwendigerweise konsistent ist, muss die Datenbank aktualisiert werden, beginnend beim letzten Checkpoint.

  An Hand der Einträge in der Logdatei kann nachvollzogen werden, welche Transaktionen seit dem letzten Checkpoint gestartet und beendet wurden. In Abb. 26 sind die fünf möglichen Fälle (*T1* bis *T5*) angegeben, in der sich Transaktionen in Bezug auf den letzten Checkpoint und den Softcrash befinden können.

  Die Transaktionen *T1*, die zum Zeitpunkt des Checkpoints bereits abgeschlossen waren, sind komplett in der Datenbank gespeichert, da spätestens zum Zeitpunkt des letzten Checkpoints alle Daten aktualisiert wurden. Bezüglich dieser Transaktionen sind keine Reaktionen erforderlich.

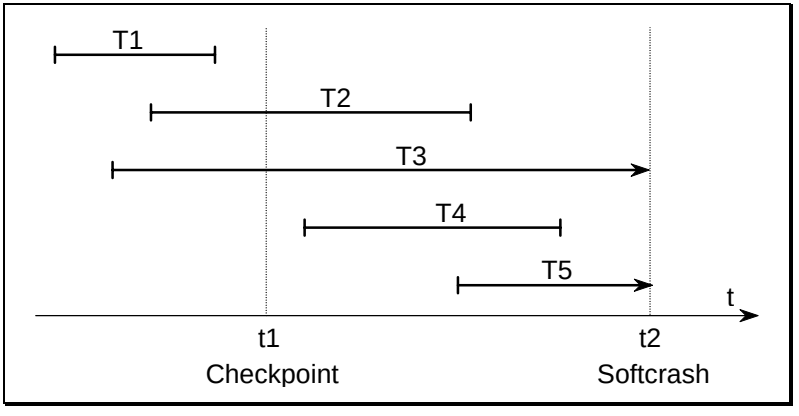


Abb. 26 Zustand der Transaktionen im Softcrash-Fall

Die Transaktionen *T2* und *T4* sind zwar abgeschlossen, es ist aber nicht sichergestellt, dass alle Änderungen auch in die Datenbank geschrieben wurden. Vom Zeitpunkt des letzten Checkpoints bis Transaktionsende wird die Datenbank bezüglich dieser Transaktionen aktualisiert. Die benötigten Informationen werden der Logdatei entnommen. Es ist dabei die Reihenfolge der einzelnen Änderungen aufs genaueste zu beachten. Hilfestellung hierzu leisten die in der Logdatei ebenfalls gespeicherten Metadaten.

Die Transaktionen *T3* und *T5* müssen zurückgesetzt werden, da sie zum Zeitpunkt des Softcrashes noch nicht beendet waren. Es müssen alle Änderungen, die bereits in die Datenbank geschrieben wurden, zurückgenommen werden. Wieder geschieht dies unter Zuhilfenahme der Logdatei.

Wir erkennen die Notwendigkeit einer Logdatei. Sie erst ermöglicht eine komplette Recovery. Checkpoints reduzieren darüberhinaus den Aufwand einer solchen Recovery, da nur Transaktionen, die zum Zeitpunkt des Checkpoints noch nicht beendet waren, restauriert werden müssen. Die restlichen und mit Abstand meisten Transaktionen (vom Typ *T1*) sind von der Recovery nicht betroffen. Eine Wiederherstellung aller Daten wird daher im Fehlerfall meist im Viertelstundenbereich liegen.

Die Häufigkeit eines Checkpoints ist Erfahrungssache und abhängig von der jeweiligen Anwendung. Zu häufige Checkpoints erhöhen den Ausgabeverkehr, sehr seltene Checkpoints führen zu enormen Stoßbelastungen wegen der dann sehr vielen Ausgaben auf Platte, so dass die dann aktuell laufenden Transaktionen erheblich verzögert werden.

7.2 Zwei-Phasen-Commit

Im letzten Abschnitt wurde gezeigt, welche Maßnahmen erforderlich sind, um eine Recovery in einer Datenbank sicherzustellen. In einigen komplexen Anwendungen wird zusätzlich datenbankübergreifend zugegriffen. Dies bedeutet, dass innerhalb einer Transaktion Daten aus mehreren Datenbanken manipuliert werden, wobei diese Datenbanken häufig noch örtlich voneinander getrennt sind. Ein Beispiel ist etwa eine Großbank mit einer Datenbankzentrale, an die mehrere Filialen oder Tochtergesellschaften mit jeweils eigenen lokalen Datenbanken angeschlossen sind. Wir sprechen hier von verteilten Datenbanken. Das Wort *verteilt* bezieht sich in diesem Abschnitt nur auf die Verteilung von Datenbanken im ganzen, jedoch nicht auf die Verteilung innerhalb einer einzelnen Datenbank. Das Aufteilen von Daten innerhalb einer Datenbank auf mehrere Rechner ist zugriffstechnisch recht komplex, und hat daher bis heute in der praktischen Anwendung nur vereinzelt Einzug gefunden. Zu Einzelheiten sei auf Kapitel 11 verwiesen.

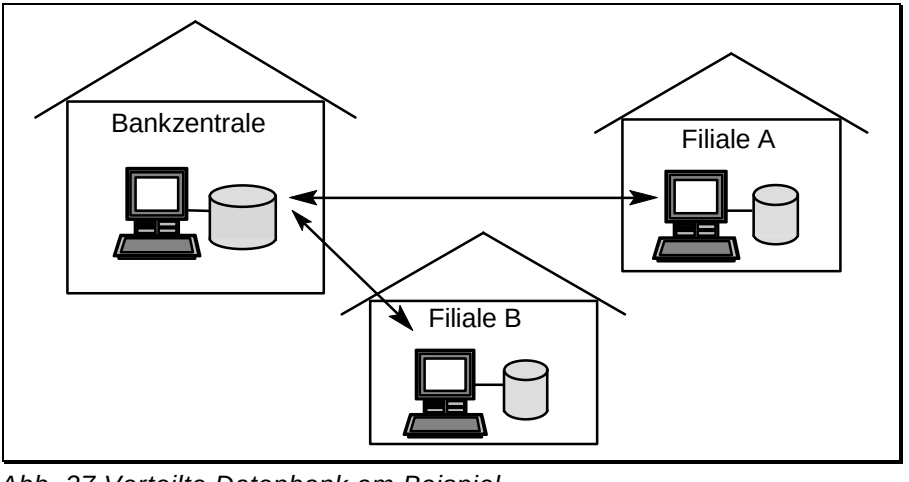


Abb. 27 Verteilte Datenbank am Beispiel

Nehmen nun Transaktionen, die auf mehrere Datenbanken zugreifen, nur auf einer einzigen Datenbank Änderungen vor, kann die Recovery gemäß des letzten Abschnitts durchgeführt werden. Problematisch wird es erst, wenn solche Transaktionen datenbankübergreifend manipulieren. Dies liegt in unserem Beispiel etwa vor, wenn Überweisungen von einer Bank zu einer anderen erfolgen. Um die Konsistenz sicherzustellen, auch über mehrere Datenbanken hinweg, müssen wir fordern, dass globale Transaktionen erst dann als abge-

schlossen gelten, wenn die lokalen Transaktionsabschnitte in allen zugreifenden Rechnern beendet sind.

Es darf demnach nicht passieren, dass beim bargeldlosen Geldtransfer von einem Geldinstitut zum anderen die Transaktion in der einen Datenbank als erfolgreich abgeschlossen gilt und dieselbe Transaktion in der anderen Datenbank wegen eines Fehlers zurückgesetzt wird. In diesem Fall wäre die Gesamtkonsistenz verletzt, da einer Sollbuchung in der einen Datenbank keine Habenbuchung in der anderen gegenüberstehen würde und umgekehrt.

Um nun diese Konsistenz auch über mehrere Datenbanken hinweg zu garantieren, wird ein entsprechendes Protokoll eingesetzt, das Zwei-Phasen-Commit. Für dieses Zwei-Phasen-Commit wird folgendes benötigt:

- Jede Datenbank arbeitet für sich wie gewohnt im Transaktionsbetrieb, d.h. sie besitzt ihre eigene Logdatei, ihre eigene Metadatenverwaltung und führt nach Beendigung einer Transaktion ein „lokales *Commit Work*“ durch. Das Gleiche gilt sinngemäß für Checkpoints und andere Performancemaßnahmen wie etwa Datenbankpufferung.
- Gleichzeitig wird einer der beteiligten Rechner als Koordinator eingesetzt. Dort wird zusätzlich ein systemweites Protokoll gefahren, das eine eigene Logdatei und ein globales *Commit Work* beinhaltet.

Der Algorithmus ist grob in Tab. 43 wiedergegeben. Mit diesem Vorgehen kann die Konsistenz des Gesamtsystems immer sichergestellt werden, egal in welcher der beteiligten Datenbanken ein Fehler auftritt. Wird etwa lokal entschieden, eine noch nicht abgeschlossene Transaktion abzubrechen, so wird diese lokal zurückgesetzt und gleichzeitig eine entsprechende Meldung an den Koordinator geschickt. Dieser gibt ein *Rollback Work* an alle angeschlossenen Datenbanken weiter. Dort werden diese Transaktionen jeweils lokal zurückgesetzt.

Es sei angemerkt, dass dieses Zwei-Phasen-Commit zwar die auftretenden Probleme meistert, aber extrem zeitintensiv ist. Unseren Erfahrungen nach ist bei verteilten Datenbanken unter Verwendung des Zwei-Phasen-Commits in der Regel mit etwa einer Verzehnfachung der Antwortzeiten gegenüber nichtverteilten Anwendungen zu rechnen! In Einzelfällen liegen die Antwortzeiten sogar noch darüber. In einigen bekannten Fällen wird daher auf das Zwei-Phasen-Commit verzichtet, um vertretbare Antwortzeiten auch in verteilten Systemen zu garantieren. In diesen nicht nachahmenswerten Fällen wird das Risiko von Inkonsistenz und Integritätsverletzung wissentlich in Kauf genommen.

Tab. 43 Algorithmus des Zwei-Phasen-Commits

Lokales Abarbeiten einer Transaktion	Jede übergreifende Transaktion arbeitet in den einzelnen Datenbanken lokal: Änderungen werden in der lokalen Logdatei protokolliert.
Melden des Transaktionsendes	Wurde eine Transaktion lokal erfolgreich beendet oder zurückgesetzt, so erfolgt eine entsprechende Meldung an den Koordinator (Phase 1).
Globales Transaktionsende	Der Koordinator sammelt alle lokalen Meldungen. Liegen nur erfolgreiche Rückmeldungen vor, so wird ein globales *Commit*, ansonsten ein *Rollback* eingetragen.
Endgültiges lokales Transaktionsende	Das Ergebnis der globalen Transaktion wird an alle lokalen Rechner zurückgeliefert. Jeder lokale Rechner übernimmt das globale Ergebnis (*Commit* oder *Rollback*) als endgültiges. Erst jetzt ist die Transaktion abgeschlossen (Phase 2).

Die Recovery wird kompliziert, wenn der Rechner ausfällt, der als Koordinator dient. In diesem Fall muss an Hand des Global-Logs ermittelt werden, welche Transaktionen als abgeschlossen gelten. Mit Hilfe der lokalen Logdateien werden dann die Änderungen in den einzelnen Datenbanken gegebenenfalls zurückgesetzt oder übernommen. Auch dies ist aufwendig, führt aber letztendlich zum erwünschten Ziel: Aufsetzen auf den letzten konsistenten Stand des Gesamtsystems!

7.3 Concurrency

Wir sind bisher mehr oder weniger stillschweigend davon ausgegangen, dass alle Transaktionen in zeitlicher Reihenfolge nacheinander ausgeführt werden. Und falls sie sich doch zeitlich überlappen, so sollten sie sich zumindest nicht gegenseitig beeinflussen. In größeren Datenbanken kann dieses Verhalten nicht garantiert werden. Großrechner erreichen schon heute einen Spitzendurchsatz von mehreren hundert Transaktionen pro Sekunde, wobei eine Transaktion meist eine Sekunde oder weniger dauert. Dies bedeutet, dass häufig weit über ein hundert Transaktionen gleichzeitig ablaufen! Aber auch auf kleineren Systemen sind heute zehn parallel laufende Transaktionen keine Seltenheit mehr. Dass sich so viele gleichzeitig ablaufende Transaktionen zu keiner Zeit gegenseitig beeinflussen, ist Wunschdenken und entspricht nicht der Realität. Trotzdem muss immer gewährleistet werden, dass die Konsistenz

der Datenbank nicht verletzt wird. Aus diesem Grund gilt folgende Grundregel:

Grundregel für Concurrency

☞ Jede Transaktion hat so abzulaufen, als sei sie momentan allein im System.

Wie diese Regel schon andeutet, spielt auch bei der Concurrency der Begriff der Transaktion eine zentrale Rolle. Nur im Transaktionsbetrieb sind Recovery und Concurrency überhaupt denkbar. Leider ist diese Grundregel auch im Transaktionsbetrieb nicht einfach zu erfüllen. Glücklicherweise lassen sich aber alle auftretenden Problemfälle in drei Kategorien einteilen, in

① das Problem der verlorengegangenen Änderung,

② das Problem der Abhängigkeit von noch nicht abgeschlossenen Transaktionen,

③ das Problem der Inkonsistenz der Daten.

Beginnen wir mit dem ersten Problem: Sei *R* eine gegebene Relation, in der Änderungen vorgenommen werden sollen. Dazu wird in der Regel wie folgt vorgegangen: Zuerst werden die momentan gespeicherten Daten der Relation *R* gelesen. Anschließend entscheidet der Anwender (oft in Abhängigkeit der gelesenen Daten) über die erforderlichen Änderungen.

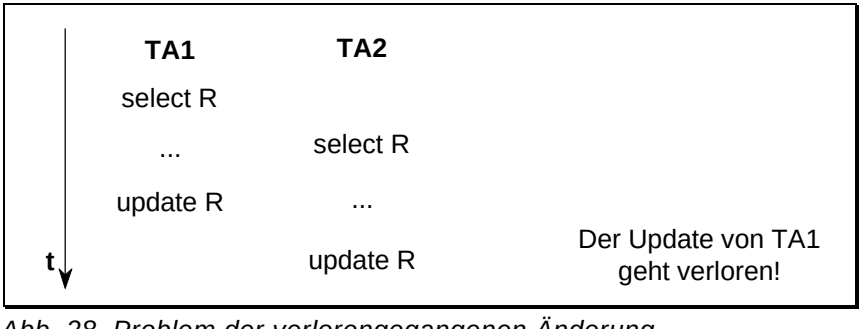


Abb. 28 Problem der verlorengegangenen Änderung

Ein Beispiel hierfür ist der Ein- und Verkauf einer Firma. Zunächst wird nachgesehen, ob ein Artikel auf Lager ist. Wenn ja, so wird die angeforderte Menge von der vorhandenen subtrahiert. Wir benötigen demnach einen *Select*-Befehl, gefolgt von einem *Update*. Zwischen diesen beiden SQL-Befehlen

werden noch weitere Befehle ausgeführt, gegebenenfalls wird sogar noch auf eine Antwort des Anwenders gewartet. Abb. 28 zeigt die Problematik auf, wenn zwei Anwender (Transaktionen *TA1* und *TA2*) fast gleichzeitig das gleiche Datum ändern wollen. *TA1* und *TA2* lesen die gleichen gespeicherten Informationen aus der Relation *R* und reagieren unabhängig voneinander. *TA1* schreibt sein Ergebnis in die Datenbank zurück, *TA2* folgt zeitlich versetzt und überschreibt damit das Ergebnis von *TA1*. Dies würde unweigerlich die Konsistenz zerstören, wenn die Transaktion *TA1* noch weitere von der Änderung abhängige Daten bearbeitet.

Das zweite Problem sieht harmloser aus, ist in letzter Konsequenz aber ähnlich problematisch. Betrachten wir hierzu Abb. 29. Die Transaktion *TA2* führt eine Änderung durch, die aber später mittels eines *Rollback Work* wieder zurückgenommen wird. Zwischenzeitlich liest die Transaktion *TA1* dieses Datum, das aber so niemals gültig wird. Noch schlimmer ist der Fall, wenn *TA1* nicht nur ein ungültiges Datum liest, sondern dieses Datum noch ändert. Das Rücksetzen der Transaktion *TA2* erfolgt mit Hilfe des Before-Images von *TA2*, und damit wird der Update von *TA1* überschrieben.

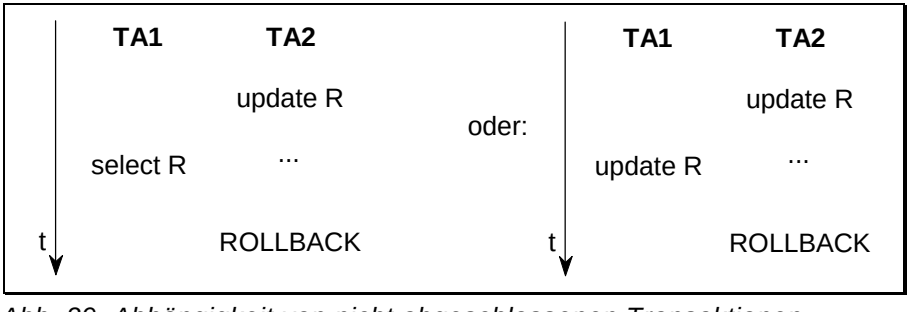


Abb. 29 Abhängigkeit von nicht abgeschlossenen Transaktionen

Zu beachten ist, dass im Falle einer Datenbankpufferung (im Arbeitsspeicher) zusätzliche Schwierigkeiten auftauchen. Wurde etwa der Block der Relation *R* nach dem Update von *TA1* bereits in die Datenbank verdrängt, und wird beim anschließenden Rollback von *TA2* der entsprechende Puffer im Arbeitsspeicher für ungültig erklärt (invalidiert) und vermerkt, dass jetzt die Daten in der Datenbank gültig sind, so wäre plötzlich der Update von *TA1* der gültige Stand.

Wir sehen, dass viele Ergebnisse willkürlich sind und von der momentanen Datenbankumgebung abhängen. Auch die Recovery bereitet unter dem Gesichtspunkt der Concurrency erhebliche Probleme. Es ist die Reihenfolge der abgearbeiteten Transaktionen zu beachten, doch im Parallelbetrieb gibt es häu-

fig kein eindeutiges früher oder später! Weiter muss das Schreiben der Metadaten und das Schreiben in die Logdatei synchronisiert werden.

Zeigen wir zunächst jedoch unser drittes Problem auf, das Problem der Inkonsistenz. Betrachten wir dazu drei Konten eines Geldinstituts (Abb. 30). Die Transaktion *TA1* überprüft den Inhalt der drei Konten und gibt die Summe dieser Konten aus. Zur gleichen Zeit bucht die Transaktion *TA2* einen Betrag von 600 DM von Konto 3 auf Konto 1 um. Damit hat sich die Summe der Konten nicht verändert (in unserem Beispiel 1400 DM), doch unsere Transaktion *TA1* liefert ein falsches Ergebnis zurück!

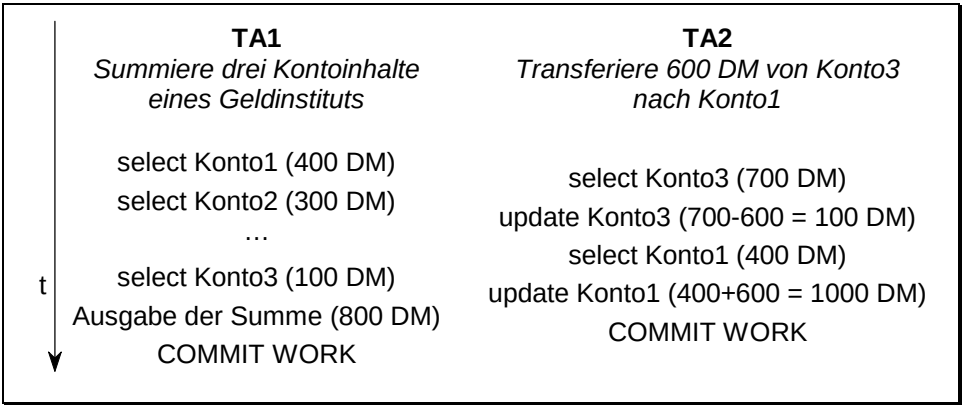


Abb. 30 Problem der Inkonsistenz

Dieser dritte Fall zeigt sehr deutlich, dass die Transaktion *TA1* ganz erheblich die Grundregel verletzt. Würde sie alleine ablaufen, erhielten wir den korrekten Betrag von 1400 DM. Läuft sie ungünstigerweise zusammen mit *TA2*, so bekommen wir ein ganz anderes Ergebnis. Wir benötigen daher Mechanismen, die dafür sorgen, dass die Grundregel unter allen Umständen eingehalten wird. Es geht nicht an, dass eine Transaktion einmal das eine und ein andermal ein anderes Ergebnis zurückliefert. In der Praxis treten die hier vorgestellten Probleme auch beim Eintrag von Daten in die Logdatei auf, wegen des dort sehr hohen Ausgabeverkehr sogar forciert.

Die in der Praxis brauchbaren Mechanismen sind auf alle drei vorgestellten Fälle anwendbar: Es handelt sich um die folgenden Methoden:

- optimistische Concurrency-Strategie,
- pessimistische Concurrency-Strategie.

Die optimistische Strategie geht von folgender Überlegung aus: Jede Transaktion darf in der Datenbank beliebig ändern, wobei obige Effekte bewusst in Kauf genommen werden. Wird allerdings bei Transaktionsende festgestellt, dass auch andere Transaktionen die gleichen Daten änderten, so wird ein *Rollback* durchgeführt und die Transaktion neu gestartet.

Ganz anders arbeitet die Strategie des pessimistischen Verfahrens. Hier werden alle in einer Transaktion angeforderten Daten, auch nur gelesene, für alle anderen Transaktionen gesperrt. Erst bei Transaktionsende werden diese Sperren wieder freigegeben. Ein Zugriff auf diese Daten ist demnach für alle anderen Transaktionen verwehrt, solange die Transaktion, die die Daten als erste anfasste, noch nicht abgeschlossen ist.

Das optimistische Verfahren ist relativ leicht implementierbar, hat aber den Nachteil, dass bei häufigen Zugriffen auf bestimmte Daten viele Transaktionen zurückgesetzt werden müssen. Dies kann im schlimmsten Fall sogar dazu führen, dass zwei Transaktionen wechselseitig auf das gleiche Datum zugreifen, so dass beide zurückgesetzt werden. Anschließend starten beide von neuem, greifen wieder zu, werden wieder zurückgesetzt, und dieses Spiel geht beliebig weiter. Außerdem löst dieses Verfahren zwar unser erstes und drittes, nicht aber ohne weiteres unser zweites Problem, das Problem der nichtabgeschlossenen Transaktionen. Wir müssten nämlich dazu auch alle nur gelesenen Daten in unser Verfahren mit einbeziehen. Es ist nun leicht ersichtlich, dass das optimistische Verfahren nur dann eingesetzt werden kann, wenn die Konfliktwahrscheinlichkeit extrem niedrig ist. Wie schon erwähnt, ist die Implementierung einfach und erfolgt über Zugriffszähler auf jeden einzelnen Block im Datenbankpuffer. Dadurch kann jederzeit festgestellt werden, ob auch andere Transaktionen auf die verwendeten Blöcke zugegriffen haben.

Leider gibt es in der Praxis kaum Datenbanken, wo nicht wenigstens einige Daten immer wieder gelesen und geändert werden. Aus diesem Grund wird heute fast ausschließlich das pessimistische Verfahren verwendet. Dieses Verfahren garantiert ebenfalls die Grundregel der Concurrency. Werden beispielsweise zwei Transaktionen fast gleichzeitig gestartet, so wird diejenige zuerst komplett ablaufen, die die Sperre zuerst setzte. Die andere Transaktion wird auf das Ende der Sperre warten müssen und kann erst dann weiterlaufen. Somit werden die beiden Transaktionen synchronisiert, die Transaktionen verhalten sich nicht nur so, als seien sie nacheinander abgelaufen, sie sind es tatsächlich. Der Aufwand des Verfahrens liegt in der erforderlichen Sperrverwaltung, aber auch in weiteren Problemen, die diese Sperrverwaltung mit sich bringt. Wir gehen deshalb im nächsten Abschnitt näher darauf ein.

7.4 Sperrmechanismen

Das im letzten Abschnitt besprochene pessimistische Verfahren erfordert Sperrmechanismen. Wir bezeichnen solche Sperren im Weiteren als Locks. Die Realisierung solcher Locks geschieht in der Praxis über Semaphore, einfache Zähler. Je nach Anwendung kann es verschiedene Lock-Granulate geben:

- Datenbank-Lock: Sperren der gesamten Datenbank,
- Tabellen-Lock: Sperren einer Relation (Tabelle),
- Tupel-Lock: Sperren eines Tupels (Zeile) einer Relation,
- Eintrags-Lock: Sperren eines einzelnen Feldes.

Praktisch jede moderne Datenbank besitzt mindestens einen Lock, der eine ganze Tabelle sperrt. Ein Lock auf eine ganze Datenbank ist hingegen nur dann von Interesse, wenn die Parallelverarbeitung minimal ist. Gelegentlich finden wir den Tupel-Lock, selten hingegen ist das Eintragslocking. Dies ist einfach erklärbar: Je feiner das Lockgranulat ist, um so geringer ist zwar die Konfliktwahrscheinlichkeit, um so aufwendiger wird aber die Lockverwaltung wegen der dann sehr zahlreichen Locks. In der Praxis muss daher ein Kompromiss gefunden werden, so dass meist Tabellenlocks und nur bei häufig zugegriffenen Daten zusätzlich Tupel- oder sogar Eintragslocks verwendet werden.

An dieser Stelle wollen wir nicht tiefer in die Probleme der Lock-Granulate eindringen. Es sei nur noch erwähnt, dass bei gleichzeitiger Verwendung mehrerer Granulate bestimmte Regeln zu beachten sind. Es darf beispielsweise nicht zunächst ein Tupel-Lock und dann erst der Tabellen-Lock der gleichen Relation angefordert werden. Zuerst ist immer der grobmaschigere Lock zu holen, ansonsten könnten Verklemmungen auftreten.

Im weiteren wollen wir der Einfachheit halber davon ausgehen, dass nur Tabellen-Locks existieren, die wir im Folgenden kurz als *Lock* bezeichnen werden, sofern Verwechslungen ausgeschlossen werden können. Sinngemäß gelten unsere weiteren Überlegungen aber auch für alle anderen Lock-Granulate.

Locks sind immer dann anzufordern, wenn ein Datum gelesen oder geschrieben werden soll. Dies geschieht entweder explizit durch den Anwendungsprogrammierer oder implizit (und damit automatisch) durch die Datenbank. Die Freigabe dieses Locks erfolgt in der Regel automatisch bei Transaktionsende. Sollte der Lock bei der Anforderung bereits anderweitig belegt

sein, so wird auf die Freigabe des Locks durch den anderen Anwender gewartet.

Dieses Vorgehen des Holens eines Locks, bevor ein Datum angefasst wird, ist extrem wichtig! Ein Datum ist nämlich nicht automatisch dadurch geschützt, dass es angefasst wird. Eine entsprechende Vorsorge ist erforderlich. Entweder wird die Datenbank vom Systemverwalter so eingestellt, dass sie die Aufgabe des Setzens des dazugehörigen Locks automatisch übernimmt, oder dies muss vor jedem Zugriff explizit selbst durchgeführt werden. Wegen der Concurrency-Probleme kann dieser Lock, bis auf wenige Ausnahmen, auch erst bei Transaktionsende freigegeben werden. Die Freigabe belegter Locks wird in SQL beim *Commit Work* (und *Rollback Work*) immer automatisch durchgeführt.

Wegen des Problems der Abhängigkeit von noch nicht abgeschlossenen Transaktionen (siehe letzter Abschnitt) müssen auch Lesezugriffe mittels Locks gesichert werden. Da in der Praxis Lesezugriffe sehr häufig sind, können leicht Konflikte zwischen konkurrierenden Transaktionen auftreten. Diese Konfliktwahrscheinlichkeit kann jedoch erheblich verringert werden, wenn Sperren in Schreib- und Lesesperren getrennt werden. Dies ist deshalb sinnvoll, weil nichts dagegen spricht, dass ein und das selbe Datum von mehreren Transaktionen gleichzeitig gelesen wird. Um dieses gleichzeitige Lesen zu ermöglichen, definieren wir zwei unterschiedliche Sperrmechanismen:

Definition (Exklusiv-Lock, Share-Lock)

☞ Ein Exklusiv-Lock auf ein Objekt weist alle weiteren Lockanforderungen auf dieses Objekt zurück.

☞ Ein Share-Lock auf ein Objekt gestattet weitere Share-Lockzugriffe auf dieses Objekt, weist aber exklusive Lockanforderungen zurück.

Die Regel zum Setzen des geeigneten Locks lautet nun:

- vor dem lesenden Zugriff auf eine Tabelle wird der Share-Lock dieser Tabelle geholt,
- vor dem schreibenden Zugriff auf eine Tabelle wird der Exklusiv-Lock dieser Tabelle geholt.

Die Regel gilt analog für Tupel- oder Eintragssperren. Aus der Definition des Share-Locks erkennen wir, dass mehrere Transaktionen gleichzeitig den gleichen Share-Lock besitzen dürfen, und damit auch lesend zugreifen können.

Nicht selten tritt nun der Fall ein, dass eine Transaktion zunächst nur lesend auf ein Datum zugreift, unter bestimmten Bedingungen später aber auch Änderungen vornehmen will. Dazu muss diese Transaktion den bisher gehaltenen Share-Lock in einen Exklusiv-Lock umwandeln. Dies geschieht einfach dadurch, dass der Share-Lock-Inhaber eine Exklusiv-Lock-Anforderung abschickt. Belegt im Augenblick keine andere Transaktion den selben Share-Lock, so wird der Lock automatisch in einen Exklusiv-Lock umgewandelt (ohne dass dieser Share-Lock vorher freigegeben wird). Andernfalls muss auf die Freigabe des auch von anderen Transaktionen gehaltenen Share-Locks gewartet werden.

Mit Hilfe dieser Sperrmechanismen wollen wir nun versuchen, unsere drei Probleme aus dem letzten Abschnitt zu lösen. Dazu gehen wir jetzt davon aus, dass alle Transaktionen vor dem Zugriff auf die Relation *R* einen Share- bzw. Exklusiv-Lock holen. Das Vorgehen beim Problem der verlorengegangenen Änderung ist in Abb. 31 angegeben. Wir erkennen, dass zunächst beide Transaktionen einen Share-Lock anfordern und auch erhalten. Der Update erfordert jedoch einen Exklusiv-Lock. Das Anfordern dieses Locks führt somit zum Warten auf Lockfreigabe, da die jeweils andere Transaktion noch einen Share-Lock hält. Letztlich warten beide Transaktionen auf die Freigabe des Locks durch die jeweils andere Transaktion. Diese Verklemmung löst sich von selbst nicht mehr auf, beide Transaktionen warten beliebig lange und können folglich auch den beabsichtigten Update nicht ausführen.

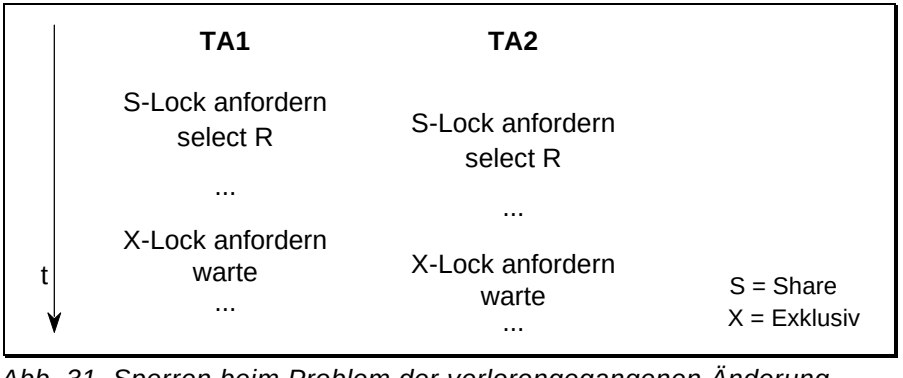


Abb. 31 Sperren beim Problem der verlorengegangenen Änderung

Bei den besprochenen Sperrmechanismen ist im Parallelbetrieb eine solche Verklemmung leider nicht generell auszuschließen. Es hat sich für diese Verklemmungen der Begriff *Deadlock* auch im deutschsprachigen Raum durchgesetzt:

Definition (Deadlock)

☞Eine Verklemmung, bei der zwei oder mehr Transaktionen gegenseitig auf die Freigabe eines oder mehrerer Locks warten, heißt Deadlock.

Glücklicherweise führt das Einführen von Locks beim Problem der Abhängigkeit von nicht abgeschlossenen Transaktionen nicht auch auf einen Deadlock. Im Gegenteil, aus Abb. 32 erkennen wir, dass wir mit Hilfe unserer Sperrmechanismen tatsächlich unser Problem gelöst haben. Da die Transaktion *TA1* einen bereits von Transaktion *TA2* gehaltenen Lock anfordert, muss sie warten, bis der Lock durch das Zurücksetzen (*Rollback Work*) wieder freigegeben wird. Erst jetzt erhält *TA1* den gewünschten Lock und kann ihren Wartezustand damit beenden. Das Lesen oder Ändern der Relation *R* ist nun kein Problem mehr, da nach dem Zurücksetzen von *TA2* die Daten der Relation gültig sind.

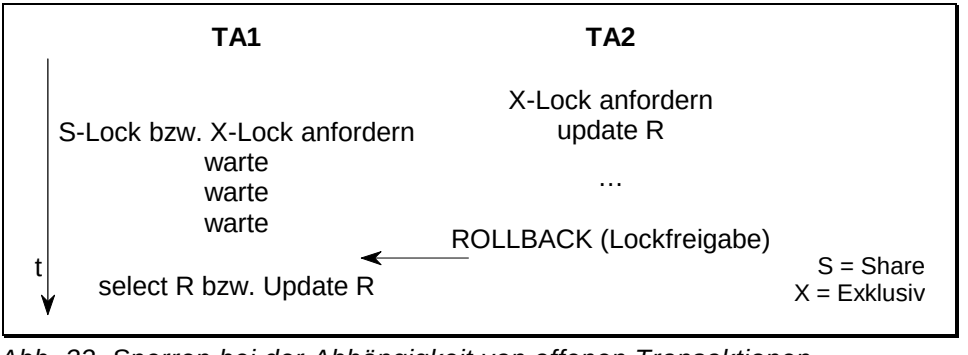


Abb. 32 Sperren bei der Abhängigkeit von offenen Transaktionen

An diesem Beispiel erkennen wir deutlich die Synchronisationswirkung der Sperrmechanismen. Die Transaktion *TA1* wird etwas verzögert, so dass die beiden Transaktionen tatsächlich nacheinander ablaufen. Dies ist notwendig, um die Grundregel der Concurrency zu erfüllen. Generell gilt: Alle miteinander konkurrierenden Transaktionen müssen synchronisiert werden. Natürlich sind Transaktionen, die keine gemeinsamen Daten bearbeiten, davon nicht betroffen. Diese dürfen weiterhin jederzeit parallel abgearbeitet werden.

Bevor wir auf das Deadlock-Problem eingehen, betrachten wir vollständigkeitshalber noch das dritte Problem, wo Inkonsistenz in den Daten auftrat. Transaktion *TA1* erhält zwar in Abb. 33 den Share-Lock zu den Konten 1 und 2, muss aber auf die Freigabe des Locks zu Konto 3 warten. Umgekehrt erhält die Transaktion *TA2* zwar den Exklusiv-Lock zu Konto 3 und den Share-Lock

zu Konto 1, nicht aber dessen Exklusiv-Lock. Damit wartet auch *TA2*, so dass wir wieder eine typische Deadlock-Situation vor uns haben.

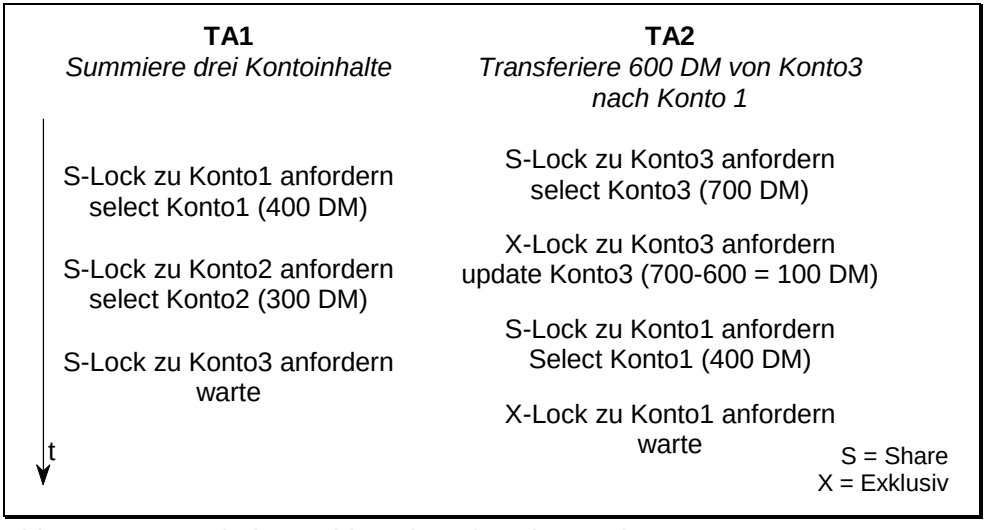


Abb. 33 Sperren beim Problem der Inkonsistenz der Daten

Vielleicht wurde unbeabsichtigterweise der Eindruck vermittelt, dass Deadlocks nur im Zusammenspiel zwischen Share- und Exklusiv-Locks auftreten. Dies ist natürlich nicht richtig. Betrachten wir nur den Fall, dass Transaktion *TA1* nacheinander Exklusivlocks zu den Relationen *R1* und *R2* anfordert. Geschieht diese Anforderung in Transaktion *TA2* in umgekehrter Reihenfolge (erst Lock auf *R2*, dann auf *R1*), so verklemmen sich die beiden Transaktionen bei paralleler Abarbeitung ebenfalls.

Der Einsatz von Sperrmechanismen führte nur beim zweiten Problem zur erhofften Lösung. In den anderen beiden Fällen sind wir auf Deadlocks gestoßen. Allerdings wären diese anderen Probleme ebenfalls gelöst, wenn die Deadlocks erfolgreich aufgelöst werden könnten. Wir müssen daher zur Lösung aller Concurrency-Probleme „nur“ noch die geeigneten Algorithmen zur Deadlockerkennung und -auflösung finden.

7.5 Deadlocks

Wir haben im letzten Abschnitt gesehen, dass das Lösen der drei Concurrency-Probleme mittels Serialisierung zu Deadlocks führen kann. Unser Da-

tenbankverwaltungssystem muss daher einen der beiden Algorithmen zufriedenstellend unterstützen:

- Deadlockvermeidung
- Deadlockerkennung und -auflösung

Beispielsweise beruht eine Strategie der Deadlockvermeidung darauf, dass alle Transaktionen geordnet werden, und dass nur dann auf bereits gehaltene Sperren gewartet wird, wenn eine bestimmte Ordnung zwischen anfordernder und Lock haltender Transaktion vorliegt. Ansonsten wird eine der beiden betroffenen Transaktionen zurückgesetzt und neu gestartet. Es lässt sich zeigen, dass unter diesen Umständen keine Deadlocks auftreten können. Allerdings verursacht dieser Algorithmus einen relativ hohen Overhead, da auch Zugriffe, die nicht zu Deadlocks führen würden, ein Rücksetzen einer Transaktion erzwingen können. In der Praxis werden daher Strategien zur Deadlockvermeidung nur dort eingesetzt, wo das Erkennen von Deadlocks sehr komplex und aufwendig ist, etwa in verteilten Datenbanken. Zu Details der Deadlockvermeidung sei auf [Date83] und [Kudl92] verwiesen.

Werden Deadlocks nicht von vornherein verhindert, so müssen Algorithmen zum Erkennen vorliegender Deadlocks eingesetzt werden. Die einfachste Methode ist, auf Lockfreigabe wartende Transaktionen zu beobachten. Überschreitet die Wartezeit dieser Transaktionen einen vorgegebenen Schwellwert, so wird angenommen, dass sie sich in einem Deadlock befinden. Wie wir sehen, ist dieser Algorithmus zwar einfach, hat aber entscheidende Nachteile: Erstens müssen sich lang wartende Transaktionen nicht immer in einem Deadlock befinden; zweitens ist der optimale Schwellwert der Wartezeit nicht leicht zu ermitteln; und drittens verstreicht eine erhebliche Wartezeit, bevor ein Deadlock erkannt wird. In der Praxis wird dieses Verfahren der Deadlockerkennung daher nur selten verwendet.

Stattdessen werden in der Regel Wartegraphen eingesetzt. Dieses Verfahren funktioniert, vereinfacht dargestellt, wie folgt: Wartet die Transaktion *TA1* auf die Freigabe eines Locks, der von der Transaktion *TA2* gehalten wird, so wird in den Wartegraphen ein Pfeil von *TA1* nach *TA2* gezeichnet. Wartet nun etwa auch *TA2* auf die Freigabe eines Locks durch *TA3*, so kommt ein Pfeil von *TA2* nach *TA3* hinzu. Falls zusätzlich noch *TA3* auf die Freigabe eines Locks durch *TA1* wartet, so führt der weitere Pfeil zu einem geschlossenen Zyklus im Wartegraphen (siehe Abb. 34): Ein Deadlock liegt vor.

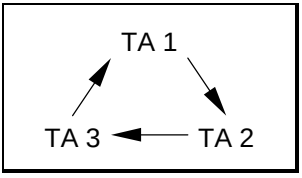


Abb. 34 Wartegraph dreier Transaktionen

Das Erkennen eines Deadlocks mit Hilfe von Wartegraphen ist also einfach. Wartet eine Transaktion auf Lockfreigabe, so wird ein Pfeil im Wartegraphen hinzugefügt. Beim Beenden der Wartezeit wegen Lockfreigabe durch die lockhaltende Transaktion wird der Pfeil wieder entfernt. Führt nun das Setzen eines Pfeils zu einem geschlossenen Zyklus, so liegt ein Deadlock vor.

Dieser Deadlock kann nur dadurch beseitigt werden, indem eine der involvierten Transaktionen zurückgesetzt wird, was auch die Freigabe aller gehaltener Locks dieser Transaktion bewirkt. Die Transaktion muss nun von neuem gestartet werden. Gleichzeitig ist der geschlossene Zyklus aufgebrochen. Die anderen betroffenen Transaktionen können weiterarbeiten. Zu Details sei wieder auf [Date83] und [Kudl92] verwiesen.

7.6 Sperren in SQL-2, MS-Access und Oracle

SQL ist eine Sprache der sogenannten vierten Generation. Wir haben dies bereits bei den Datenbankzugriffen gemerkt. SQL fragte dort nicht, wie ein Datum ausgelesen oder geschrieben wird, sondern nur, welches Datum gelesen oder geschrieben werden soll. Ähnlich verhält es sich bei den Sperrmechanismen in SQL-2 (SQL-1 unterstützt Sperren syntaktisch nicht).

Standardmäßig verlangt SQL, dass Concurrency sicher unterstützt, dass also alle drei weiter oben in diesem Kapitel beschriebenen Probleme der Serialisierung von Transaktionen gelöst werden. SQL interessiert allerdings nicht das *Wie* dieser Lösung. Gleichzeitig erlaubt SQL, falls gewünscht, diese sichere Serialisierung gezielt zu verletzen. SQL definiert dabei 3 Unsicherheitslevel:

- Beliebiges Lesen: Hier werden absichtlich Lesezugriffe immer erlaubt, wodurch das im Abschnitt 7.3 aufgezeigte Problem 2 auftreten kann.
- Nichtwiederholbares Lesen: Hier wird nicht garantiert, dass ein einmal gelesener Eintrag innerhalb der gleichen Transaktion nicht von anderen Transaktionen verändert werden kann. Ein weiterer Zugriff auf den gleichen Eintrag kann also durchaus andere Ergebnisse liefern.

- Phantom-Effekt: Hier wird zugelassen, dass sich die Anzahl der gelesenen Zeilen, die eine gewisse Bedingung erfüllen, im Laufe einer Transaktion ändern kann. Es kann also vorkommen, dass beim zweiten Lesen eine neue, zusätzliche Zeile auftaucht, das Phantom.

Realisieren lassen sich diese „Unsicherheiten“ wie folgt: Das beliebige Lesen wird dadurch möglich, dass vor dem lesenden Zugriff keinerlei Sperren geholt oder gesetzt werden. Das nichtwiederholbare Lesen wird erreicht, indem direkt nach dem Lesezugriff die gehaltene Sperre wieder freigegeben wird, und nicht erst bei Transaktionsende. Der Phantom-Effekt ist ohne größeren Aufwand nur vermeidbar, wenn bei Tupel- oder Eintragslocking zusätzlich auch ein Share-Lock auf die Relation gesetzt wird. Wird dieser Share-Lock immer bis Transaktionsende gehalten, so kann das Phantom nicht auftreten.

Standardmäßig verlangt SQL-2, dass die drei oben aufgeführten Verletzungen der Concurrency nicht auftreten dürfen. Möchte man diese explizit aber doch zulassen, so kann dies mit Hilfe des *Set-Transaction*-Befehls geschehen:

```
SET TRANSACTION ISOLATION LEVEL  Level
```

Mögliche Levelangaben sind: *Read Uncommitted*, *Read Committed*, *Repeatable Read* und *Serializable*, wobei der letzte Level standardmäßig gesetzt ist und volle Serialisierung parallel ablaufender Transaktionen garantiert. Diese Angaben beeinflussen die zugelassenen Unsicherheiten. Eine Aufstellung hierzu finden wir in Tab. 44. In dieser Tabelle wird aufgezeigt, welche Verletzungen der Concurrency bei welcher Levelangabe möglich sind.

Tab. 44 Levelangaben und ihre Auswirkungen

Levelangabe	beliebiges Lesen möglich:	Lesen nicht wiederholbar:	Phantom möglich:
Read Uncommitted	ja	ja	ja
Read Committed	nein	ja	ja
Repeatable Read	nein	nein	ja
Serializable	nein	nein	nein

Natürlich ist in der Regel eine Verletzung der Concurrency unbedingt zu vermeiden. Es gibt jedoch tatsächlich Anwendungen, wo die eine oder andere Levelangabe durchaus ihre Berechtigung besitzt. Stellen wir uns nur den Fall vor, dass parallel zum Tagesbetrieb eine Statistik mitläuft. Für diese Statistik

ist es unerheblich, ob ein bestimmtes Datum kurz vor oder nach seiner Änderung aufgenommen wird. Weiter soll diese Statistikerfassung den Regelbetrieb nicht stören. Hier ist es deshalb durchaus vorstellbar, diese Statistiktransaktion als *Read Uncommitted* zu starten.

MS-Access kennt diese in SQL-2 neu aufgenommenen Möglichkeiten zur Unterstützung des Parallelbetriebs nicht. Stattdessen bedient sich MS-Access eigener Mechanismen. Unter Optionen kann dort eine von drei Sperrmechanismen gewählt werden:

keine Sperrungen
alle Datensätze
bearbeiteter Datensatz

Der Mechanismus *keine Sperrungen* entspricht in etwa dem Level *Read Uncommitted*. Werden alle Datensätze gesperrt, so erfolgt immer das Sperren von allen Relationen, die zur Zeit bearbeitet werden. Im dritten Fall werden nur die gerade bearbeiteten Tupel gesperrt. Greift eine Transaktion, die in MS-Access explizit mit dem Befehl *BeginTrans* gestartet werden muss, auf einen gesperrten Datensatz zu, so erfolgt eine Fehlermeldung. Der Anwender muss explizit reagieren. Ein Warten auf Sperrfreigabe erfolgt nicht automatisch. Die Befehle *CommitTrans* und *Rollback* beenden eine Transaktion in MS-Access, der zweite Befehl setzt dabei die Transaktion zurück. Standardmäßig setzt MS-Access keine Sperrungen.

Oracle unterstützt SQL-2 in wesentlichen Teilen. Im *Set-Transaction*-Befehl ist genau einer der zwei Level *Read Committed* und *Serializable* erlaubt. Der erste ist standardmäßig gesetzt. Hier sind parallele lesende Zugriffe erlaubt, wobei Änderungen durch parallel lesende Transaktionen grundsätzlich erst nach einem erfolgten *Commit Work* gesehen werden. Der höhere Sicherheitslevel *Serializable* ist mit dem *Set-Transaction*-Befehl jeweils explizit zu Beginn einer Transaktion einzustellen und gilt dann genau für diese Transaktion. Der Datenbankadministrator kann darüberhinaus generell den Sicherheitslevel *Serializable* als Standard vorgeben. Der auch im SQL-Standard enthaltene Befehl *Set Transaction Read Only* erlaubt außerdem, nur lesende Zugriffe während einer Transaktion zuzulassen. Gesetzte Sperren werden grundsätzlich beim *Commit Work* bzw. *Rollback Work* wieder freigegeben.

Weiter bietet Oracle dem Anwender an, Datensperren innerhalb von Programmen selbst zu setzen. Damit können beliebige Level bis hin zu *Serializable* realisiert werden. Der Befehl zum Setzen einer Sperre in Oracle lautet:

```
LOCK TABLE Tabellenname IN Lockmodus MODE
```

Die wichtigsten Lockmodi sind:

```
[ ROW ] EXCLUSIVE
[ ROW ] SHARE
```

Damit kann eine Relation exklusiv oder nur zum Lesen gesperrt werden. Die zusätzliche Angabe *Row* veranlasst Oracle, nur die betroffenen Tupel der Relation zu sperren. Dieses sehr feine Sperrgranulat erlaubt daher den Zugriff vieler parallel arbeitender Benutzer ohne merkliche Behinderungen. Oracle arbeitet in den beiden Sicherheitsleveln *Read Committed* und *Serializable* intern ebenfalls mit den „Row"-Sperren. Es sei darauf hingewiesen, dass mit diesen Befehlen Relationen nicht automatisch für fremde Zugriffe gesperrt sind. Nur wenn man sich an das Protokoll hält, indem vor lesenden Zugriffen ein *Share*-Lock und vor schreibenden ein *Exclusiv*-Lock geholt wird, kann Concurrency garantiert werden. Tritt ein Deadlock auf, so wird dieser von Oracle sofort erkannt. Eine der involvierten Transaktionen wird über eine entsprechende Fehlermeldung informiert.

7.7 Zusammenfassung

Dieses Kapitel zeigt die Wichtigkeit des Transaktionsbegriffs auf. Sowohl für Recovery als auch für Concurrency bilden Transaktionen die Grundeinheit. Bemerkenswert ist der hohe Aufwand, der sich hinter einem sicheren Datenbankbetrieb verbirgt, ein Aufwand in zweierlei Hinsicht: erstens beim Datenbankhersteller mit der Implementierung der entsprechenden Möglichkeiten und zweitens im Betrieb durch den zusätzlichen Rechenaufwand und die häufigen Plattenzugriffe.

Wir haben gesehen, dass Concurrency in der Praxis mittels Sperrmechanismen sichergestellt wird. Diese Sperrmechanismen garantieren eine sequentielle Abarbeitung aller Transaktionen, die auf gemeinsame Daten zugreifen. Da Deadlocks auftreten können, müssen diese vom Datenbankverwaltungssystem erkannt und aufgelöst werden. Letzteres geschieht in der Regel mittels des Zurücksetzens einer der betroffenen Transaktionen. Dies ist allerdings nur mittels Recoverymaßnahmen möglich. Wir erkennen die enge Verzahnung zwischen Recovery und Parallelverarbeitung.

Die vorgestellten Mechanismen zur Sicherstellung der Recovery unterstützen auch voll den Parallelbetrieb. Erst die Concurrency erzwingt beispielsweise wegen der vielen parallel abzuarbeitenden Transaktionen Checkpoints. Auch wird das Arbeiten mit der Logdatei, das Halten von Seiten im Cache und das Führen von Metadaten bei zugelassener Parallelverarbeitung erheblich aufwendiger.

Recovery garantiert durch das Speichern von Before- und After-Images zusammen mit dem Commit-Eintrag in der Logdatei die Rekonstruierbarkeit einer Datenbank in fast allen nur denkbaren Situationen. Beispielsweise muss bei Stromausfall nur auf den letzten Checkpoint aufgesetzt und die Datenbanksitzung ab diesem Zeitpunkt unter Zuhilfenahme der Logdatei nachvollzogen werden.

Als wichtigstes Ergebnis halten wir nochmals fest, dass Recovery und Concurrency ohne den Transaktionsbetrieb nicht denkbar wären. Recovery und Concurrency garantieren die Konsistenz im Fehlerfall bzw. im Parallelbetrieb. Ihre „Arbeitseinheiten“ sind daher konsistenzerhaltende Mutationen: die Transaktionen.

7.8 Übungsaufgaben

1) In einem sicheren Datenbankbetrieb wurde gerade das Transaktionsende in die Logdatei geschrieben. Doch noch vor der Rückmeldung der im Prinzip beendeten Transaktion an den Benutzer stürzt das ganze System ab. Wird beim nächsten Hochfahren die Transaktion deshalb zurückgesetzt? Begründen Sie Ihre Antwort!
2) Was sind Checkpoints? Beschreiben Sie den Nachteil, wenn eine Datenbank ohne Checkpoints arbeiten würde.
3) In nicht wenigen Datenbankanwendungen ist die Recovery-Unterstützung deaktiviert oder zumindest stark eingeschränkt. Woran mag dies liegen?
4) Welche Schritte müssen vom Systemadministrator bzw. vom Datenbankverwaltungssystem im Einzelnen durchgeführt werden, wenn eine einzelne Transaktion wegen eines Software-Fehlers abstürzt?
5) Der Systemverwalter bemerkt im laufenden Datenbankbetrieb, dass die Magnetplatte, auf der die Daten der Datenbank gespeichert sind, nicht mehr fehlerfrei arbeitet (Schreibfehler). Welche Maßnahmen müssen im Einzelnen ergriffen werden, um mögliche fehlerbehaftete Schreibvorgänge seit der letzten Sicherung zu eliminieren?
6) Unter welchen Voraussetzungen kann auf ein Before-Image verzichtet werden?

7) Unter welchen Voraussetzungen kann auf ein After-Image verzichtet werden?
8) Wann wird ein Zwei-Phasen-Commit benötigt?
9) Auf zwei Rechnern *A* und *B* sei je eine Datenbank installiert. Transaktionen vom Typ1 greifen nur auf die Datenbank von Rechner *A*, Transaktionen vom Typ2 nur auf die von Rechner *B* zu. Transaktionen vom Typ3 greifen gleichzeitig auf beide Datenbanken zu, werden aber ausschließlich vom Rechner *A* aus gestartet. Kann man unter diesen einschränkenden Bedingungen auf ein Zwei-Phasen-Commit verzichten?
10) Was macht einen Zwei-Phasen-Commit Ihrer Meinung nach so extrem zeitaufwendig?
11) Wie erkennt man Deadlocks? Wie beseitigt man sie, ohne die Konsistenz der Datenbank zu zerstören?
12) Im Parallelbetrieb kann man auch ohne Sperrmechanismen auskommen. Um welches Verfahren handelt es sich, und warum wird es kaum eingesetzt?
13) In einem kleinen Mehrbenutzer-Datenbankverwaltungssystem existiere als Sperrmechanismus nur ein einziger Lock (globaler Datenbanklock). Kann in diesem System ein Deadlock überhaupt entstehen?
14) Im Parallelbetrieb ist es nicht immer leicht zu sagen, ob eine Transaktion *A* vor oder nach einer Transaktion *B* ablief. Kann dieses Problem immer entschieden werden, wenn wir mit Sperrmechanismen arbeiten?
15) Betrachten wir das Problem der Inkonsistenz der Daten und deren Lösung mittels Locks in Abb. 33. Laufen wir ebenfalls in einen Deadlock, wenn wir in Transaktion *TA1* zunächst Konto 3, dann Konto 2 und Konto 1 abgefragt hätten? Wäre dann ohne Locks das Problem der Inkonsistenz der Daten überhaupt aufgetreten?

8 Sicherheit und Integrität

Die Begriffe Sicherheit und Integrität werden häufig im gleichen Zusammenhang gebraucht. Dies liegt daran, dass es sich bei der Integrität genaugenommen um die *Sicher*stellung der Korrektheit der gespeicherten Daten handelt. Auch wir haben deshalb Sicherheit und Integrität in einem Kapitel zusammengefasst. Trotzdem haben beide Themen eine andere Zielsetzung:

- Um Sicherheit zu gewährleisten, wird jeder Benutzer überprüft, ob er berechtigt ist, die Dinge zu tun, die er gerade versucht.
- Um Integrität zu gewährleisten, wird überprüft, dass die Dinge, die gerade versucht werden, auch korrekt ablaufen.

Natürlich müssen Sicherheit und Integrität ineinandergreifen: Vor jedem Zugriff wird die Zugriffsberechtigung überprüft, beim Zugriff wird die Korrektheit kontrolliert. Nur so ist gewährleistet, dass der Gesamtdatenbestand integer ist und bleibt; geändert von ausschließlich integren (zugelassenen) Personen und kontrolliert auf fehlerfreie Eingaben und Operationen.

Während die Sicherheit bereits mit SQL-1 gut unterstützt wird, existieren in SQL-1 erhebliche Lücken zum Thema Integrität. Erst mit SQL-1+ und insbesondere mit SQL-2 (im Intermediate Level) werden diese Lücken hinreichend gut geschlossen. Da diese SQL-2-Möglichkeiten von den Datenbankherstellern meist noch nicht vollständig implementiert sind, werden wir beim Thema Integrität zusätzlich zu den SQL-2-Möglichkeiten auch auf die eher rudimentären SQL-1-Mittel eingehen.

8.1 Sicherheit

Wenn wir das Thema Sicherheit wirklich ernst nehmen, beginnt Sicherheit nicht erst beim Datenbankverwaltungssystem. Der erste Sicherheitsaspekt ist die physische Aufstellung des Rechners und die Ermittlung des Personenkreises, der Zutritt zum Rechnerraum besitzt. Die zweite Sicherheitsstufe ist der Anschluss ans Datennetz und die Festlegung der Betriebssystem- und Netzkennungen und die dazugehörigen Schutzwörter zum softwareseitigen Zugriff

auf diesen Rechner. Dies ist Aufgabe des Netzadministrators und des Betriebssystemverwalters. Bei besonders schützenswerten Datenbeständen spielt auch die Art der Zugriffskontrolle eine wichtige Rolle. Neben Schutzwörtern wird hier meist eine zusätzliche Identifizierung, etwa mittels Chip-Karte, verlangt. Das Betriebssystem und die Datenbank müssen ferner dafür sorgen, dass der Zugriff auf die Daten der Datenbank ausschließlich über die Datenbank erfolgen kann. Weitere Schutzmaßnahmen sind denkbar, wie hardwareunterstützter Speicherschutz, Verschlüsselung von Daten, regelmäßige Änderung der Schutzwörter und vieles mehr.

Erst die nächste nicht weniger wichtige Sicherheitsstufe betrifft schließlich das Datenbankverwaltungssystem. Der Datenbankadministrator vergibt Datenbank-Benutzerkennungen mit dazugehörigen Schutzwörtern und erteilt den einzelnen Benutzern die Erlaubnis zum Zugriff auf Daten. Diese Erlaubnis erstreckt sich entweder auf alle Daten einer Datenbank oder bezieht sich, was die Regel ist, nur auf eine Teilmenge eines Datenbankbestandes. Diese Teilmenge kann für jeden Benutzer verschieden sein, abhängig von den Aufgaben des jeweiligen Benutzers. Die optimale Erstellung von entsprechenden Zugriffsberechtigungen erfordert ein hohes Maß an Verantwortung des Administrators und eine sorgfältige Einteilung der Zuständigkeitsbereiche jeder einzelnen Benutzergruppe.

Hat der Datenbankadministrator die einzelnen Kennungen mit den entsprechenden Zugriffsmöglichkeiten vergeben, so muss das Datenbankverwaltungssystem die Einhaltung dieser Vorschriften kontrollieren. Diese Kontrolle erfolgt vor jedem einzelnen Zugriff auf ein Datum der Datenbank. Besitzt der Benutzer die entsprechende Zugriffserlaubnis nicht, so wird der Zugriff mit einer entsprechenden Fehlermeldung abgewiesen. Im Transaktionsbetrieb könnte eine solche Zugriffsverweigerung zum Abbruch und damit zum Zurücksetzen der aktuellen Transaktion führen.

Um diese Kontrolle durchführen zu können, wird das Datenbankverwaltungssystem alle vergebenen Zugriffsberechtigungen intern speichern. Wie bereits in Kapitel 6 besprochen, existieren hierzu eigene Systemtabellen, wobei der Benutzer auf seine eigenen Einträge in der Regel Lesezugriffsrechte besitzt.

SQL unterstützt die Sicherheit im Datenbankbetrieb hinreichend. Es gibt hierzu zwei sich ergänzende Konstrukte: Die Gewährung und das Entziehen von Zugriffen auf Tabellen und Sichten mittels der Befehle *Grant* und *Revoke*, und die Erstellung geeigneter Sichten. Beginnen wir mit der Vergabe der Zugriffsrechte. (*Grant*-Befehl).

8.1.1 Der Grant- und Revoke-Befehl

In SQL wird mit dem *Grant*-Befehl ein Zugriffsrecht vergeben, und mit dem *Revoke*-Befehl wieder entzogen. Um Missbrauch zu verhindern, ist das Ausführen eines *Grant*- und *Revoke*-Befehls nur einem bestimmten Benutzerkreis gestattet.

Grundsätzlich gilt, dass der Eigentümer einer Basisrelation, einer Sicht oder eines Gebiets alle Rechte auf *sein* Eigentum besitzt. Diese Rechte können dem Eigentümer auch nicht entzogen werden und erlöschen erst mit dem Entfernen der entsprechenden Relationen oder Gebiete aus der Datenbank.

In SQL ist der Benutzer, der einen entsprechenden *Create*-Befehl ausführt, automatisch Eigentümer dieses neu erzeugten Datenbankelements. Der Systemverwalter wiederum legt beim Erzeugen einer Datenbank fest, welche Benutzer dieses Privileg zum Erzeugen von Relationen und Gebieten überhaupt erhalten.

Da der Eigentümer alle Rechte besitzt, kann er nach Belieben *seine* Relationen oder Gebiete mittels eines *Alter*-Befehls verändern oder mittels des entsprechenden *Drop*-Befehls auch löschen. Er vergibt mit dem *Grant*-Befehl gezielt Zugriffsrechte an andere Benutzer und entzieht sie bei Bedarf mit dem *Revoke*-Befehl wieder. Wegen dieser umfassenden Rechte wird der Eigentümer einer Datenbank in der Praxis meist der Systemverwalter selbst oder ein entsprechend autorisierter Benutzer sein.

Umgekehrt besitzen alle Benutzer aus Sicherheitsgründen standardmäßig keinerlei Rechte auf das Eigentum anderer Benutzer. Eine Datenbank macht in der Praxis aber erst dadurch Sinn, dass neben dem Eigentümer auch andere Personen auf die Inhalte von Basisrelationen und Sichten zugreifen können. Deshalb wird der Eigentümer entsprechende Rechte an andere weitergeben. Grundsätzlich kann der Eigentümer aber, auch wenn er es wollte, keine Rechte weitergeben, die den Aufbau von Relationen beeinflussen. Dies betrifft alle DDL-Befehle etwa zum Anfügen von Spalten oder gar zum Löschen einer Relation. Zur Weitergabe von reinen Zugriffsrechten auf die Dateninhalte steht ihm jedoch der *Grant*-Befehl zur Verfügung.

Wichtig

☞ Mittels des *Grant*-Befehls wird angegeben, welcher Benutzer auf welche Relation (oder Sicht oder Gebiet) welche Zugriffsrechte erhält.

Die Syntax des *Grant*-Befehls lautet (zur Notation siehe Anhang B):

```
GRANT  Zugriffsrecht  [ , ... ] ON [ TABLE ] { Tabellenname  |  Sichtname }
                                 TO  Benutzer  [ , ... ]
                                 [ WITH GRANT OPTION ]

GRANT  Zugriffsrecht  ON DOMAIN  Gebietsname
                      TO  Benutzer  [ , ... ]
                      [ WITH GRANT OPTION ]
```

Im ersten Fall werden Zugriffsrechte auf Relationen (die Angabe *Table* ist wahlfrei und ist nur ein Füllwort) und im zweiten Fall Rechte auf Gebiete vergeben. Gebiete sind die Definitionsbereiche der Attribute. Wir haben darüber bereits in Kapitel 3 kurz gesprochen und werden im folgenden Abschnitt zur Integrität näher darauf eingehen. Die zu vergebenden Zugriffsrechte werden durch Kommata voneinander getrennt aufgezählt. Die einzelnen Rechte sind in Tab. 45 zusammengefasst. Ebenfalls können, durch Kommata getrennt, mehrere Benutzer gleichzeitig angegeben werden, die diese aufgeführten Rechte erhalten sollen. Die optionale Klausel *With Grant Option* erlaubt das Weitergeben der erworbenen Rechte und wird weiter unten besprochen.

Tab. 45 Zugriffsrechte in SQL

Zugriffsrecht	erlaubt ...
Select	den lesenden Zugriff auf die angegebene Relation
Update	das Ändern von Inhalten der angegebenen Relation
Update (x)	das Ändern des Attributwerts x der angegebenen Relation
Delete	das Löschen von Tupeln der angegebenen Relation
Insert	das Einfügen neuer Tupel in die angegebene Relation
Insert (x)	das Einfügen des Attributwerts x in die angegebene Relation
References (x)	das Referenzieren des Attributs x der angegebenen Relation
Usage	das Verwenden des angegebenen Gebiets

Die ersten fünf der in Tab. 45 aufgeführten Zugriffsrechte stehen seit SQL-1 zur Verfügung und sind leicht nachvollziehbar. Hiermit werden auf die angegebenen Relationen einschließlich Sichten Zugriffe zum Lesen (*Select*), Ändern (*Update*), Löschen (*Delete*) und Einfügen (*Insert*) vergeben. Diese Zugriffsrechte stimmen also genau mit den Namen der entsprechenden SQL-Zugriffsbefehle überein. Das Recht zum Ändern kann zusätzlich auf ein bestimmtes Attribut eingeschränkt werden, wobei statt eines einzelnen Attributs auch eine durch Kommata getrennte Attributsliste erlaubt ist.

Mit SQL-2 wurde auch für das Einfügerecht *Insert* die Möglichkeit geschaffen, dieses auf bestimmte Spalten einzuschränken. Nur die in der Spaltenliste angegebenen Attribute dürfen eingefügt werden. Alle anderen Attribute werden mit einem Standardwert, in der Regel dem *Null*-Wert, belegt. Neu ist auch das Referenzrecht. Nur wer dieses Recht besitzt, kann mittels Integritätsbedingungen die angegebenen Attribute referenzieren. Ebenfalls neu ist das Usage-Recht. Dieses bezieht sich als einziges Recht nicht auf Relationen, sondern auf Gebiete. Wer das Usage-Recht auf ein solches Gebiet besitzt, darf dieses Gebiet in SQL-Befehlen benutzen.

Im Allgemeinen kann ein Benutzer Rechte, die er von einem anderen Benutzer erhielt, nicht an einen dritten weiterreichen. Nur wenn ihm auch diese Erlaubnis ausdrücklich gegeben wurde, steht ihm dieses Weitergaberecht zu. Ein Benutzer drückt dieses Recht zur Erlaubnis der Weitergabe von Zugriffsrechten im *Grant*-Befehl mittels der optionalen Klausel *With Grant Option* aus.

Wegen dieser Klausel *With Grant Option* besteht die Möglichkeit, dass auch Nichteigentümer Zugriffsrechte zu Relationen oder Gebieten an andere Benutzer vergeben dürfen. Es versteht sich dabei von selbst, dass eine Weitergabe von Zugriffsrechten grundsätzlich nicht möglich ist, wenn man diese nicht selbst besitzt.

Will ein Benutzer all seine Rechte an einer Relation oder einem Gebiet an andere Benutzer weitergeben, so kann er statt der komplett aufgeführten Zugriffsliste kurz

```
ALL PRIVILEGES
```

schreiben. Alle in der Benutzerliste des *Grant*-Befehls aufgeführten Benutzer erhalten dann automatisch die gleichen Zugriffsrechte, die der diesen Befehl eingebende Benutzer selbst auf das Objekt besitzt. Auch die Benutzerliste kann mittels des Bezeichners *Public* abgekürzt werden. In diesem Fall erhalten alle Benutzer die aufgeführten Zugriffsrechte auf das entsprechende Objekt.

Einmal gewährte Zugriffsrechte können jederzeit mittels des SQL-Befehls *Revoke* wieder entzogen werden. Die Syntax des *Revoke*-Befehls ähnelt der des *Grant*-Befehls und lautet

```
REVOKE [ GRANT OPTION FOR ] { Zugriffsrecht [ , ... ] | ALL PRIVILEGES }
   ON [ TABLE ] { Tabellenname | Sichtname }
   FROM Benutzer [ , ... ]
   { RESTRICT | CASCADE }
```

```
REVOKE [ GRANT OPTION FOR ] Zugriffsrecht
   ON DOMAIN Gebietsname
   FROM Benutzername [ , ... ]
   { RESTRICT | CASCADE }
```

Wird einem Benutzer ein Zugriffsrecht entzogen, das er weitergegeben hat, und ist im *Revoke*-Befehl die Option *Cascade* angegeben, so ist es auch diesen anderen Benutzern entzogen. Wird einem Benutzer ein Zugriffsrecht von mehreren Anwendern gewährt, so verliert er dieses Recht erst endgültig, wenn ihm alle Anwender dieses Zugriffsrecht entziehen. Wird im *Revoke*-Befehl die Option *Restrict* verwendet, so wird die Ausführung des Befehls abgewiesen, wenn die zu entziehenden Rechte auch an andere weitergegeben wurden. Ist zusätzlich die Option *Grant Option For* angegeben, so werden nicht die aufgezählten Zugriffsrechte selbst, sondern nur das Recht des Weitergebens dieser Rechte entzogen. Diese Option ist damit das Gegenstück zur Option *With Grant Option* im *Grant*-Befehl.

Zum Schluss dieses Abschnitts wollen wir ein kleines Beispiel angegeben. Wir gestatten dem Benutzer *Perschef* das Recht zum Lesen der Relation *Personal* (Tab. 18) und das Recht zum Ändern der Attribute *Gehalt* und *Vorgesetzt* dieser Relation. Er darf diese Rechte auch an andere Benutzer weitergeben:

```
GRANT Select, Update (Gehalt, Vorgesetzt)
   ON Personal TO Perschef
   WITH GRANT OPTION ;
```

Oracle unterstützt keine Gebiete (Domains), so dass die entsprechenden *Grant*- und *Revoke*-Befehle hier nicht gültig sind. Alle Zugriffsrechte außer *Usage* können aber gesetzt bzw. wieder entzogen werden. Darüberhinaus unterstützt Oracle Rechte zum Hinzufügen von Spalten (Recht: *Alter*) und von Indexen (Recht: *Index*). Weiter werden in Oracle im *Revoke*-Befehl die Bezeichner *Grant Option For*, *Restrict* und *Cascade* nicht unterstützt. Die Wirkung des Befehls entspricht hier immer der Option *Cascade*: Die betroffenen Benutzer und alle Benutzer, an die diese Rechte weitergereicht wurden, verlieren die angegebenen Zugriffsrechte. Eine Ausnahme bildet nur das Entziehen des Referenzrechts. Hier muss explizit die Option *Cascade Constraints* gesetzt werden, um das Entziehen kaskadierend durchzuführen.

MS-Access kennt die SQL-Befehle *Grant* und *Revoke* nicht. Hier gibt es innerhalb von SQL nur die Option *With OwnerAccess Option* innerhalb eines *Select*-Befehls, um bestimmte Zugriffsrechte zu setzen. Außerdem existieren

einige spezielle Befehle, die Zugriffsrechte setzen, entziehen und verändern (z.B. *RunPermission*-Befehl, *Permission*-Eigenschaft).

8.1.2 Zugriffsrecht und Sichten

Im letzten Abschnitt haben wir gesehen, dass ein Eigentümer mit Hilfe des *Grant*-Befehls Zugriffsrechte an andere Benutzer weitergeben kann. Das betroffene Objekt ist dabei entweder ein Gebiet oder eine Relation, beim Ändern, Einfügen oder Referenzieren auch ein oder mehrere Attribute einer Relation. Doch dieses Granulat ist für viele Anwendungen zu grob.

Nehmen wir beispielsweise sensitive Daten wie die Personaldaten. Die Relation *Personal* aus Tab. 18 enthält unter anderem auch das Gehalt jedes Mitarbeiters. Ferner sind auch alle Daten der Vorgesetzten angegeben (als Vorgesetzte betrachten wir alle Personen, die selbst keine Vorgesetzten haben, im entsprechenden Feld also den Eintrag *Null* aufweisen). Sicherlich wird diese Relation nicht jedem Mitarbeiter der Firma zugänglich gemacht werden. Andererseits sind Teile dieser Relation (ohne die Gehaltsangabe) von informativem Wert, eventuell sogar für das gesamte Personal.

In SQL gibt es jedoch keine Möglichkeit, nur einzelne Teile (Restriktionen oder Projektionen) einer Relation für bestimmte Benutzer lesend zu sperren. Entweder ein Benutzer kann die gesamte Tabelle lesen, oder er hat keinerlei Leserechte. Doch mit Hilfe von Sichten können wir das Problem umgehen. Wir können zusätzlich zur angesprochenen Relation *Personal* eine Sicht *VPers* einführen, erzeugt etwa mit folgendem SQL-Befehl:

```
CREATE VIEW VPers AS
    SELECT Persnr, Name, Ort, Vorgesetzt
    FROM   Personal
    WHERE Vorgesetzt IS NOT NULL ;
```

Diese Sicht *VPers* enthält keine Informationen zum Gehalt der einzelnen Mitarbeiter, ebenso wurden die Daten aller Vorgesetzten weggelassen. Erlaubt der Eigentümer jetzt jedem Benutzer Leserechte auf die Sicht *VPers*, aber keine Zugriffsrechte auf die Relation *Personal*, so sind die empfindlichen Daten hinreichend geschützt, ohne dass der Allgemeinheit wichtige Informationen vorenthalten wurden. Andererseits wird der Personalleiter vermutlich direkten Zugriff auf die Tabelle *Personal* erhalten, sowohl lesend als auch schreibend. Wir erkennen, dass erst durch das Zusammenwirken der Vergabe

von Zugriffsrechten und des Einrichtens geeigneter Sichten ein optimaler Zugriffsschutz und damit Sicherheit im Datenbankbetrieb gewährleistet wird.

Wie bereits am Anfang des Kapitels erwähnt wurde, genügen die Maßnahmen des Datenbankverwaltungssystems allein nicht, um für die notwendige Sicherheit der Datenbankdaten zu sorgen. Es muss vielmehr auch sichergestellt sein, dass nicht von außerhalb auf die Daten der Datenbank zugegriffen werden kann. Hier ist eine entsprechende Verzahnung des Datenbankverwaltungssystems mit dem Betriebssystem erforderlich, um jeglichen Zugriff direkt von der Ebene des Betriebssystems auszuschließen. Dies beinhaltet auch, dass nicht mit Editoren auf die Datenbankdaten direkt zugegriffen werden kann. Gegebenenfalls sind besonders sensitive Daten zusätzlich zu verschlüsseln. Diese Verschlüsselung ist auch dann angebracht, wenn solch sensitive Informationen über öffentliche Netze transportiert werden.

Doch auch dieser Schutz muss noch nicht hinreichend sein. Bisher unentdeckte Fehler im System könnten etwa Querzugänge ermöglichen. Hier bieten Großrechnerdatenbanken meist noch einen weiteren Schutzmechanismus an: eine Audit-Einrichtung. Eine Audit-Einrichtung ist eine Protokollierung des gesamten Datenbankverkehrs. Hier werden neben dem durchgeführten Befehl die ausgeführten Änderungen, vor allem aber das Datum, die Uhrzeit, der Benutzer und die Netzadresse, von wo der Befehl abgeschickt wurde, gemerkt. Auf diese Art und Weise können unerlaubte Zugriffe zwar nicht restlos verhindert werden, doch jeder Benutzer, berechtigt oder nicht, ist durch das Protokoll immer identifizierbar. Dies allein ist schon eine sehr hohe Abschreckung vor illegalen Zugriffen. Es sei allerdings nicht verschwiegen, dass solche Audits zusätzliche, erhebliche Rechnerleistung kosten. Sie werden daher meist nur in besonders zu sichernden Datenbanken eingesetzt.

Zusammenfassend dürfen wir sagen, dass Sicherheit erst durch das Zusammenwirken auf allen Sicherheitsebenen gewährleistet werden kann, beginnend bei den physischen Gegebenheiten bis hin zum Datenbankverwaltungssystem. Die saubere Aufgabentrennung in Systemadministration und Anwender, die Vergabe entsprechender Zugriffsrechte, das eventuelle Verschlüsseln von Daten und das eventuelle Installieren einer Audit-Einrichtung gewährleisten einen optimalen Schutz aus Sicht des DBMS.

Für die Vergabe von Zugriffsrechten steht in SQL der *Grant*-Befehl zur Verfügung. Zusammen mit dem Erzeugen geeigneter Sichten können so für jeden Anwender „maßgeschneiderte“ Zugriffsrechte eingestellt werden. Weitere Informationen zu diesem wichtigen Thema Zugriffsschutz finden wir in Abschnitt 8.3 weiter unten in diesem Kapitel.

8.2 Integrität

Die Zuverlässigkeit des Datenbestands einer Datenbank und die Übereinstimmung dieser Daten mit der realen Welt hängt davon ab, wie zuverlässig die Dateneingabe in die Datenbank und die Weiterverarbeitung dieser Daten ist. Beides, sowohl die Eingabe als auch die Weiterverarbeitung, wird in der Regel über Anwenderprogramme gesteuert, wobei der Benutzer zu entsprechenden Dateneingaben aufgefordert wird.

Sowohl bei der Eingabe als auch der internen Weiterverarbeitung lauern Fehlerquellen, verursacht durch fehlerhafte Software, insbesondere aber wegen fehlerhafter Eingaben durch den Menschen. Wie leicht kann sich der Anwender vertippen oder beim Ablesen von Daten in die falsche Zeile geraten. Erkennt das Datenbankverwaltungssystem auch nur eine fehlerhafte Eingabe nicht, so ist bereits die Integrität verletzt, die Datenbank unterscheidet sich inhaltlich von seiner realen Umgebung.

Integrität heißt nicht nur, dass die Daten in der Datenbank in sich schlüssig, also konsistent sind, sondern vielmehr auch, dass diese Daten mit den realen Gegebenheiten übereinstimmen. Bezieht beispielsweise Herr Meyer ein Gehalt von 4000 DM im Monat, so muss genau dieser Betrag auch in der Datenbank abgespeichert sein.

Gerade diese Eingabeüberprüfungen sowohl beim *Insert* als auch beim *Update* werden in SQL-1 nicht und in SQL-1+ nicht vollständig unterstützt. Erst die 1992 verabschiedete SQL-2-Norm bietet hier eine Menge von Integritätskonstrukten. Diese Möglichkeiten reichen über die Überprüfung der Referenz- und Entitäts-Integrität deutlich hinaus. Wie wir bereits in den Kapiteln 2 und 3 vermerkten, ist das Thema Integrität sehr weit gespannt. Es beinhaltet im Datenbankbereich:

- die Entitäts-Integrität,
- die Referenz-Integrität,
- die Eingabeüberprüfung auf Korrektheit (semantische Integrität),
- die Zugriffsberechtigung,
- den Transaktionsbetrieb.

Die Begriffe Entitäts- und Referenz-Integrität tauchen natürlich in dieser Form nur bei relationalen Datenbanken auf. Bei den anderen Datenbankmodellen gibt es aber analoge Integritätsbetrachtungen, etwa entspricht die

Referenz-Integrität bei netzwerkartigen Datenbanken der Integrität der Setbeziehungen (siehe Abschnitt 10.4).

Die Entitäts- und Referenz-Integrität wird in SQL-2 durch die entsprechenden Spalten- und Tabellenbedingungen im *Create-Table*-Befehl unterstützt: Die Konstrukte heißen *Primary Key* und *Foreign Key … References*. Insbesondere die in SQL-2 hinzugekommenen Möglichkeiten, bei Fremdschlüsselverweisen anzugeben, wie beim Löschen oder Ändern von Referenzen die entsprechenden Fremdschlüsseldaten anzupassen sind (*On Delete* und *On Update*), tragen erheblich zur Erhaltung der Referenz-Integrität bei.

Natürlich kann auch durch entsprechende Programmiermaßnahmen die Referenz-Integrität garantiert werden, wenn obige Konstrukte von einem Datenbankhersteller nicht angeboten werden. Auch alle anderen Integritätsregeln lassen sich generell programmtechnisch auch außerhalb von SQL oder einer anderen Zugriffssprache überprüfen. Dies hat jedoch zwei Nachteile: Zum Einen ist rein logisch gesehen die Integrität eine Eigenschaft der Datenbank und gehört damit in den Bereich der Datenbankbeschreibung (DDL). Integrität wird schließlich nur dadurch garantiert, dass vor jedem Zugriff auf eine Relation überprüft wird, ob diese Relation überhaupt existiert und, falls ja, die notwendigen Zugriffsrechte vorliegen; und dann während des Zugriffs sichergestellt wird, dass keine Integritätsregeln verletzt werden.

Den zweiten Nachteil sieht vor allem der Anwendungsprogrammierer: An jeder Stelle im Programm, wo Daten geändert werden, müsste in diesem Fall eine Überprüfung auf Korrektheit der Änderung erfolgen. Der Programmcode würde sehr umfangreich, was die Fehleranfälligkeit, die Unübersichtlichkeit und auch den Wartungsaufwand erheblich erhöht.

Nachdem wir uns bereits in den Kapiteln 3, 5 und 6 ausführlich mit der Entitäts-Integrität und der referentiellen Integrität beschäftigt haben, wollen wir uns in diesem Abschnitt auf die semantische Integrität konzentrieren. Hier geht es um die Korrektheit der abgespeicherten Daten (im Vergleich zur nachzubildenden realen Umgebung).

Jede Eingabe und auch jede interne Änderung kann zu inkorrekten Daten führen. Wie leicht können Namen falsch geschrieben oder Zahlen falsch eingetippt werden. Solch inkorrekte Eingaben sind in der Praxis nicht selten und lassen sich nie ganz vermeiden. Jedes Datenbanksystem sollte jedoch möglichst viele dieser Falscheingaben abfangen. Dies geschieht optimal in mehreren Schritten:

1) Der Benutzer kann nur über Bildschirmmasken Daten eingeben.

2) Die Eingaben über die Bildschirmmaske werden sofort auf das korrekte Format überprüft (zum Beispiel werden bei Zahlenfeldern nur numerische Werte als Eingaben akzeptiert).
3) Wann immer möglich werden Daten automatisch generiert, z.B. bei neuen Personal- oder Bestellnummern.
4) Überprüfung auf Korrektheit im Datenbankanwendungsprogramm.
5) Überprüfung auf Korrektheit im Datenbankverwaltungssystem.

Die hier angegebenen Schritte sind geordnet: Wir beginnen mit Schritt 1 direkt an der Schnittstelle Mensch-Maschine und gelangen schließlich mit Schritt 5 bis ins innere des Datenbankverwaltungssystems. Von hier gibt es noch weitere interne Schritte bis hin zur Überprüfung der physischen Integrität. Dies ist allerdings Sache des Verwaltungssystems selbst.

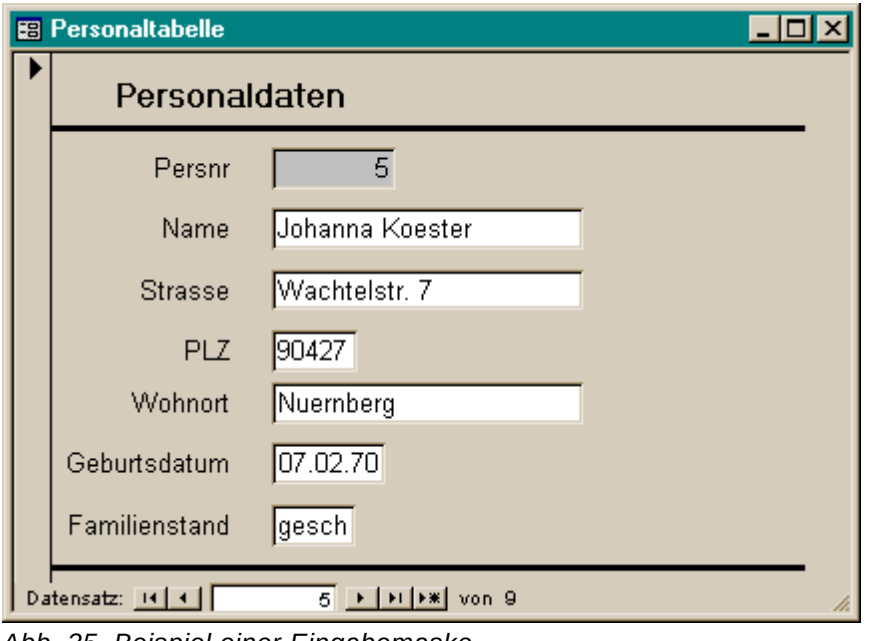


Abb. 35 Beispiel einer Eingabemaske

Der erste Schritt garantiert, dass der Benutzer nur an bestimmten Stellen der Benutzeroberfläche Daten eingeben kann. Diese Eingabefelder sind in der Länge beschränkt, und mittels des zweiten Schritts werden nur bestimmte Tasteneingaben akzeptiert. Darüberhinaus werden beim Eingeben neuer Tupel mittels des dritten Schritts einige Angaben automatisch generiert, so dass hier eine Falscheingabe unmöglich wird. Insbesondere Nummerneingaben wie Bestell-, Liefer- oder Kundennummern sollten auf diese Art und Weise vergeben

werden. Da diese Nummern meist auch Primärschlüssel sind, wird dadurch gleichzeitig die Entitäts-Integrität garantiert. In Abb. 35 ist ein einfaches Beispiel einer Eingabemaske angegeben.

Fehlerhafte Eingabedaten, die die bisherigen drei Hürden überwinden konnten, müssen jetzt in den Schritten 4 und 5 mittels Plausibilitätsüberprüfungen in den Programmen und im Datenbanksystem ausgefiltert werden. Leider können auch dann noch falsche Eingaben abgespeichert werden, sei es dass Frau Meyer versehentlich mit ‚i' geschrieben wird, oder dass ein falscher Geldbetrag eingetippt wird. Doch dass ein Mitarbeiter kaum 100 Stunden pro Woche arbeiten kann, dass ein Stundenlohn zwischen 10 und 200 DM liegt, oder dass der Lagerort einer Ware nur einer der Lagerstätten der Firma sein kann, sollte auf jeden Fall überprüft werden, um nur einige einfache Beispiele zu nennen.

Spätestens jetzt fragen wir uns, wer diese Überprüfung vornehmen soll, das Anwendungsprogramm (in Schritt 4) oder das Datenbanksystem (in Schritt 5)? Hier gibt es eine ganz allgemeine Antwort: Eigenschaften von Daten, die (über einen längeren Zeitraum) unveränderlich sind, sind grundlegende Eigenschaften. Sie sind damit Bestandteil einer Entität und damit auch einer Relation. Existiert etwa eine feste Skala von 1 bis 10 zur Mitarbeiterbewertung, so gehört diese Eigenschaft des Bewertungsfeldes in die Datenbankbeschreibung und ist als Bedingung schon zum Zeitpunkt des Erzeugens der Relation festzulegen.

Andere Daten, wie etwa der Stundenlohn, werden sich mit der Zeit ändern. Hier ist zu überlegen, ob die Überprüfung in Schritt 4 oder 5 erfolgen soll. Geschieht diese Abfrage im Anwendungsprogramm (Schritt 4), so gibt es grundsätzlich zwei Möglichkeiten: Zum Einen wird nach jeder Dateneingabe das Datum direkt im Programm nach den vorgegebenen Einschränkungen überprüft. Diese Methode hat den Nachteil, dass dadurch der Programmcode unnötig aufgebläht wird. Die zweite, wesentlich elegantere Methode ist die Triggerprogrammierung. Trigger sind (meist kleine) Unterprogramme, die bei bestimmten Ereignissen automatisch aufgerufen werden. Moderne Eingabemaskengeneratoren bedienen sich dieser Technik: Wird in einer Bildschirmmaske ein Eingabefeld geändert, so wird beim Verlassen dieses Feldes der entsprechende Trigger gestartet. Dieser überprüft die Eingabe und gibt gegebenenfalls eine Fehlermeldung mit der Aufforderung zur Korrektur aus. Eine Bildschirmmaske besteht somit aus der eigentlichen Oberfläche und den Triggerprogrammen zu den einzelnen Eingabefeldern. Praktisch alle modernen Datenbanken unterstützen diese Programmierung mittels Bildschirmmasken, so auch Oracle und MS-Access.

Kommen wir zum Datenbankverwaltungssystem zurück. Dieses kann in Schritt 5 wesentliche Eigenschaften von Entitäten selbständig und an zentraler Stelle überprüfen. Dies erspart allen Anwendungsprogrammierern eine Menge an Codierungsaufwand und verbessert die Übersicht der meist sowieso recht komplexen Programme. In SQL wird die semantische Integrität durch folgende Konstrukte unterstützt:

- Spalten- und Tabellenbedingungen (*Check*)
- Allgemeine Bedingungen (*Create Assertion*)
- Gebietsdefinitionen (*Create Domain*)
- Sichten (zusammen mit der Option *With Check Option*)

Spalten- und Tabellenbedingungen haben wir nur kurz in Kapitel 6 angesprochen. Wir wollen deshalb hier weitere Beispiele angeben:

```
ALTER TABLE  Personal  ADD  CHECK ( Gehalt  BETWEEN 2000 AND 6000 ) ;
ALTER TABLE  Personal  ADD  Zulage  INTEGER  CHECK ( Zulage >= 0 ) ;
ALTER TABLE  Personal  ADD  CHECK ( Gehalt + Zulage <= 8000 ) ;
```

Hier wurde die Relation *Personal* aus Tab. 18 durch ein Attribut *Zulage*, eine Spalten- und zwei Tabellenbedingungen erweitert. Wir erkennen, dass ab sofort das Gehalt zwischen 2000 und 6000 DM liegen muss, die Zulage nicht negativ sein darf, und Gehalt und Zulage zusammen den Betrag von 8000 DM nicht übersteigen dürfen. Es sei ausdrücklich erwähnt, dass auch Unterabfragen in den *Check*-Bedingungen erlaubt sind. Bereits in Kapitel 6 wurde darauf hingewiesen, dass sich die Möglichkeiten von Abfragen mit denjenigen der *Where*-Klausel im *Select*-Befehl decken. Soll etwa ein Artikel grundsätzlich nicht teurer verkauft werden als in der Relation *Teilestamm* eingetragen ist, so wird dies mittels folgenden SQL-Befehls garantiert:

```
ALTER TABLE Auftragsposten ADD
   CONSTRAINT ConPreis
   CHECK ( Gesamtpreis <= (SELECT  Anzahl*Preis
                           FROM    Teilestamm
                           WHERE Teilnr = Teilenr )  ) ;
```

In vielen Fällen ist es wünschenswert, solche Bedingungen nicht in Relationen aufzunehmen, sondern eigenständig zu verwalten. Dies wäre der Fall, wenn bestimmte Bedingungen logisch zusammengehören, sich aber auf mehr als eine Relation beziehen. Dazu bietet SQL den *Create-Assertion*-Befehl zum

Definieren allgemeiner Bedingungen an. Obige Tabellenbedingung würde mit Hilfe dieser allgemeinen Bedingung wie folgt lauten:

```
CREATE ASSERTION AssPreis CHECK
(   NOT EXISTS ( SELECT * FROM Auftragsposten, Teilestamm
                 WHERE Teilnr = Teilenr
                   AND   Gesamtpreis > Anzahl * Preis )    ) ;
```

Zu beachten ist die unterschiedliche Wirkung des obigen *Alter-Table*- und des *Create-Assertion*-Befehls. Im ersten Fall wird die Bedingung immer beim Ändern von Daten in der Relation *Auftragsposten* geprüft. Im zweiten Fall wird die Bedingung darüberhinaus auch getestet, wenn sich der Preis eines Artikels der Relation *Teilestamm* ändert.

Die Syntax des *Create-Assertion*-Befehls lautet ganz allgemein:

```
CREATE ASSERTION  Bedingungsname  CHECK (  Bedingung  )
```

Diese allgemeine Bedingung drückt aus, dass bestimmte Tupel nicht existieren dürfen. Die *Check*-Bedingung im *Create-Assertion*-Befehl beginnt daher in der Praxis fast immer mit dem Operator *Not Exists*. Allgemeine Bedingungen können mit dem *Drop-Assertion*-Befehl wieder gelöscht werden. Die Syntax hierfür lautet:

```
DROP ASSERTION  Bedingungsname
```

In der obigen Aufzählung der in SQL angebotenen Integritätskonstrukte tauchten auch Gebietsdefinitionen auf. Diese wurden in Kapitel 3 als die Definitionsbereiche der Attribute einer Relation vorgestellt (siehe Abb. 16 auf Seite 67). In der Praxis reicht es nicht aus, nur Datenstrukturen wie *Integer* oder *Character*-Felder zuzulassen. Darf etwa eine Firma grundsätzlich nur Niederlassungen in westeuropäischen Hauptstädten, so sollte das Gebiet für die Niederlassungsorte auch auf diese Städte eingeschränkt werden.

Mit SQL-2 gibt es die Möglichkeit, solche Gebiete selbst zu definieren. Die Definition des Gebiets aller Hauptstädte der Europäischen Union lautet beispielsweise:

```
CREATE DOMAIN EU_Hauptstadt AS CHARACTER (15)
   CHECK ( VALUE IN ('Berlin' ,'London' ,'Paris' ,'Rom' ,'Madrid' ,'Lissabon' ,
                     'Amsterdam' ,'Dublin' ,'Brüssel' ,'Luxemburg' ,'Athen' ,
                     'Kopenhagen' ,'Wien' ,'Helsinki' ,'Stockholm' )   ) ;
```

Der Bezeichner *Value* ist neu. Er ist hier der Platzhalter in der *Check*-Bedingung und steht für den in Frage kommenden Attributswert. In einem *Create-Table*-Befehl darf als Datentyp jetzt auch der Name *EU_Hauptstadt* verwendet werden, vorausgesetzt natürlich, dass man das *Usage*-Recht auf dieses Gebiet besitzt. Ganz allgemein sind Datentypen vordefinierte Typen wie *Integer* und *Character(x)* oder mittels des *Create-Domain*-Befehls definierte Typen. Die Syntax des *Create-Domain*-Befehls ist im Folgenden angegeben:

```
CREATE DOMAIN  Gebietsname  [ AS ]  Datentyp
   [   [ CONSTRAINT  Bedingungsname ]  CHECK (  Bedingung  )  ]    [ ... ]
```

Der Bezeichner *As* ist lediglich ein Füllwort und darf weggelassen werden. Soll eine Bedingung einen Namen erhalten, so ist der Bezeichner *Constraint* gefolgt von diesem Namen anzugeben. Wieder erlaubt die *Check*-Bedingung beliebige Unterabfragen. Ein Gebiet besteht also aus einem zugrundeliegenden Datentyp (dies kann auch ein bereits definiertes Gebiet sein) und aus einer Liste von Bedingungen. Diese Liste darf auch leer sein. Eine Gebietsdefinition kann mittels

```
DROP DOMAIN  Gebietsname  { RESTRICT  |  CASCADE }
```

auch wieder entfernt werden. Wird die Option *Restrict* angegeben, so misslingt das Entfernen des Gebiets, wenn noch Verweise auf dieses Gebiet existieren. Mit der Option *Cascade* werden auch Attribute von Sichten und Tabellen- oder Spaltenbedingungen gelöscht, die dieses Gebiet referenzieren. Dieses automatische Löschen gilt jedoch nicht für Tabellenattribute, die diesen Gebietsnamen verwenden. In diesem Fall werden die *Check*-Bedingungen der Gebietsdefinition direkt in die betroffenen Basisrelationen übernommen. Das Integritätsverhalten von Basisrelationen wird also nicht durch den *Drop-Domain*-Befehl beeinflusst.

Soll eine *Check*-Bedingung einer Gebietsdefinition gelöscht werden, oder sollen weitere Bedingungen hinzugefügt werden, so steht dazu der *Alter-Domain*-Befehl zur Verfügung. Die Syntax lautet:

```
ALTER DOMAIN  Gebietsname
        ADD  [ CONSTRAINT  Bedingungsname  ]  CHECK (  Bedingung  )
    |   DROP  Bedingungsname
```

Das Löschen einer Bedingung führt dazu, dass bei allen Attributen der Datenbank, die auf diesem Gebiet basieren, diese Bedingung nicht mehr überprüft wird. Umgekehrt gibt es beim Hinzufügen zusätzliche Überprüfungen.

Viele Datenbankhersteller bieten noch nicht die Vielfalt der in SQL-2 unterstützten und hier behandelten Integritätskonstrukte an. Weder MS-Access noch Oracle kennen allgemeine Bedingungen (Create Assertion) und Gebiete (Create Domain). Weiter kennt MS-Access keine *Check*-Bedingungen, aber auch in Oracle sind in den Tabellenbedingungen keine Unterabfragen erlaubt. Eine komplexe *Check*-Bedingung muss daher bis heute entweder im Anwendungsprogramm selbst programmiert werden oder in SQL mittels einer Sicht realisiert werden, die die *With-Check-Option*-Klausel verwendet. Unser Beispiel der Verkaufspreisbeschränkung würde dazu lauten:

```
CREATE VIEW VAuftragsposten AS
   SELECT * FROM Auftragsposten
   WHERE Gesamtpreis <= ( SELECT  Anzahl * Preis
                          FROM    Teilestamm
                          WHERE Teilnr = Teilenr  )
   WITH CHECK OPTION ;
```

In der Sicht *VAuftragsposten* werden jetzt alle Änderungen abgewiesen, die die *Where*-Bedingung nicht erfüllen. Jetzt muss nur noch sichergestellt werden, dass auf die Relation *Auftragsposten* nur über die Sicht *VAuftragsposten* zugegriffen werden darf. Die Methode der Verwendung von Tabellenbedingungen und Gebieten zur Unterstützung der Integrität ist natürlich eleganter und sollte, falls unterstützt, immer vorgezogen werden.

Die einzelnen Integritätskonstrukte haben also folgende Aufgabe:

Wichtig

☞ Integritätskonstrukte beschreiben, für welches Attribut einer Relation bei welchen Operationen (*Insert, Update, Delete*) welche Regeln zu beachten sind, und welche Aktion zu geschehen hat, wenn eine Regel verletzt wird!

In der Aufzählung der Integritätsthemen am Anfang dieses Abschnitts wurde auch die Transaktion erwähnt. Transaktionen sind unerlässlich für die Integrität einer Datenbank, da nur so die Konsistenz der Daten gesichert werden kann: Wird nämlich eine Verletzung einer Integritätsregel festgestellt, so muss in der Regel eine Ausnahmebehandlung gestartet werden. Unter Um-

ständen können weitere geplante Änderungen nicht mehr ausgeführt werden, so dass aus Konsistenzgründen auch bereits erfolgte Änderungen zurückgesetzt werden müssen. Und dies ist im Allgemeinen nur im Transaktionsbetrieb möglich. Es ist in einigen Fällen auch angebracht, vor Transaktionsende nochmals global zu überprüfen, ob die erfolgten Änderungen die Integrität der Datenbank nicht verletzt haben. Wenn doch, so ist ein Rollback erforderlich.

Wir erkennen, dass eine Transaktion die kleinste Einheit sowohl für Recovery, als auch für Concurrency und Integrität darstellt. Da jeder einzelne dieser drei Begriffe aus einem korrekt laufenden Datenbankbetrieb nicht wegzudenken ist, ist ein (korrekter) Datenbankbetrieb zwangsläufig immer auch ein Transaktionsbetrieb.

8.3 Ergänzungen zum Relationenmodell

Dieses Kapitel zeigte uns bisher, dass es nicht nur wichtig ist, eine Datenbank aus der gegebenen realen Welt optimal zu entwerfen. Vielmehr spielen beim Gesamtdesign auch die Zugriffe durch die späteren Benutzer eine wichtige Rolle. Bei größeren Datenbanken kann leicht der Überblick verloren gehen, wenn sowohl die Datenbankstruktur als auch die Sicht des Benutzers berücksichtigt werden sollen. Dem lässt sich jedoch abhelfen. Genaugenommen besteht eine funktionierende Datenbank nämlich aus vier Schichten:

- der physischen Schicht,
- der logischen Datenbankschicht,
- den logischen Benutzerschichten,
- den Oberflächen für die Zugriffe auf die Datenbank.

Diese vier Schichten sind sauber voneinander entkoppelt. Die physische Schicht liegt, zumindest bei relationalen Datenbanken, allein in der Verantwortung des Datenbankanbieters. Für die Oberflächen zeichnet andererseits der Datenbankprogrammierer verantwortlich. Für den Datenbankdesigner verbleiben die beiden dazwischenliegenden Schichten, wobei auch diese beiden Schichten durch eine exakt definierbare Schnittstelle getrennt werden können. Dies wollen wir in den nächsten Absätzen aufzeigen.

Die logische Datenbankschicht ist die Betrachtungsweise, die sich direkt aus dem Datenbankentwurf, etwa mittels eines Entity-Relationship-Modells ergibt. Hier werden die einzelnen Relationen, deren Beziehungen zueinander und

weitere aus der realen Welt vorgegebene Bedingungen festgelegt. Dies führt schließlich zu *Create-Table-*, *Create-Assertion-* und *Create-Domain*-Befehlen. Die *Create-Table*-Befehle enthalten dabei neben den Attributsangaben auch alle erforderlichen Spalten- und Tabellenbedingungen.

Insbesondere sei hier nochmals darauf hingewiesen, dass mit *Check*-Bedingungen nicht gegeizt werden sollte. Alle Bedingungen, die aus der realen Welt abgelesen werden können, sollten bereits in das Design mit aufgenommen werden. Nur so wird garantiert, dass alle Überprüfungen global und damit einheitlich verwaltet werden. Desweiteren wird dadurch der Anwendungsprogrammierer erheblich entlastet.

Auf diese logische Datenbankschicht setzen nun die einzelnen logischen Benutzerschichten auf. Diese werden mit Hilfe von Sichten und dem Gewähren von Zugriffsrechten auf diese Sichten erzeugt. Dabei ist es möglich und auch sinnvoll, jedem Benutzer seine eigene Sicht zu erstellen. Unter Benutzer verstehen wir dabei nicht jeden einzelnen Anwender, sondern jeweils Gruppen von Anwendern (z.B. *Mitarbeiter*, *Leiter*, *Verwalter* oder *Gast*).

Eine Datenbank kann als Folge dieser Überlegungen durchaus mehr Sichten als Relationen besitzen. Wir sehen aber auch, dass der Datenbankdesigner die logische Datenbankschicht unabhängig von Benutzerzwängen entwerfen kann. Erst nach dem Entwurf der logischen Schicht beschäftigt er sich mit den Benutzerschichten.

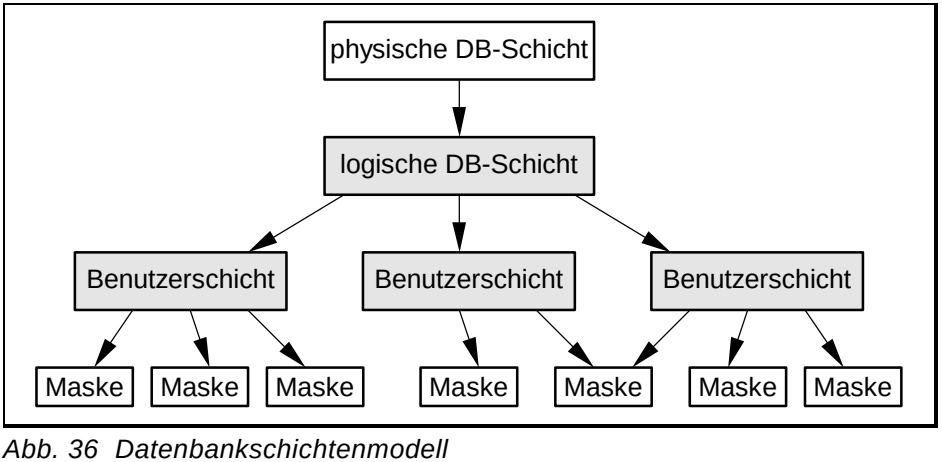


Abb. 36 Datenbankschichtenmodell

Aus Abb. 36 erkennen wir schließlich sehr schön, dass es nur eine physische und eine logische Datenbankschicht gibt. Ganz im Gegensatz dazu bauen sich aus diesen Schichten hierarchisch mehrere Benutzerschichten auf. Und

auf jede einzelne dieser Schichten setzt wiederum eine Menge von Eingabemasken auf.

8.4 Zusammenfassung

In diesem Kapitel haben wir kennengelernt, dass mit Hilfe des *Grant*-Befehls Zugriffsrechte auf die einzelnen Basisrelationen, Sichten und Gebiete vergeben werden. Diese Rechte können getrennt nach Lese- und den verschiedenen Schreibzugriffen eingestellt werden. Auch können diese Rechte gegebenenfalls an Dritte weitergereicht werden. Wir haben auch gesehen, dass Sichten und Zugriffsrechte sehr eng miteinander verknüpft sind. Schließlich wurde auch der *Revoke*-Befehl behandelt, mit dem vorher mit dem *Grant*-Befehl vergebene Rechte wieder entzogen werden können.

Im zweiten Abschnitt wurden hauptsächlich Möglichkeiten zur Verbesserung der semantischen Integrität besprochen. Dies erreichen wir mittels Eingabemasken und Trigger von der Anwendungsebene aus. Fast noch wichtiger sind jedoch die DDL-Befehle zur Unterstützung der Integrität. Dies sind vor allem die *Check*-Bedingung, die allgemeine Bedingung (*Create-Assertion*-Befehl) und die Gebietsdefinition. Wir haben gesehen, dass alle Einschränkungen, die bereits zum Zeitpunkt des Erzeugens der Datenbank bekannt sind, auch zu diesem Zeitpunkt mit anzugeben sind. Nur die sich im laufenden Betrieb ergebenden Einschränkungen werden mit Eingabemasken und Trigger abgefangen.

Die vielen Bedingungen zur Verbesserung der Integrität und die Vergabe der Zugriffsrechte zusammen mit Sichtdeklarationen führen beim Erzeugen einer Datenbank zu einer Vielzahl von zusätzlichen Befehlen. Aus diesem Grund empfiehlt es sich, zusätzlich zum Relationenmodell Benutzerschichten einzuführen. Die entsprechende Vorgehensweise haben wir im letzten Abschnitt kennengelernt.

8.5 Übungsaufgaben

Alle hier angesprochenen Relationen beziehen sich auf Relationen der Beispieldatenbank *Radl* aus Anhang A.

1) In einem sicheren Datenbankverwaltungssystem muss vor jedem Zugriff auf eine Relation überprüft werden, ob der Benutzer die erforderlichen Zugriffsrechte

besitzt. Ein Datenbankzugriff besteht demnach faktisch aus zwei Zugriffen (Sicherheitsüberprüfung und eigentlicher Zugriff). Stimmt diese Aussage wirklich?

2) Schreiben Sie einen Befehl, der dem Benutzer *Gast* Änderungsrechte auf die Attribute *Bestand*, *Reserviert* und *Bestellt* der Relation *Lager* und Leserechte auf die gesamte Relation einräumt.
3) Entziehen Sie dem Benutzer *Gast* die unter 2) gewährten Rechte wieder.
4) Schreiben Sie alle notwendigen Befehle, damit der Benutzer *Gast* nur Leserechte auf die Attribute *Teilenr*, *Lagerort* und *Bestand* der Relation *Lager* bekommt. Weiter darf er Tupel dieser Relation nicht sehen, falls Mindestbestand plus reservierte Teile größer als der tatsächliche Bestand sind. Diese Rechte darf der Benutzer *Gast* auch weiterreichen.
5) Der Benutzer *Gast* erhält Leserecht auf die Relation *Lieferant*, jedoch nicht auf das Attribut *Sperre* und nur für in Regensburg ansässige Lieferanten. Schreiben Sie die erforderlichen Befehle.
6) Um die Integrität zu optimieren, sollen die Attribute *GebDatum*, *Stand*, *Gehalt* und *Beurteilung* der Relation *Personal* auf zulässige Werte überprüft werden. Es ist bekannt, dass alle Mitarbeiter zwischen 15 und 65 Jahre alt sind, entweder ledig, verheiratet, geschieden oder verwitwet sind, das Gehalt zwischen 800 und 10000 DM liegt und die Beurteilung entweder *Null* oder einen Wert zwischen 1 und 10 besitzt. Fügen Sie diese Bedingungen mittels eines *Alter-Table*-Befehls hinzu, wobei sicherzustellen ist, dass diese Bedingungen, falls gewünscht, auch wieder entfernt werden können (*Constraint*!). Das heutige Datum liefert SQL in der Konstanten CURRENT_DATE zurück. Nehmen Sie an, dass ein Jahr 365 Tage besitzt.
7) Schreiben Sie die unter 6) angegebenen Bedingungen mittels eines *Create-Assertion*-Befehls.
8) Auch mit SQL-1 muss auf die unter 6) angegebenen Bedingungen und deren automatische Überprüfung durch das DBMS nicht verzichtet werden. Was muss dazu berücksichtigt werden und wie lauten die dazu notwendigen SQL-Befehle?
9) Im Attribut *Aufgabe* der Relation *Personal* gibt es nur eine beschränkte Anzahl von möglichen Aufgaben. Definieren Sie ein Gebiet *Berufsbezeichnung*, das eine Ansammlung von möglichen Berufen enthält.
10) Es erweist sich als sinnvoll, die unter 9) angegebenen Berufsbezeichnungen noch zu ergänzen. Bitte fügen Sie noch die Bezeichnungen *Lackierer* und *Schlosser* hinzu.

9 Eingebettetes SQL

Bei SQL handelt es sich wie bei vielen Sprachen der vierten Generation um keine vollständige Sprache: Die Sprache SQL umfasst keine Kontrollanweisungen wie Schleifen- oder Verzweigungsanweisungen und auch keine Datenstrukturen. Vielmehr beschränkt sich SQL auf die Beschreibung von und den Zugriff auf Datenbanken. Wir können folglich allein mit SQL-Mitteln keine Programme schreiben, wir benötigen dazu eine „Wirts“-Programmiersprache, in die SQL eingebettet ist. Der Vorteil dieser Einbettung ist, dass der Programmierer seine bevorzugte Sprache weiterverwenden kann. Nur zum Zweck des Zugriffs auf die Datenbank selbst greift er auf SQL-Befehle zurück.

Leider wurde in SQL-1 versäumt, diese Einbettung von SQL in eine Programmierumgebung eindeutig festzulegen. Dies wurde jetzt zwar von SQL-2 nachgeholt, viele SQL-Dialekte (z.B. MS-Access) halten sich aber in diesem Punkt nicht an den neuen Standard. Wir werden auf Besonderheiten in MS-Access am Ende dieses Kapitels kurz eingehen.

Dieses Kapitel beginnt mit den ersten Schritten zur Einbettung von SQL in die Wirtssprache C++ (wir werden in diesem Kapitel ausschließlich die Sprache C++ verwenden, die Einbettung in andere Sprachen geschieht völlig analog). Danach werden wir den Gebrauch von Transaktionen in Programmen kennenlernen, um dann in einem eigenen Abschnitt zum mächtigsten SQL-Abfragewerkzeug zu gelangen, den SQL-Cursorn. Wir beenden dieses Kapitel mit Hinweisen auf Besonderheiten in MS-Access.

9.1 Einbettung von SQL in C++

Wir zeigen in diesem Abschnitt die grundsätzliche Vorgehensweise der Einbettung von SQL in eine Wirtssprache, hier C++. Im wesentlichen ist dieses Vorgehen für alle SQL-Implementierungen und Programmiersprachen gleich. Wir orientieren uns hier beispielhaft an Oracle und C++.

Unser fertiges Datenbankprogramm wird sich kaum von einem gewöhnlichen C++-Programm unterscheiden, insbesondere besitzt es die Funktion *main()*, Variablen und Datenstrukturen und vermutlich weitere Funktionen. Soll in diesem Programm auf eine Datenbank zugegriffen werden, so werden

die bereits bekannten SQL-Befehle anstelle von C++-Anweisungen verwendet. Zur eindeutigen Unterscheidung der SQL-Befehle vom C++-Code werden diesen Befehlen die Worte „EXEC SQL“ vorangestellt. Diese Worte dürfen anderweitig im Programm nicht verwendet werden, so dass dadurch die SQL-Befehle eindeutig von C++-Anweisungen getrennt sind.

Diese Mischung aus C++- und SQL-Befehlen wird nun zunächst von einem Präcompiler, bei Oracle *Pro*C* genannt, vorübersetzt. Dieser Präcompiler durchsucht das gesamte Quellprogramm, überprüft die Korrektheit im SQL-Teil und ersetzt die SQL-Befehle durch entsprechende C++-Unterprogrammaufrufe. Dieses nun reine C++-Programm wird jetzt im nächsten Schritt von einem gewöhnlichen C++-Compiler übersetzt. Zuletzt werden durch den Linker die erforderlichen Module, insbesondere auch die SQL-Module, hinzugebunden. Dazu werden von den Datenbankherstellern umfangreiche SQL-Bibliotheken zur Verfügung gestellt. Nach diesen Schritten erhalten wir ein komplett übersetztes, ablauffähiges Programm. Wir können diese Vorgehensweise an Hand der Abb. 37 nachvollziehen. Selbstverständlich lässt sich diese Erstellung auch automatisieren, unter UNIX etwa mittels eines geeigneten Makefiles.

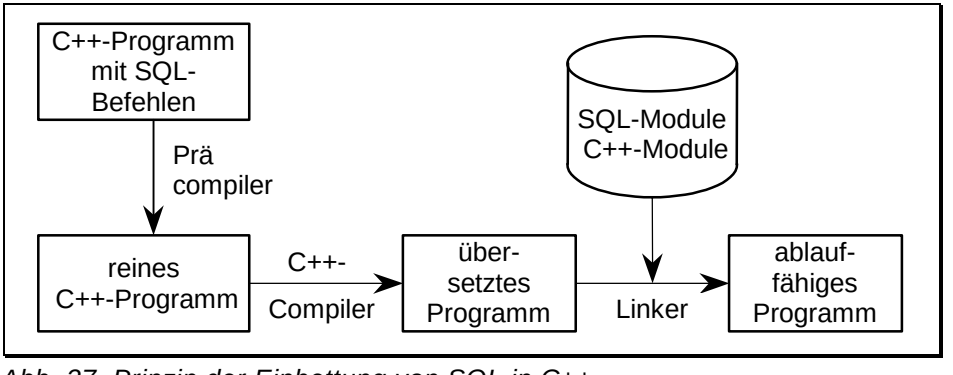


Abb. 37 Prinzip der Einbettung von SQL in C++

9.2 Programmieren in C++ mit eingebettetem SQL

Beim Arbeiten mit C++ und SQL benötigen wir Bezüge zwischen beiden Sprachen, die wir mittels Variablen und Parameter herstellen. Hierzu müssen allerdings gewisse Regeln eingehalten werden. Zunächst müssen alle im C++-Teil vorkommenden Variablen wie gewohnt im C++-Programm deklariert

werden. Sollen C++-Variablen auch in den SQL-Befehlen verwendet werden, so müssen diese explizit auch SQL bekanntgemacht werden. Dies geschieht mittels einer sogenannten *Declare Section*. Dieser SQL-Deklarationsabschnitt sieht wie folgt aus:

```
EXEC SQL BEGIN DECLARE SECTION ;
  // Deklaration aller C++-Variablen, die auch in SQL-Befehlen verwendet werden
EXEC SQL END DECLARE SECTION ;
```

Die so deklarierten C++-Variablen sind auch in allen SQL-Befehlen sichtbar, die dieser *Declare-Section* folgen. Sie dürfen in SQL-Befehlen überall dort verwendet werden, wo variable Werte erlaubt sind. Zur Unterscheidung zu Bezeichern für Attribute und Relationen sind diesen C++-Variablen innerhalb von SQL-Befehlen Doppelpunkte voranzustellen.

Die Verwendung von Variablen ist besonders wichtig in Verbindung mit dem Suchbefehl *Select*. In der in Kapitel 4 behandelten Form können jedoch Ergebnisse von *Select*-Befehlen nicht in Variablen gespeichert werden. Wir müssen dazu eine entsprechende Erweiterung für eingebettetes SQL verwenden. Wir wollen diese Erweiterung gleich kennenlernen:

```
SELECT [ ALL | DISTINCT ]    Spaltenauswahlliste
    INTO     Variable [ , ... ]
    FROM  . . .
```

Hier handelt es sich um den bereits bekannten *Select*-Befehl, dem die Klausel *Into* hinzugefügt wurde. Diese Klausel steht zwischen der *Select*- und *From*-Klausel. Mit dem Aufruf dieses neuen Befehls werden die Ergebnisse nicht mehr auf Bildschirm ausgegeben, sondern in den angegebenen Variablen gespeichert. Es versteht sich von selbst, dass die Auswahlliste und die Variablenliste in der *Into*-Klausel aus gleich vielen Elementen mit zueinander kompatiblen Datentypen bestehen müssen. Weiter ist vorauszusetzen, dass dieser *Select*-Befehl nur genau eine Reihe auswählt, die dann in der Variablenliste hinterlegt wird. Betrachten wir dazu ein kleines Beispiel. Wir wollen wissen, wo Frau Forster wohnt. Dazu durchsuchen wir die Relation *Personal* aus Tab. 18 und speichern das Ergebnis in der Variablen *Wohnort*. Ausschnittsweise lautet dann das Programm:

```
EXEC SQL BEGIN DECLARE SECTION ;
        char Wohnort [15];
        char Name [25];
EXEC SQL END DECLARE SECTION ;
```

```
strcpy (Name, "%Forster");          // Suchstring
EXEC SQL  SELECT Ort
          INTO :Wohnort
          FROM Personal
          WHERE Name LIKE :Name ;
```

Zu beachten ist in diesem Beispiel der Unterschied zwischen dem Attribut *Name* und der Variablen gleichen Namens. Die Variable unterscheiden wir durch den führenden Doppelpunkt. *Wohnort* ist eine weitere Variable, die in der *Into*-Klausel verwendet wurde.

Obiges Beispiel arbeitet nur dann einwandfrei, wenn bei der Suche genau einmal der Name *Forster* gefunden wird. Laut SQL ist es ein Fehler, die *Into*-Klausel zu verwenden, wenn der Name *Forster* nicht oder mehr als einmal erscheint. Viele SQL-Implementierungen, so auch Oracle, liefern bei Mehrfachvorkommnissen zwar das erste oder letzte gefundene Tupel zurück, doch dies ist natürlich ein Zufallsergebnis. Diese sogenannten „Einreihen"-*Select*-Befehle sollten also nur verwendet werden, wenn maximal ein Ergebnis erwartet wird, etwa beim Suchen nach einem bestimmten Primärschlüsselwert. Um festzustellen, ob überhaupt ein Ergebnis zurückgeliefert wurde, müssen wir noch die Ausnahmebehandlung in SQL kennenlernen.

Im eingebetteten SQL liefert jeder ausgeführte SQL-Aufruf einen Fehlercode zurück, der in der Ganzzahl-Variablen *Sqlcode* abgelegt wird, seit SQL-2 auch in der aus 5 Zeichen bestehenden Zeichenkette *Sqlstate*. Mindestens eine dieser beiden Variablen muss daher in der Wirtssprache definiert werden. Diese beiden Variablen werden mit jedem ausgeführten SQL-Befehl neu gesetzt. Durch Abprüfen einer dieser Variablen direkt nach einem SQL-Befehl kann somit festgestellt werden, ob der Befehl erfolgreich abgearbeitet wurde.

Tab. 46 Fehlercodes der Variable SQLCODE

Code	Ursache
0	Erfolgreiche Beendigung
100	Daten nicht gefunden
< 0	Fehler

Die in SQL definierten drei Werte der Variable *Sqlcode* finden wir in Tab. 46. Der Wert 100 wird immer dann zurückgeliefert, wenn eine Abfrage keine Ergebnisse lieferte. Bei Fehlern wird in *Sqlcode* ein negativer Wert zurückgegeben. Dieser Wert ist im Standard nicht festgelegt. Meist werden den einzelnen Fehlern unterschiedliche negative Werte zugeordnet, was die Fehlersuche vereinfacht.

Tab. 47 Fehlercodes der Variable SQLSTATE (Auswahl)

Code	Ursache
'00'	Erfolgreiche Beendigung
'01'	Warnung
'02'	Daten nicht gefunden
'08'	Verbindungsaufbau-Fehler
'0A'	Merkmal wird nicht unterstützt
'22'	Datenfehler (z.B. Division durch Null)
'23'	(Tabellen/Spalten-)Bedingung ist verletzt
'24'	Ungültiger Cursor-Status
'25'	Ungültiger Transaktions-Status
'2A'	SQL-Syntax- oder Zugriffsfehler
'2B'	Abhängiges Privileg existiert
'2D'	Nichterlaubte Transaktionsbeendigung
'34'	Ungültiger Cursorname
'3D'	Ungültiger Katalogname
'3F'	Ungültiger Schemaname
'40'	Rollback
'42'	Syntax- oder Zugriffsfehler
'44'	Check-Bedingung ist verletzt

Die Variable *Sqlcode* wird seit SQL-2 von der Variablen *Sqlstate* verdrängt, deren erste zwei Zeichen den Fehlercode und deren weitere drei Zeichen den dazugehörigen Subcode angeben. Der Code ‚00' bezeichnet das erfolgreiche Abarbeiten des Befehls. In diesem Fall ist der Subcode ebenfalls ‚000'. In allen anderen Fällen enthält der Code von Null abweichende Ziffern und/oder Großbuchstaben. Eine kleine Auswahl der möglichen Fehlercodes ist in Tab. 47 zusammengefasst. Eine Tabelle aller in SQL definierten Fehlercodes einschließlich Subcodes finden wir in [MeSi93]. Wir erkennen aus Tab. 47, dass in der Variablen *Sqlstate* die Fehlercodes in Abhängigkeit von der Fehlerursache genau festgelegt wurden, ganz im Gegensatz zur Variable *Sqlcode*. Im obigen Programmbeispiel empfiehlt sich direkt im Anschluss an den *Select*-Befehl folgende Programmanweisung:

```
if ( strcmp (SQLSTATE,"00000") == 0 )              //  oder: if ( SQLCODE == 0 )
    cout << "Der Wohnort lautet: " << Wohnort << endl;
else   if ( strcmp (SQLSTATE,"02000") == 0 )      //  oder: if ( SQLCODE == 100 )
    cout << "Der Name " << Name << " ist nicht in der Relation enthalten\n";
else
    cout << "Fehler in Select-Befehl; Fehlercode: " << SQLSTATE << endl;
```

Wir setzen in diesem Fall natürlich voraus, dass die Variable *Sqlstate* (bzw. *Sqlcode*) im Programm korrekt definiert wurde. Da diese Variable nicht innerhalb von SQL-Befehlen verwendet wird, muss diese Deklaration auch nicht notwendigerweise innerhalb einer *Declare Section* erfolgen. In Oracle muss eigenartigerweise die Variable *Sqlstate*, nicht jedoch *Sqlcode*, doch innerhalb der Declare Section definiert werden.

Wir benötigen jetzt noch einen kleinen Schritt, um ein voll lauffähiges Programm zu erhalten. Wir müssen dem Programm mitteilen, mit welchem SQL-Server wir in Kontakt treten wollen. Bisher wurde immer stillschweigend davon ausgegangen, dass wir bereits an einem SQL-Server eingeloggt sind und dort dann auch alle SQL-Befehle ausführen können. Diese Voraussetzung gilt selbstverständlich nicht für ein eigenständiges Programm. Dieses wird von der Betriebssystemebene aus gestartet und muss daher selbständig die Verbindung zum gewünschten SQL-Server herstellen. Den dazu benötigten *Connect*-Befehl zum Verbindungsaufbau haben wir bereits in Kapitel 6 kennengelernt.

Die einfachste Art des Verbindungsaufbaus ist der Aufruf des Default-Servers. Wir werden diesen Aufbau im folgenden C++-Programm verwenden. Dieses komplette Programm erhöht das Gehalt aller Mitarbeiter um 2%. Vorab wird zunächst die monatliche Mehrbelastung der Firma errechnet und ausgegeben. Erst nach einer Sicherheitsabfrage wird diese Erhöhung schließlich durchgeführt. Ausgangsbasis ist die Relation *Personal* aus der Beispieldatenbank *Radl* im Anhang A.

```
#include <iostream.h>
#include <string.h>

EXEC SQL  BEGIN DECLARE SECTION ;
                  float mehrbetrag ;
                  char SQLSTATE [6] ;          // 6 Zeichen wegen '\0' !
                        // SQLSTATE (oder SQLCODE) muss definiert werden
EXEC SQL  END DECLARE SECTION ;

int main ( )
{   char abfrage;

    EXEC SQL  CONNECT TO DEFAULT ;     // Einloggen in den Default-Server
    if ( strcmp (SQLSTATE,"00000") )
    {   cout << "Fehler beim Oeffnen der DB, Code: " << SQLSTATE;    return 0;
    }
    cout << "Programm zum Erhöhen des Gehalts um 2%:\n\n";

    // Mehrbelastung fuer den Betrieb ermitteln:
    EXEC SQL  SELECT Sum (0.02 * Gehalt)
                  INTO     :mehrbetrag
                  FROM     Personal ;
```

```
    if ( strcmp (SQLSTATE,"00000") )
    {   cout << "Fehler beim Lesen von Personal, Fehlercode: " << SQLSTATE;
        return 0;
    }

        // Ausgeben des Mehrbetrags. Abfrage, ob Erhoehung durchzufuehren ist:
    cout << "Die Mehrbelastung betraegt " << mehrbetrag << " pro Monat.\n"
           << "Soll die Gehaltserhoehung durchgefuehrt werden (j/n)? ";
    cin >> abfrage;

    if (abfrage != 'j')
        cout << "keine Gehaltserhoehung.\n";

    else                              // Gehaltserhoehung erfolgt:
    {   EXEC SQL  UPDATE Personal
                      SET Gehalt = 1.02 * Gehalt ;
        if ( strcmp (SQLSTATE,"00000") != 0 )
        {   cout << "Fehler beim Update, Fehlercode: " << SQLSTATE ;  return 0;
        }
        cout << "Gehaltserhoehung wurde durchgefuehrt.\n ";
    }

    // Änderungen in der Datenbank endgültig festlegen und beenden:
    EXEC SQL  COMMIT WORK ;
    EXEC SQL  DISCONNECT CURRENT ;          // Ausloggen aus Datenbank
    cout << "Alle Gehälter wurden um 2% erhöht.\n\n";

    return 0;
}
```

Wir erkennen an diesem Programm, dass wir nach jedem SQL-Befehl den aktuellen SQL-Status abfragten und gegebenenfalls das Programm abbrachen. Beispielsweise wird ein Zugriff auf die Relation *Personal* misslingen, wenn wir keine hinreichenden Zugriffsrechte auf diese Relation besitzen. In diesem Fall wird eine entsprechende Meldung ausgegeben. Weiter finden wir am Programmende vor dem Abbau der Verbindung zum SQL-Server noch den Befehl *Commit Work*. Erst dieser Befehl garantiert die durchgeführte Änderung. Nach diesem Befehl kann die Gehaltserhöhung nur mittels eines neuen weiteren *Update*-Befehls rückgängig gemacht werden. Natürlich sollten alle unsere Programme immer als Transaktionen ablaufen (siehe auch den nächsten Abschnitt und die Ausführungen zu Transaktionen in Kapitel 7).

Hätten wir in unserem Beispiel die Transaktion nicht abgeschlossen, so wäre es herstellerabhängig, ob nach dem Beenden eines Programms implizit ein *Commit Work* oder ein *Rollback Work* durchgeführt wird. Wir wollen dies nicht dem Zufall überlassen und beenden immer unser Programm mit dem entsprechenden Transaktionsbefehl. Im Falle des Abbruchs wegen eines auf-

getretenen Fehlers war ein explizites Zurücksetzen der Transaktion nicht erforderlich, wir hatten ja noch nichts geändert.

Ein explizites Abbrechen der Verbindung zum SQL-Server wäre nicht zwingend notwendig. Dies wird bei Programmende automatisch geschehen. Trotzdem ist es ein guter Programmierstil, den Verbindungsabbau explizit durchzuführen.

Es kann recht lästig sein, immer nach jedem SQL-Befehl den SQL-Status abzufragen. SQL unterstützt daher eine allgemeine Fehlerbehandlung. Wir wollen auch auf diese Möglichkeit eingehen. Der SQL-Fehlerbehandlungsbefehl lautet *Whenever*, die Syntax ist im Folgenden angegeben:

```
WHENEVER   { SQLERROR | NOT FOUND }
           { CONTINUE | GOTO  Label }
```

Dieser eingebettete SQL-Befehl wirkt auf jeden folgenden SQL-Befehl, und zwar bis zum jeweils nächsten *Whenever*-Befehl mit der gleichen Bedingung (*Sqlerror* bzw. *Not Found*). Die Aktion *Continue* bewirkt, dass ab sofort keine automatische Fehlerbehandlung mehr erfolgt. In diesem Fall sollten Fehler mittels *Sqlcode* oder *Sqlstate* direkt nach den einzelnen SQL-Befehlen selbst abgefangen werden. Ist hingegen die Aktion *Goto* gesetzt, so wird zur Marke *Label* des C++-Programms gesprungen. Ist die Bedingung *Not Found* angegeben, so wird bei nicht gefundenen Daten die entsprechende Aktion gestartet. Dies entspricht dem Fall *Sqlcode*=100 bzw. *Sqlstate*='02000'. Mit der Bedingung *Sqlerror* werden alle anderen Fehler abgefangen (*Sqlstate* siehe Tab. 47 ab Zeile 4, *Sqlcode* < 0).

An der angegebenen Labelposition kann jetzt zentral auf Fehlermeldungen reagiert werden. Dieses Vorgehen besitzt den großen Vorteil, dass der normale Programmcode nicht mit umfangreichen Fehlerbehandlungen überfrachtet wird. Damit bleibt der eigentliche Programmablauf übersichtlich und folglich leichter wartbar.

Zugegebenermaßen entspricht die *Goto*-Sprunganweisung im *Whenever*-Befehl nicht den heutigen Vorstellungen einer strukturierten Programmierung. Doch dies war der Kompromiss für die Einbettbarkeit der Sprache SQL in praktisch alle Programmiersprachen. Viele Datenbankhersteller, so auch Oracle, bieten allerdings zusätzlich die Möglichkeit, mit Hilfe des *Whenever*-Befehls in ein Unterprogramm zu springen. Beispiele zum *Whenever*-Befehl finden wir im folgenden Absatz und in Anhang A.

9.3 Transaktionsbetrieb mit eingebettetem SQL

Wie bereits im Kapitel zur Recovery und Concurrency erwähnt wurde, können einzelne SQL-Befehle zur Inkonsistenz einer Datenbank führen. Erst eine gewisse Anzahl von Änderungszugriffen auf eine Datenbank gewährleistet wieder die Konsistenz. Wir sprechen von Transaktionen. Eine moderne Zugriffssprache wie SQL muss dieses Verhalten natürlich unterstützen. Hierzu gibt es, wie bereits mehrfach erwähnt, in SQL folgende beiden Befehle:

```
COMMIT  [ WORK ]
ROLLBACK  [ WORK ]
```

Alle Änderungen in der Datenbank sind nur vorläufig. Erst mittels des SQL-Befehls *Commit Work* werden diese Änderungen endgültig gespeichert. Bei der Datenbankprogrammierung ist demnach dringend zu beachten, an Konsistenzpunkten jeweils den *Commit-Work*-Befehl zu setzen. Weiter wird im Fehlerfall die gerade laufende Transaktion zurückgesetzt. Dies geschieht mit dem Befehl *Rollback Work*. Alle Mutationen seit dem letzten *Commit Work* werden damit zurückgenommen. Betrachten wir zur Vertiefung noch einen Ausschnitt aus einem kleinen Programm, das den Familienstand von Frau Wolters (Personalnummer 8) in der Radl-Datenbank ändert:

```
    ...
    EXEC SQL WHENEVER SQLERROR GOTO fehler;
    cout << "Geben Sie den neuen Familienstand an: "; cin >> stand;

    EXEC SQL UPDATE Personal
                 SET    Stand = :stand
                 WHERE  PersNr = 8;
    EXEC SQL COMMIT WORK;          // Sichern seit letztem Commit
    return 0;

fehler:
    cout << "Fehler beim Update in der Relation Personal\n";
    EXEC SQL ROLLBACK WORK;       // Rücknahme seit letztem Commit
    ...
```

Sollte der *Update*-Befehl misslungen sein, etwa weil ein inkorrekter Familienstand eingegeben wurde, so wird zur Sprungmarke *fehler* verzweigt, und nach einer Ausgabemeldung werden alle seit dem letzten *Commit Work* durchgeführten Änderungen zurückgesetzt.

9.4 SQL-Cursor

Wir haben im vorletzten Abschnitt den um die *Into*-Klausel erweiterten *Select*-Befehl kennengelernt. Damit ist es möglich, Abfragen aus Relationen in Variablen abzuspeichern. Der SQL-Standard erlaubt aber nur Abfragen, die höchstens ein Tupel einer Relation auswählen. Dies genügt in der Praxis jedoch bei weitem nicht. Denn häufig muss eine Relation Tupel für Tupel nach bestimmten Eigenschaften durchsucht werden. Diese Möglichkeit der zeilenweisen Suche bietet SQL mit den sogenannten SQL-Cursorn an. Zum Arbeiten mit SQL-Cursorn benötigen wir vier neue SQL-Befehle und die Variable *Sqlstate* (oder *Sqlcode*). Betrachten wir zunächst die neuen Befehle.

Tab. 48 Befehle zum Arbeiten mit SQL-Cursorn

SQL-Befehl	Beschreibung
DECLARE CURSOR	deklariert einen SQL-Cursor
OPEN	öffnet einen SQL-Cursor und setzt diesen vor die erste Zeile der Relation
FETCH	setzt den Cursor auf die nächste Zeile und gibt diesen Zeileninhalt aus
CLOSE	schließt einen geöffneten SQL-Cursor

In Tab. 48 finden wir zunächst den *Declare-Cursor*-Befehl. Er ähnelt sehr dem Erstellen von Sichten. Mit diesem Befehl wird quasi eine virtuelle Relation erzeugt, die wir dann mit den weiteren Befehlen Zeile für Zeile durchsuchen. Analog zum *Create-View*-Befehl wird der enthaltene *Select*-Befehl zum Aufbau dieser temporären Relation verwendet:

```
DECLARE Cursorname CURSOR FOR
   Select-Befehl
   [ FOR { READ ONLY | UPDATE [ OF Spalte [ , ... ] ] } ]
```

Mit dem Wort *Select-Befehl* ist hier die nicht eingebettete Version gemeint. Es darf demnach keine *Into*-Klausel verwendet werden. Weitere Einschränkungen existieren nicht, insbesondere ist auch das Sortieren mit der *Order-By*-Klausel erlaubt. Die *For*-Klausel der *Cursor*-Deklaration dient zum Kennzeichnen von Attributen, die beim Durchsuchen dieser Relation geändert werden dürfen. Standardmäßig ist das Ändern aller Attribute in der *Cursor*-Relation erlaubt. Voraussetzung ist allerdings, dass sich jedes Tupel und jedes Attribut dieser *Cursor*-Relation immer eindeutig auf den Inhalt einer Basisrela-

tion zurückführen lässt. Dazu müssen die gleichen Bedingungen wie für nichtänderbare Sichten erfüllt sein (siehe Abschnitt 6.3). Zusätzlich dürfen die Tupel nicht geordnet sein (keine *Order-By*-Klausel). Ist eine dieser Bedingungen nicht erfüllt, so gilt für die *Cursor*-Relation automatisch *Read Only*. Ist die *Cursor*-Relation änderbar, so können mittels der *For*-Klausel eine oder mehrere Attribute als änderbar gekennzeichnet werden. Auf alle anderen Attribute darf dann nur lesend zugegriffen werden. Mit dem Bezeichner *For Read Only* werden Änderungsversuche grundsätzlich abgewiesen.

Die anderen drei Cursor-Befehle besitzen eine einfache Syntax:

```
OPEN  Cursorname
CLOSE  Cursorname
FETCH  [ FROM ]  Cursorname  INTO  Variablenliste
```

Alle drei Befehle beziehen sich mit dem Cursornamen auf die *Declare-Cursor*-Definition, die in einer Datei vor den obigen drei Befehlen stehen muss. Es ist zu beachten, dass ein *Declare-Cursor*-Befehl einen Cursor lediglich deklariert. Erst mit dem *Open*-Befehl wird die dazugehörige temporäre Relation erzeugt und der Cursor vor das erste Tupel dieser Relation gesetzt. Nach dem *Open*-Befehl können wir jetzt mittels des *Fetch*-Befehls den Cursor Tupel für Tupel weiterschalten und den Inhalt der Tupel in die Variablenliste des *Fetch*-Befehls ausgeben. Es versteht sich von selbst, dass die Anzahl der Variablen und deren Datentypen im *Fetch*-Befehl mit der in der Attributsliste der *Cursor*-Deklaration übereinstimmen muss.

Beim Durchsuchen der Relation wird der Cursor früher oder später zum letzten Tupel der Relation gelangen. Ein weiterer *Fetch*-Befehl wird somit keine weiteren Daten mehr lesen können und daher mit einem Fehler enden. In den Variablen *Sqlstate* und *Sqlcode* wird dann der Wert '02000' bzw. 100 zurückgeliefert. Das Durchsuchen einer Relation mittels SQL-Cursor besitzt deshalb in der Praxis meist folgendes Aussehen:

```
EXEC SQL DECLARE Curs CURSOR FOR ... ;
            ...
EXEC SQL OPEN Curs ;
EXEC SQL FETCH FROM Curs INTO ... ;
while ( strcmp (SQLSTATE, "02000") != 0 )        // oder: SQLCODE != 100
{
    ...
    EXEC SQL FETCH FROM Curs INTO ... ;
}
EXEC SQL CLOSE Curs ;
```

Zu beachten ist dringend die Anordnung der *Fetch*-Befehle und der *While*-Schleife. Der erste *Fetch*-Befehl steht vor der *While*-Schleife, um schon beim ersten Zugriff den *Sqlstate*-Wert korrekt abzufragen. Schließlich könnte die Anzahl der durch die *Declare-Cursor*-Deklaration ausgewählten Tupel leer sein. Ein weiterer *Fetch*-Befehle steht am Ende der Schleife, so dass im Schleifenkopf wieder der Status eines *Fetch*-Befehls abgefragt wird (und nicht der Status eines anderen SQL-Befehls!).

Häufig sind neben dem Abfragen von Tupeln auch Änderungen erforderlich, und zwar jeweils in dem Tupel, auf das der Cursor gerade zeigt und dessen Inhalt soeben in die Variablen mittels eines *Fetch*-Befehls eingelesen wurde. Oder dieses Tupel soll ganz aus der Relation entfernt werden. Beides ist möglich, vorausgesetzt Änderungen wurden in der *Cursor*-Deklaration erlaubt. Zum Löschen oder Ändern von Tupeln, auf die der Cursor aktuell zeigt, stehen Varianten des *Delete*- und *Update*-Befehls zur Verfügung. Sie lauten:

```
DELETE FROM { Tabellenname | Sichtname }
 WHERE CURRENT OF Cursorname

UPDATE { Tabellenname | Sichtname }
 SET { Spalte = Spaltenausdruck } [ , ... ]
 WHERE CURRENT OF Cursorname
```

In den beiden bereits bekannten Befehlen wurde die *Where*-Klausel durch die *Where-Current*-Klausel ersetzt. Weiter beziehen sich beide Befehle auf die Basisrelation oder änderbare Sicht, die in der *Cursor*-Deklaration angegeben wurde. Eine entsprechende eindeutige Beziehung muss vorliegen, da beide Änderungsbefehle eine änderbare *Cursor*-Deklaration voraussetzen. SQL wird in diesem Fall die geforderte Änderung vornehmen. Andernfalls wird SQL die Änderung mit einer entsprechenden Fehlermeldung zurückweisen.

Auf ein kleines, aber wichtiges Problem sei noch hingewiesen: Es ist ein Fehler, in *Fetch*-Variablen *Null*-Werte einzulesen. In diesem Fall wird in der Variable *Sqlstate* der Fehlercode '22002' hinterlegt und der Wert der Variablen, in die dieser *Null*-Wert geschrieben werden soll, ist undefiniert. Um dieses Problem zu lösen gibt es zwei Möglichkeiten:

- Der zugrundeliegende Cursor wird mit einer entsprechenden *Where*-Klausel so definiert, dass keine *Null*-Werte auftreten können.
- Es werden *Indikator*-Variablen eingesetzt.

Eine *Indikator*-Variable zeigt eingelesene *Null*-Wert an. Die erweiterte Syntax der Variablenliste im *Fetch*-Befehl, aber auch in der *Into*-Klausel des *Select*-Befehls lautet:

```
{ Variable [ [ INDICATOR ] Variable ] } [ , ... ]
```

Wir schreiben also zu einer Variable noch eine weitere, getrennt durch das Wort *Indicator*. In diese weitere Variable wird der Wert 0 abgelegt, falls kein *Null*-Wert auftritt. Ansonsten wird ein Wert ungleich 0 gespeichert, und auch die Variable *Sqlstate* liefert wegen dieses *Null*-Wertes keinen Fehlercode mehr zurück. Durch Abfragen der Indikator-Variable kann daher immer überprüft werden, ob ein gültiger Wert ungleich *Null* aus der Datenbank gelesen wurde. Als Besonderheit ist in Oracle zusammen mit C++ zu beachten, dass die Indikator-Variable als *Short Int* zu definieren ist; in MS-Access sind *Indikator*-Variablen nicht definiert.

Betrachten wir ein einfaches Beispiel: Ist ein Cursor *Curs* definiert, der die Personalnummer, den Namen und die Personalnummer des Vorgesetzten anzeigt, so liest folgender Programmausschnitt das nächste Tupel ein und gibt aus, wenn ein Mitarbeiter keinem Vorgesetzten zugeordnet ist:

```
FETCH FROM Curs INTO :nr, :name, :vorgesetztnr INDICATOR :ind;

cout << "Persnr: " << nr << ", Name: " << name ;
if  ( ind == 0 )
    cout << ", Vorgesetzter: " << vorgesetztnr << endl;
else   //  kein Vorgesetzter:
    cout << ", kein Vorgesetzter" << endl;
```

In diesem Beispiel seien die Variablen *nr* und *vorgesetztnr* geeignete *Int*-Variablen, *ind* sei eine *Short*-Variable (in Oracle) und *name* eine hinreichend große Zeichenkette.

Wir schließen diesen Abschnitt mit einem Verweis auf ein umfangreiches Beispiel zu SQL-Cursorn in Anhang A (ab Seite 314). In diesem Beispiel wird ein bestehender Auftrag in der *Radl*-Datenbank storniert. Dies beinhaltet zunächst die Überprüfung, ob der Auftrag überhaupt existiert. Wenn ja, so wird ein Cursor deklariert. Dieser hängt von der Variablen *Auftragsnummer* ab. Jedes Tupel dieses Auftrags wird dann mittels einer *While*-Schleife ausgegeben. Nach Rückfrage werden diese Tupel aus den Relationen *Auftrag*, *Auftragsposten*, *Teilereservierung* und *AVO_Belegung* gelöscht. Da noch Abhän-

gigkeiten zur Relation *Lager* existieren, es wurden eventuell Teile für den Auftrag reserviert, muss mittels eines weiteren Cursors die Relation *Teilereservierung* durchsucht werden. Im Fehlerfall wird die gesamte Transaktion zurückgesetzt Es wird dringend empfohlen, dieses Beispielprogramm genau nachzuvollziehen.

9.5 Besonderheiten in MS-Access

Die in den bisherigen Abschnitten behandelte Einbettung von SQL in die Sprache C++ ist Bestandteil der meisten SQL-Datenbankhersteller, so auch in Oracle zusammen mit *Pro*C*. Viele Datenbanken bieten aber auch eine eigene Programmiersprachenumgebung an, Oracle etwa *PL/SQL*. Auch in MS-Access wird SQL in die Sprache *Visual Basic* eingebettet. Betrachten wir diese Einbettung stellvertretend für andere.

MS-Access ist eine Windows-Anwendung und unterstützt das Arbeiten mit Fenstern hervorragend. Diese grafische Oberfläche lässt auch das direkte Eingeben von SQL-Befehlen zu, ebenso das Programmieren mit Modulen in der Programmiersprache Visual Basic. Um SQL-Befehle direkt einzugeben, muss bei der im Internet erhältlichen Software (siehe Anhang D) nur der Schalter „SQL-Abfrage“ ausgewählt werden. In MS-Access selbst ist eine Abfrage zu starten (mittels des Menüs Einfügen/Abfrage), die Entwurfsansicht und dann das Menü Abfrage/SQL-spezifisch/Datendefinition zu wählen. Es kann dann immer genau ein SQL-Befehl eingegeben werden. Bei einer Abfrage wird das Ergebnis sofort angezeigt.

Sollen umfangreichere Aktionen durchgeführt werden, so können wir uns manchmal mit Makros behelfen. Makros sind eine Aneinanderreihung von mehreren Aktionen, etwa mehreren Abfragen. Bei komplexeren Aktionen müssen aber umfangreiche Prozeduren in der Sprache *Visual Basic* geschrieben werden. Diese Prozeduren verwaltet MS-Access als sogenannte Module. Die Schnittstelle zwischen der Sprache Visual Basic und SQL ist etwas gewöhnungsbedürftig, insbesondere ist kein vorangestelltes *EXEC SQL* notwendig. Auch die Transaktionsverarbeitung unterscheidet sich vom SQL-Standard. Jede Transaktion muss explizit gestartet werden. Es gibt in MS-Access folgende Transaktionsbefehle:

```
BeginTrans
CommitTrans
Rollback
```

BeginTrans sorgt innerhalb eines Arbeitsbereichs (Workspace) dafür, dass die Transaktionsmechanismen greifen. *CommitTrans* beendet eine Transaktion erfolgreich, *Rollback* setzt eine Transaktion zurück.

Dieser Abschnitt ist für den Benutzer mit Grundkenntnissen in MS-Access gedacht. Wir wollen hier vor allem den Zusammenhang zwischen SQL und Visual Basic aufzeigen. Dazu wiederholen wir das Beispiel zum Ändern des Gehalts um 2 Prozentpunkte aus dem Abschnitt 9.2:

```
Option Compare Database          ' Zeichenketten normal vergleichen
Option Explicit                  ' Variablen muessen definiert werden

Function GehaltErhoehen()
' Variablendeklaration:
Dim db As DATABASE               ' Datenbankvariable
Dim Transakt As Workspace        ' Arbeitsbereich fuer Transaktion
Dim SelectBefehl As Recordset    ' SelectBefehls-Bereich definieren
Dim UpdateBefehl As String       ' Nimmt den Update Befehl auf

On Error GoTo FehlerBehandlg     ' bei Fehlern verzweigen

Set db = CurrentDb               ' Datenbank oeffnen
Set Transakt = DBEngine.Workspaces(0)    ' Transaktionsbereich setzen

Transakt.BeginTrans              ' Transaktionsbeginn

'Belastung durch Gehaltserhoehung ermitteln:
Set SelectBefehl = db.OpenRecordset
                        ("Select Sum (0.02 * Gehalt) As Mehr From Personal")

' Ausgeben des Mehrbetrags. Einlesen, ob Erhoehung erfolgen soll:
If msgBox("Die Erhoehung kostet " & SelectBefehl!Mehr & " DM je Monat." & _
            Chr(13) & "Soll die Gehaltserhoehung durchgefuehrt werden", _
            vbYesNo, "Info zur Gehaltserhoehung") = vbNo Then
   msgBox "Keine Gehaltserhoehung.", vbInformation
Else
   ' Update wird durchgefuehrt:
   UpdateBefehl =    "Update Personal " & _
                     "Set Gehalt = 1.02 * Gehalt"      ' Erstellen des Befehls
   db.Execute UpdateBefehl                    ' Ausfuehren des Befehls
   msgBox "Gehaltserhoehung wurde durchgefuehrt.", vbInformation
End If

Transakt.CommitTrans
msgBox "Transaktion ist beendet.", vbInformation

Exit Function            'Beendet die Funktion

FehlerBehandlg:
   Transakt.Rollback
   msgBox "Ein Fehler ist aufgetreten", vbInformation

End Function
```

Aus diesem Modul erkennen wir, dass in MS-Access keine *Declare Section* existiert, dass es eine Art *Whenever*-Befehl gibt (*On Error GoTo*) und dass keine *Into*-Klausel in SQL-Befehlen unterstützt wird. Kommentare beginnen in Visual Basis immer mit einem Hochkomma-Zeichen und enden automatisch am Ende der Zeile. Betrachten wir das Modul im Einzelnen:

Die ersten beiden Zeilen sind spezielle Optionen, die von MS-Access standardmäßig gesetzt werden. Die erste Option gibt an, wie Zeichenkettenvergleiche durchgeführt werden, die zweite zwingt den Anwender, Variablen vor ihrer Verwendung zunächst zu definieren. Die weiteren Zeilen bestehen aus einer Funktion *GehaltErhoehen*. In dieser Funktion werden zunächst die benötigten Variablen deklariert, insbesondere eine Variable für die Datenbank, eine für den Arbeitsbereich und eine für den *Select*-Befehl. Anschließend erfolgt das Öffnen der aktuellen Datenbank, das Einrichten eines Arbeitsbereichs und das Starten einer Transaktion in diesem Bereich.

Jetzt wird mit der Funktion *OpenRecordset* ein *Select*-Befehl ausgeführt und die erste (und hier auch einzige) Zeile in die Variable *SelectBefehl!Mehr* eingelesen. Dieser Inhalt wird in einer Message-Box angezeigt, und im Falle einer positiven Antwort wird der *Update*-Befehl ausgeführt. Dazu wird der Befehl in eine Zeichenkette geschrieben und dann mit dem *Execute*-Befehl gestartet. Zuletzt wird die Transaktion mit dem Befehl *CommitTrans* beendet.

Einige Befehle im obigen Beispiel sind nicht leicht lesbar. Dies liegt auch an der Eigenheit von MS-Access, dass ein Befehl immer in einer Zeile stehen muss. Falls nicht, so ist als Fortsetzungszeichen das Unterstrichzeichen (‚_‘) zu verwenden. Natürlich ist vorher eine lange Zeichenkette zu beenden und in der nächsten Zeile fortzusetzen. Zeichenketten lassen sich dabei mit dem Zeichen ‚&‘ aneinanderfügen.

Auf das umfangreiche Beispiel zu SQL-Cursorn in Anhang A, auf das im letzten Abschnitt hingewiesen wurde, wollen wir auch in MS-Access nicht verzichten. Eine gleichwertige MS-Access-Version finden wir ebenfalls in Anhang A ab Seite 317.

9.6 Zusammenfassung

Eingebettetes SQL wird durch vier neue Begriffe geprägt: *Declare Section*, Fehlerkontrolle mittels *Sqlstate*/*Sqlcode*, Einzeilen-*Select* und *Cursor*.

Wir haben gesehen, dass SQL eine reine Datenbanksprache ist. Zum Programmieren müssen wir alle SQL-Befehle in eine Programmiersprache ein-

betten. Wir haben dies am Beispiel der Sprache C++ demonstriert. Um Variablen sowohl in der Programmiersprache als auch in SQL verwenden zu können, benötigen wir einen gemeinsamen Definitionsbereich, die *Declare Section*.

Um Ergebnisse aus Datenbankabfragen weiterzuverarbeiten, müssen diese in Variablen gespeichert werden. Eine Möglichkeit dazu bietet der Einzeilen-*Select*-Befehl, ein um eine *Into*-Klausel erweiterter *Select*-Befehl. Die Abfrage mittels dieses Befehls muss sich aber auf maximal ein Tupel beziehen.

Ist dies nicht sichergestellt, sind *Cursor* zu verwenden. Dieses mächtige Werkzeug im eingebetteten SQL erlaubt es, beliebige Abfragerelationen Tupel für Tupel zu durchsuchen. Auch ein interaktives Ändern des gerade betrachteten Tupels wird unterstützt.

Zur Kontrolle aller abgearbeiteten SQL-Befehle dienen schließlich die Variablen *Sqlcode* und *Sqlstate*. Nur mit diesen Variablen ist es möglich zu erkennen, ob die SQL-Befehle auch erfolgreich umgesetzt oder gewünschte Aktionen wegen eingetretener Fehler abgebrochen wurden.

9.7 Übungsaufgaben

Alle hier angesprochenen Relationen beziehen sich auf Relationen der Beispieldatenbank *Radl* aus Anhang A. Verwenden Sie zur Einbettung eine beliebige Programmiersprache, die vorgestellten Lösungen benutzen C++ als Wirtssprache.

1) Schreiben Sie mit eingebettetem SQL ein Programm, das alle Daten des Lieferanten mit der Nummer 1 auf Bildschirm ausgibt.
2) Schreiben Sie mit eingebettetem SQL ein Programm, das einen Familiennamen einliest, und dann ausgibt, ob dieser Name in der Personaltabelle existiert. Wenn ja, so wird auch ausgegeben, wie oft dieser Name vorkommt.
3) Schreiben Sie mit eingebettetem SQL ein Programm, das mitteilt, ob die Personalnummer 10 vergeben ist, und wenn ja, an welche Person.
4) Auf Seite 234 wird das prinzipielle Vorgehen beim Suchen mittels Cursor beschrieben. Wie sieht der Programmcode aus, wenn in C++ statt der *While*-Schleife eine *Do-While*-Schleife verwendet wird?
5) Eine neue Lieferung eines Lieferanten ist eingetroffen. Zwecks Inventarisierung ist ein Programm zu schreiben, das alle Teile (*Teilnr*, *Bezeichnung*, *Mass*, *Einheit*), die dieser Lieferant liefert, nacheinander auflistet. Es ist zu jedem dieser Teile einzugeben, wieviel geliefert wurde. Das Programm erhöht entsprechend den Bestand und reduziert gegebenenfalls das Attribut *Bestellt*.

6) Ein Auftrag zieht meist Arbeiten nach sich: Es sind Arbeitsvorgänge notwendig, wobei aus reservierten Teilen fertige Produkte hergestellt werden. Nach Beendigung dieser Arbeiten muss die Datenbank entsprechend aktualisiert werden. Schreiben Sie dazu ein Programm, das die Auftragsnummer einliest und die Auftragsdaten zur Kontrolle ausgibt. Wird bestätigt, dass die Auftragsarbeiten bereits ausgeführt sind, so werden die entsprechenden Daten aktualisiert. Betroffen sind die Relationen *Teilereservierung*, *Lager* und *AVO_Belegung*. In der Relation *Lager* ist zu beachten, dass die reservierten Teile vom Ist-Bestand abzuziehen sind. Sollten der Ist-Bestand dadurch negativ werden, so scheint ein Buchungsfehler vorzuliegen. In diesem Fall ist das Programm mit einer aussagekräftigen Fehlermeldung abzubrechen, alle schon vorgenommenen Änderungen sind zurückzunehmen.

10 Nicht-Relationale Datenbanken

Relationale Datenbanken gibt es erst seit etwa 1980. Zu dieser Zeit existierte bereits eine große Anzahl von Datenbanken. Der Umstieg von diesen meist netzwerkartigen oder hierarchischen Datenbanken auf relationale ist in der Regel kostspielig und mit Laufzeitnachteilen und höheren Ein- und Ausgaberaten verbunden. So sind auch heute noch große Datenbestände in nichtrelationalen Datenbanken abgespeichert. Vorsichtige Schätzungen gehen davon aus, dass zur Zeit immer noch mindestens 50% des Datenbankbestandes nichtrelational verwaltet wird!

Alle nichtrelationalen Modelle der letzten Jahrzehnte haben gemeinsam, dass sie mehr oder weniger Abbilder der zugrundeliegenden physischen Datenstrukturen sind. Dies wird verständlich, wenn wir an die Rechnerleistung der 60er und 70er Jahre denken. Nur optimal an die Hardwarestrukturen angepasste Datenbestände garantierten hinreichend schnelle Zugriffe. Dieser Kompromiss führt aber dazu, dass bei den logischen Datenstrukturen die physischen Strukturen durchschimmern, ganz im Gegensatz zu den vorgestellten relationalen Modellen. Erst in neuester Zeit wird versucht, mittels zusätzlicher Schnittstellen (z.B. SQL) zumindest dem Anwender diese hardwarenahe Oberfläche zu verbergen.

Ein weiterer Unterschied zwischen relationalen und nichtrelationalen Modellen liegt in der Einbettung: Relationale Datenbanken werden meist mittels der Sprache C/C++ eingebettet, die älteren nichtrelationalen fast ausschließlich mittels COBOL, in geringem Maße mittels PL/1 und erst seit neuestem auch mit C. Einbettung heißt, dass die Datenbankzugriffe innerhalb einer Wirtssprache (C, COBOL) aufgerufen werden.

In diesem Kapitel werden die invertierten Listen und die hierarchischen und netzwerkartigen Datenbankmodelle vorgestellt. Alle drei Modelle werden nur soweit eingeführt, dass die Funktionsweise ersichtlich wird. Um darüberhinaus noch bessere Vorstellungen von der Arbeitsweise dieser Modelle zu bekommen, wird bei den hierarchischen Modellen am Beispiel von IMS und bei den netzwerkartigen am Beispiel von UDS die Datenbankprogrammierung kurz erläutert. Einzelheiten hierzu finden wir in [Gee77], [UDS83] und [UDS86].

10.1 Invertierte Listen

Invertierte Listen sind nichts anderes als eine Ansammlung von Dateien, die in ein Datenbankverwaltungssystem eingebettet sind. Hier handelt es sich meist um ISAM-Dateien, aber auch B-Bäume oder Hashing dienen als Zugriffsbeschleunigung. Den Namen *Invertierte Listen* erhielten diese Datenbanken aus der meist hohen Anzahl invertierter Dateien für die Sekundärschlüssel. Beispiele für Datenbanken, die als invertierte Listen aufgebaut sind, sind:

DATACOM/DB	von Computer Associates (für IBM-Rechnern),
R/2	von SAP Walldorf (überwiegend für Großrechner).

Invertierte Listen unterscheiden sich nur wenig von relationalen Datenbanken. Schließlich besitzen auch relationale Modelle eine Indexverwaltung, wobei die Indizes häufig als invertierte Funktionen bezeichnet werden. Wir können sogar so weit gehen, relationale Modelle als invertierte Listen anzusehen.

Umgekehrt ist es ebenfalls meist relativ leicht, invertierte Listen mit einer „relationalen Oberfläche“ wie SQL zu versehen. In diesem Fall sind dann invertierte Listen quasi relationale Datenbanken. Trotzdem bleiben deutlich erkennbare Unterschiede zwischen invertierten Listen und den relationalen Datenbanken:

- Alle invertierten Listen mit relationaler Oberfläche lassen sich auch unter Umgehung dieser Oberfläche zugreifen. Dies ist sowohl aus Kompatibilitätsgründen als auch teilweise aus Sachzwängen heraus erforderlich. Bestimmte interne Strukturen müssen hier explizit verwaltet werden. Der Vorteil ist, dass dadurch hardwarenah eingegriffen und optimiert werden kann, was sich meist positiv auf die Laufzeit und den Resourcenbedarf auswirkt.
- Invertierte Listen sind immer physisch nach dem Primärschlüssel sortiert, relationale Modelle nicht notwendigerweise. Dies engt bei invertierten Listen die Wahl der zugrundeliegenden physischen Datenstruktur ein. Auch sind daher Primärschlüssel aus zusammengesetzten Attributen schwerer realisierbar als im relationalen Modell.

Ansonsten herrschen die Gemeinsamkeiten vor. Wie allgemein bei Datenbanken können wir auch hier die Zugriffe in DML- und DDL-Befehle aufteilen. Typische DML-Befehle bei invertierten Listen sind beispielsweise:

- *Locate First*: Suche den ersten Eintrag einer Tabelle T und gib dessen Adresse A zurück.
- *Locate First With Search Key Equal*: Suche den ersten Eintrag mit dem angegebenen Schlüssel in T und gib dessen Adresse A zurück.
- *Locate Next*: Suche ab einer angegebenen Adresse A den nächsten Eintrag und gib die neue Adresse zurück.
- *Locate Next With Search Key Equal*: Suche ab einer angegebenen Adresse A den nächsten Eintrag mit dem angegebenen Suchschlüssel und gib die neue Adresse zurück.
- *Retrieve*: Gibt den Eintrag an der angegebenen Adresse A aus.
- *Update*: Ändere den Eintrag an der angegebenen Adresse A.
- *Delete*: Lösche den Eintrag an der angegebenen Adresse A.
- *Store*: Speichere einen neuen Eintrag und gib die Speicheradresse A zurück.

Dieser Auszug aus typischen DML-Befehlen bei invertierten Listen zeigt, dass sehr viel mit Adressen gearbeitet wird. Die ersten beiden Befehle werden etwa verwendet, wenn wir mit der Suche in einer bestimmten Tabelle beginnen. Diese Befehle liefern Adressen zurück, mit deren Hilfe dann mit den anderen *Locate*-Befehlen weitergesucht werden kann. Auch das Lesen der Einträge (*Retrieve*) oder der Update geschehen über die Adressen, die die *Locate*-Befehle zurückliefern. Dieses Vorgehen lässt meist schnelle Suchzugriffe zu, führt aber auch zu vielen unterschiedlichen DML-Befehlen.

Leider gibt es bei invertierten Listen keine allgemeinen Integritätsregeln. Ein mathematisch fundierter Aufbau wie in relationalen Datenbanken mittels Fremdschlüsseln existiert hier nicht. Häufig sind auch nichteindeutige Schlüssel erlaubt. Nur wenige Datenbanken bieten eine hinreichende Unterstützung zur Integrität standardmäßig an. Folglich muss meist der Benutzer, sprich der Anwendungsprogrammierer, für die erforderlichen Abfragen sorgen, insbesondere auch zur Sicherstellung der Referenz-Integritätsregel.

10.2 Hierarchische Datenbanken

Grob gesprochen ist eine hierarchische Datenbank eine geordnete Menge von Bäumen. Die folgende exaktere Definition dürfte zunächst Verwirrung stiften:

Definition (hierarchische Datenbank)

☞Eine hierarchische Datenbank ist eine geordnete Menge, die aus einer Vielzahl von Inhalten eines gegebenen Baumtyps besteht.

Diese Definition klingt nicht einsichtig, insbesondere ist zunächst außer der Hierarchie keine Ordnung erkennbar. Wir werden im weiteren Verlauf dieses Abschnitts für Klärung sorgen. Dazu betrachten wir zunächst das Aussehen einer hierarchischen Datenbank. Wie bei anderen Datenbanken auch, müssen wir zwischen dem strukturellen Aufbau einer Datenbank und den Dateninhalten unterscheiden. Dies war bei relationalen Datenbanken nicht sonderlich schwer. Ein Entity-Relationship-Diagramm zeigt mit den Beziehungen zwischen den einzelnen Relationen den strukturellen Aufbau. Jede dieser Relationen enthält dann Tupel mit gespeicherten Daten.

Beginnen wir mit dem strukturellen Aufbau von hierarchischen Datenbanken. Wir definieren zunächst die Baumtypen, und erst dann werden wir uns ansehen, wie die Daten in diesen Strukturen abgespeichert werden. Ein Baumtyp besteht aus einem einzigen Satztyp, der Wurzel (im englischen: record type), zusammen mit einer geordneten Menge von keinem oder mehreren Unterbaumtypen. Ein solcher Unterbaumtyp ist wieder ein Baumtyp, insbesondere besteht er ebenfalls aus einem Satztyp und eventuell mehreren Unterbaumtypen.

Sehen wir uns den Aufbau an einem Beispiel an, der Weiterbildungsdatenbank aus Abb. 38. Diese Datenbank enthält alle Informationen zu den einzelnen angebotenen Kursen, dem Ort, dem Datum, dem Dozenten und den teilnehmenden Studenten. Darüberhinaus sind die erforderlichen Voraussetzungen an Vorkursen angegeben.

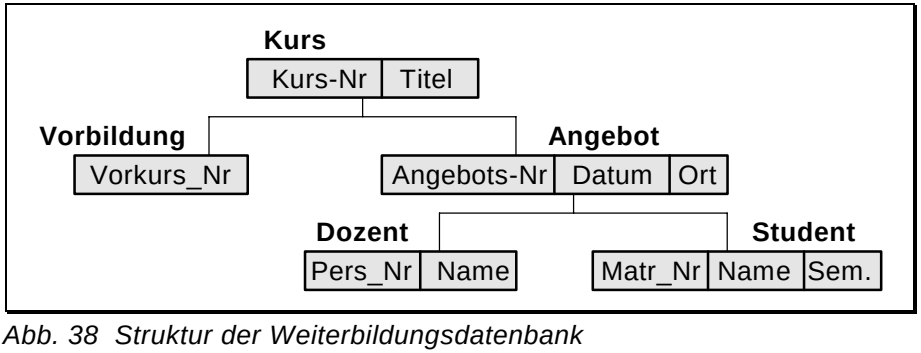


Abb. 38 Struktur der Weiterbildungsdatenbank

Diese Weiterbildungsdatenbank enthält fünf Satztypen. Diese sind hierarchisch aufgebaut. Jeder der schraffierten Bereiche (*Kurs*, *Vorbildung*, *Angebot*, *Dozent*, *Student*) ist ein Baumtyp. So besteht etwa *Kurs* aus einem Satztyp (*Kurs-Nr*, *Titel*) und den Unterbaumtypen *Vorbildung* und *Angebot*. Zum besseren Verständnis führen wir noch einige in hierarchischen Datenbanken gebräuchliche Begriffe ein:

Definition (Vater, Sohn, Bruder in hierarchischen Datenbanken)

☞ Der übergeordnete Typ eines Baumtyps heißt Vater, jeder untergeordnete Typ Sohn. Besitzen zwei Baumtypen den gleichen Vater, so heißen sie Brüder (engl.: twins).

Im obigen Beispiel ist etwa *Angebot* ein Sohn von *Kurs*, gleichzeitig Vater von *Student* und *Dozent* und letztlich Bruder zu *Vorbildung*. Wir sehen auch, dass es zwischen den einzelnen Baumtypen keine Fremdschlüssel gibt, so auch etwa keine Kursnummer im Baumtyp *Angebot*. Die Hierarchie ist hier das (einzige) verbindende Glied.

Wir haben damit schon einen der Hauptunterschiede zwischen relationalen und hierarchischen Datenbanken erwähnt: Während bei ersteren die Zusammenhänge zwischen den einzelnen Relationen mittels Fremdschlüssel hergestellt werden, sind die Zusammenhänge in hierarchischen Systemen ausschließlich mittels Vater-Sohn-Beziehungen gegeben.

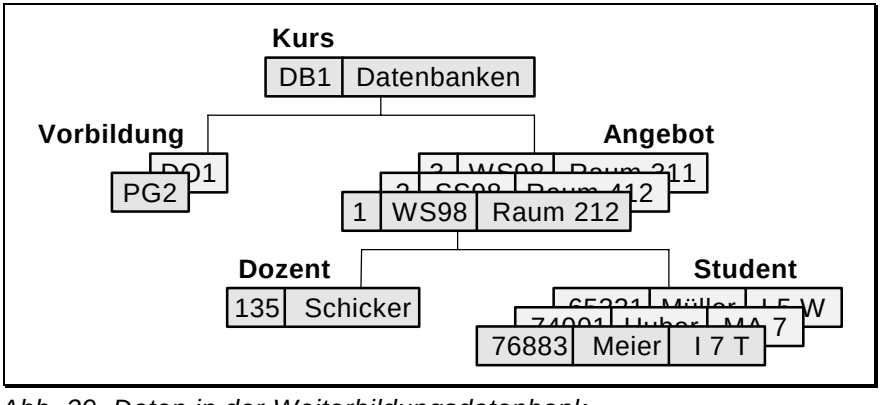


Abb. 39 Daten in der Weiterbildungsdatenbank

Nachdem wir die Baumtypen kennengelernt haben, wollen wir jetzt auch zeigen, wie die Nutzdaten in dieser Struktur abgespeichert werden. Betrachten wir dazu unsere Weiterbildungsdatenbank in Abb. 39, die mit einigen Beispieldaten gefüllt ist. In dieser Datenbank sind zu jedem Wurzelinhalt (zu je-

dem Kurs) die Kursangebote, die erforderlichen Vorkurse, und zu jedem Angebot die teilnehmenden Studenten und der Dozent angegeben. In diesem Beispiel sind aus Übersichtlichkeitsgründen viele Daten nicht mit aufgenommen worden. Die Abbildung enthält nur einen einzigen Kurs (*Datenbanken*) und auch hier nur die Teilnehmer und den Dozenten des Kursangebots mit der Nummer 1. Natürlich kann es weitere Kurse geben, und auch die anderen Angebote des Kurses *Datenbanken* besitzen ebenfalls Teilnehmer und Dozenten. Diese Art der Speicherung in hierarchischen Datenbanken führt demnach je nach Inhaltsmenge zu einer starken Auffächerung der Wurzel zu den Söhnen, Enkeln und Urenkeln.

Diese Weiterbildungsdatenbank zeigt deutlich auf, dass zwar die Typdefinition recht übersichtlich und anschaulich ist, dass aber das Abspeichern von Daten innerhalb dieser Baumstruktur zu recht komplexen Gebilden führt. Zur besseren Veranschaulichung können wir uns hier zusätzlich zur Hierarchie noch Lineare Listen zum Aufnehmen der Daten innerhalb jeder Hierarchiestufe vorstellen, wobei von jedem Listenelement aus Verzweigungen zu Söhnen existieren können.

Die schnelle Suche sowohl entlang der Hierarchie als auch innerhalb der Listen erzwingt eine Anordnung der einzelnen Elemente. In jeder hierarchischen Datenbank sind alle Dateneinträge geordnet! Wir können daher beispielsweise immer angeben, ob der Student *Huber* in dieser Ordnung vor dem vorausgesetzten Kurs *PG2* kommt oder nicht. Die Ordnung in einem hierarchischen Baumsystem lässt sich sehr anschaulich an unserem Kursbeispiel aufzeigen und kann in Abb. 40 nachvollzogen werden. In dieser Abbildung geben die Zahlen von 1 bis 10 die Reihenfolge der Anordnung an.

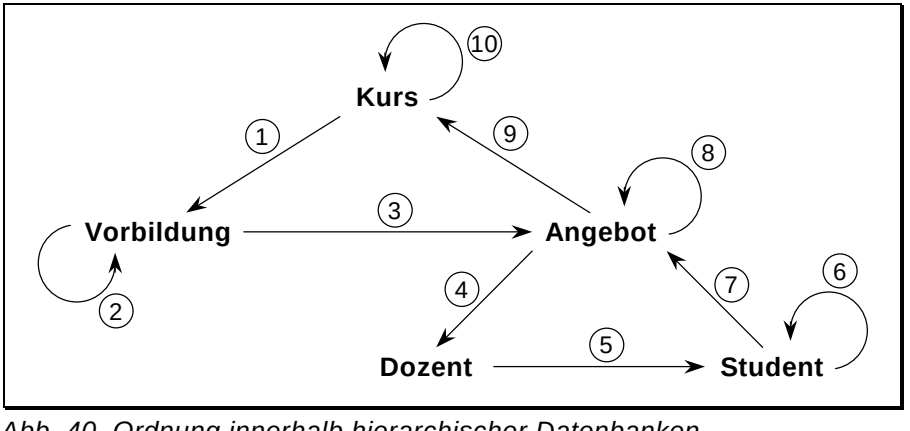


Abb. 40 Ordnung innerhalb hierarchischer Datenbanken

Die Ordnung ist einfach zu merken, wenn die folgenden Regeln beachtet werden: Auf einen Baumknoteneintrag folgt immer der linke Sohn, existiert kein Sohn, oder sind die Söhne bereits aufgelistet, so folgen die weiteren Einträge des gleichen Baumknotens. Existieren auch keine weiteren Einträge, so gehen wir schließlich zum rechts folgenden Bruder. Ist kein weiterer rechter Bruder vorhanden, so kehren wir zum Vater zurück.

Wenden wir diese Regeln bei jedem Baumknoteneintrag an, so impliziert dies, dass zunächst alle Söhne (und damit auch Enkel und eventuelle Urenkel) vollständig durchlaufen werden, bevor weitere Einträge des gleichen Typs und schließlich weitere Brüder angereiht werden. In unserem Beispiel wird also zunächst unser Kurs *Datenbanken* vollständig abgearbeitet, bevor der nächste Kurs an die Reihe kommt.

Versuchen wir die Ordnung mittels unseres Kursbeispiels nachzuvollziehen. Beginnen wir mit dem ersten Kursangebot. Gemäß der Ordnung folgen nacheinander alle Dozenten (④)und dann (über ⑤) alle Studenten (⑥) dieses Kursangebots. Erst dann kommt das nächste Angebot (⑧), gefolgt von der entsprechenden Dozenten- und Studentenliste dieses Angebots. Gibt es schließlich keine weiteren Angebote mehr, so wird zum Vater zurückgekehrt (⑨). Erst jetzt reiht sich der nächste Kurs mit all seinen Söhnen an (⑩).

Wegen dieser vorhandenen eindeutigen Anordnung aller Elemente einer hierarchischen Datenbank ist die Hierarchie genaugenommen flach. Jetzt werden wir auch die Definition der hierarchischen Datenbanken am Anfang dieses Abschnitts verstehen. Genaugenommen liegt nur eine geordnete Menge von Einträgen vor. Wie die Daten nun tatsächlich abgespeichert sind, hängt stark vom jeweiligen Hersteller der Datenbank ab. Wir werden einige Speicherungsmöglichkeiten am Beispiel von IMS im nächsten Abschnitt aufzeigen.

Trotzdem dürfen wir die Hierarchie nicht vergessen. Die DML-Befehlen sind nämlich sehr stark von der Ordnung und der Hierarchie beeinflusst. Hier spiegelt sich die große Leistungsfähigkeit dieser Datenbanken wieder. Es wird sowohl entlang der Ordnung als auch entlang der Hierarchie gesucht, und die Daten sind gerade für diese Suchvorgänge optimiert abgespeichert. In der Regel finden wir in diesen Datenbanken folgende DML-Befehle:

- Positionierung auf den gewünschten Baum (z.B. auf den Kurs *DB1*).
- Positionierung zum nächsten Baum (z.B. auf den Kurs, der auf *DB1* folgt).
- Positionierung zum nächsten Satz innerhalb des Baums entlang der Hierarchie (also vom Vater zu einem Sohn oder umgekehrt).

- Positionierung zum nächsten Satz innerhalb eines Baum entlang der Ordnung (z.B. vom Dozenten zum ersten Studenten).
- Einfügen von Sätzen (z.B. Einfügen eines weiteren Angebots).
- Löschen von Sätzen (z.B. Löschen eines Angebots oder eines Studenten).

Beim Löschen eines Satzes erkennen wir ein Problem: Wollen wir etwa ein Angebot löschen, so hängen der Dozent und die Studenten dieses Angebots quasi in der Luft. Sie besitzen ja keine Verbindung mehr. Aus Datenintegritätsgründen sind daher mit einem Vater auch alle seine Söhne zu löschen. Das Gleiche gilt sinngemäß auch für einen Update. In der Terminologie von Fremdschlüsseln bei relationalen Datenbanken sind demnach Söhne gegenüber ihren Vätern schwache Entitäten, und es gilt entsprechend:

```
NOT NULL
ON DELETE CASCADE
ON UPDATE CASCADE
```

Dies sollte als eine kleine allgemeine Einführung in die hierarchischen Datenbanken genügen. Im nächsten Abschnitt zeigen wir noch am Beispiel der Datenbank IMS, wie diese hierarchischen Systeme implementiert werden.

10.3 Hierarchisches System IMS

Wir haben im letzten Abschnitt bereits den prinzipiellen Aufbau hierarchischer Datenbanken kennengelernt. Es sind aber noch Fragen offen geblieben, insbesondere zur Technik des Speicherns dieser Systeme und zu typischen Zugriffs- und Datenbeschreibungsbefehlen. Diese Fragen wollen wir am Beispiel der weltweit verbreitetsten hierarchischen Datenbank, IMS (Information Management System), beantworten. Die Datenbank IMS existiert seit 1968 und wird von der Firma IBM angeboten. Sie ist die erste bedeutende Großdatenbank der Welt, inzwischen aber schon etwas in die Jahre gekommen.

Wir werden in diesem Abschnitt noch sehr genau sehen, dass die Baumstruktur hierarchischer Systeme zwar einfach und einprägsam, die Implementierung jedoch nur extrem komplex zu realisieren ist. Ein Verbergen der physischen Struktur ist unter keinen Umständen möglich. Weiter zeigt IMS deutlich, dass nur mit zusätzlichem hohen Aufwand die hierarchische Struktur ge-

ringfügig aufgebrochen werden kann. Beginnen wir aber zunächst mit dem logischen Aufbau dieses Systems.

Die logische Struktur der Datenbank ist in der Database Description (DBD) und in den Program Communication Blocks (PCB) definiert. Dabei entspricht ein DBD grob einer Tabelle und ein PCB einer Sicht in relationalen Datenbanken, genauer einem *Create-Table*- bzw. einem *Create-View*-Befehl.

Mit den DBDs wird demnach die Datenbank beschrieben. Leider ist dies nicht ganz einfach. Es gibt in IMS nämlich „physische“ DBDs und „logische“ DBDs. Die „physischen“ DBDs beschreiben die physische Struktur der Datenbank genau, einschließlich der wichtigsten Integritätsregeln (etwa: Jeder Sohn besitzt einen Vater). Die „logischen“ DBDs setzen auf diesen „physischen“ DBDs auf und beschreiben die Datenbank schon etwas mehr aus Sicht des Benutzers, insbesondere werden hier noch weitere Integritätsregeln aufgenommen.

Die PCBs wiederum setzen auf den „logischen“ DBDs auf. Sie sind die eigentliche Schnittstelle zum Benutzer. Die PCBs unterscheiden sich von den DBDs dadurch, dass hier Hierarchiedaten, die in einer bestimmten Anwendung nicht interessieren, verborgen werden können. Es können einzelne Einträge oder ganze Hierarchiezweige weggelassen werden. Der verbleibende Baum muss aber ein Teil des ursprünglichen Baums sein, der in den DBDs beschrieben wurde.

Diese Baumstruktur und die feste Anordnung der Daten innerhalb dieser Struktur erlauben nur eine geringe Flexibilität gegenüber Änderungswünschen in der Struktur. Alle neuen Daten müssen dieser Baumstruktur angepasst werden. Um diese Einengung zumindest etwas zu lockern, wurde in IMS ein Sekundärschlüssel eingeführt. Mittels dieses Sekundärschlüssels kann ebenfalls auf die Datenbank zugegriffen werden. Genaugenommen erhalten wir über diesen Schlüssel einen weiteren Zugriffspfad. Dieser Pfad ist auch hierarchisch geordnet, kann aber zur ursprünglichen Struktur völlig konträr sein. Zugriffe auf die Datenbank sind somit sowohl direkt als auch über den Sekundärschlüssel möglich. Die Beschreibung der Sekundärschlüssel erfolgt über eigene PCBs. Leider unterscheiden sich PCBs im Aufbau, je nachdem ob es sich um das Abbild eines „Original“-DBD oder eines Sekundärschlüssels handelt.

Wir wollen jetzt ein Beispiel zu einem Sekundärschlüssel angeben und betrachten dazu wieder unsere Weiterbildungsdatenbank. Diesmal sind wir daran interessiert, welcher Student welche Kurse bereits besuchte und welche er im Augenblick gebucht hat. Diese Daten lassen sich aus der ursprünglichen Da-

tenstruktur nur äußerst zeitaufwendig ermitteln (sequentielles Durchsuchen aller Kurse nach allen Studenten!). Eine mögliche Struktur des Sekundärschlüssels zwecks schneller Suche nach Studenten ist aus Abb. 41 abzulesen. Mit Hilfe dieser neuen Hierarchie können jetzt die Studentendaten sehr schnell aufgefunden werden.

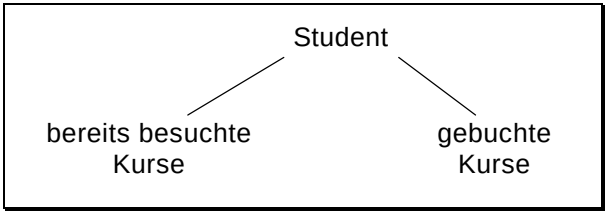


Abb. 41 Sekundärschlüssel auf Weiterbildungsdatenbank

Diese Sekundärschlüssel haben viel Ähnlichkeiten mit der Datenorganisation. Dort waren die Daten nach dem Primärschlüssel geordnet, während die Sekundärschlüssel über invertierte Dateien abgebildet werden mussten. Auch hier erfordert der Sekundärschlüssel eine eigene interne Verwaltung. Vermutlich wegen dieser Komplexität erlaubt IMS nur maximal einen solchen zusätzlichen Schlüssel. Der Vorteil ist der enorm schnelle Zugriff über zwei Strukturen, der Original- und der virtuellen Sekundärhierarchie.

Tab. 49 DBD der Weiterbildungsdatenbank

1	DBD	NAME=BILDGDBD
2	SEGM	NAME=KURS, BYTES=36
3	FIELD	NAME=(KURS-NR, SEQ), BYTES=3, START=1
4	FIELD	NAME=(TITEL, SEQ), BYTES=33, START=4
5	SEGM	NAME=VORBILD, PARENT=KURS, BYTES=3
6	FIELD	NAME=(VORKRS#, SEQ), BYTES=3, START=1
7	SEGM	NAME=ANGEBOT, PARENT=KURS, BYTES=21
8	FIELD	NAME=(ANGEBT#, SEQ), BYTES=3, START=1
9	FIELD	NAME=(DATUM, SEQ), BYTES=6, START=4
10	FIELD	NAME=(ORT, SEQ), BYTES=12, START=10
11	SEGM	NAME=DOZENT, PARENT=ANGEBOT, BYTES=24
12	FIELD	NAME=(PERS_NR, SEQ), BYTES=6, START=1
13	FIELD	NAME=(NAME, SEQ), BYTES=18, START=7
14	SEGM	NAME=STUDENT, PARENT=ANGEBOT, BYTES=27
15	FIELD	NAME=(MATR_NR, SEQ), BYTES=6, START=1
16	FIELD	NAME=(NAME, SEQ), BYTES=18, START=7
17	FIELD	NAME=(SEMESTER, SEQ), BYTES=3, START=25

Wir wollen uns jetzt endlich dem Aufbau der DBDs und PCBs widmen. Beginnen wir mit den DBDs. Der DBD unserer Weiterbildungsdatenbank ist in Tab. 49 angegeben. Wir erkennen, dass er aus einer Aufzählung von Segmenten besteht, wobei zu jedem Segment die einzelnen Felder des Segments aufgeführt werden. Jedes Segment ist ein Baumknoten, wobei der Vater des Segments mit anzugeben ist. Die Reihenfolge der einzelnen Segmente ist zwingend vorgeschrieben, sie geben nämlich die spätere Anordnungsreihenfolge wieder. Jedes Segment besteht aus Einzelinformationen, den Feldern. Zu jedem Feld ist die Größe des Feldes und die Startadresse innerhalb des Segments mit anzugeben. Die Summe der Feldgrößen eines Segments muss gleich der Größe des Segments sein. Das häufig vorkommende Wort *Seq* gibt die Anordnung an, hier also sequentiell.

Beim DBD in Tab. 49 handelt es sich nur um einen Auszug aus dem tatsächlichen DBD, viele Einzelheiten wurden weggelassen. Wir wollen hier die Details nicht weiter betrachten. Als einzige interessante Bemerkung sei nur erwähnt, dass IMS keine Datentypen kennt. Es werden hier nur Angaben über die Datengröße jedes Feldes eingetragen.

Kommen wir nun zu den PCBs. Bereits weiter oben wurde erwähnt, dass PCBs eine Art Sicht sind, die dem Benutzer angeboten werden. Betrachten wir dazu wieder unser Beispiel. Einige Anwender interessieren sich vielleicht je Kurs nur für die Angebote und die teilnehmenden Studenten. Unsere Sicht, sprich PCB, hätte dann im Vergleich zur ursprünglichen Hierarchie in Abb. 38 ein wesentlich einfacheres Aussehen. Diese Sicht ist in Abb. 42 dargestellt.

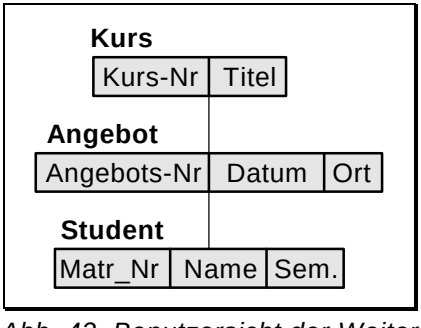


Abb. 42 Benutzersicht der Weiterbildungsdatenbank

Zu dieser Benutzersicht finden wir jetzt den dazugehörigen PCB in Tab. 50. Wieder enthält diese Tabelle nur die wichtigsten Einträge. In der ersten Zeile muss der Bezug zum DBD hergestellt werden. Es ist hierbei zu beachten, dass die Baumwurzel des PCBs mit der des DBD übereinstimmen muss. Der Name *SENSEG* bedeutet *Sensitive Segment* und *SENFLD* steht für *Sensitive Field*.

Die Option *PROCOPT* gibt die Zugriffsrechte an, wobei *G* für *Get* steht (*I=Insert, R=Replace, D=Delete*).

Tab. 50 PCB einer Sicht der Weiterbildungsdatenbank

1	PCB	DBDNAME=BILDGDBD
2	SENSEG	NAME=KURS, PROCOPT=G
3	SENSEG	NAME=ANGEBOT, PARENT=KURS, PROCOPT=G
4	SENFLD	NAME=ANGEBT#, START=1
5	SENFLD	NAME=DATUM, START=4
6	SENSEG	NAME=STUDENT, PARENT=KURS, PROCOPT=G

In unseren bisherigen Datenbeschreibungen sind weder der Sekundärschlüssel enthalten, noch irgendwelche Integritätsregeln. Um all diese Möglichkeiten wie Sekundärschlüssel, Integritätskonstrukte oder Transaktionen überhaupt gleichzeitig verwenden zu können ohne interne Widersprüche zu erhalten, ist eine Unzahl von Regeln zu beachten, die die Schnittstelle weiter komplizieren. Wir betonen, dass wir uns mit diesen Beispielen wirklich nur an der Oberfläche der insgesamt möglichen Eingaben bewegen.

Mit diesem nicht gerade ermutigenden Ausblick auf die Datenbeschreibung in IMS wenden wir uns jetzt der Zugriffssprache (DML) zu. Im Gegensatz zu den meisten relationalen Datenbanken bietet IMS keine eigene Oberfläche an. Vielmehr ist IMS ausschließlich über die Programmiersprachen COBOL, PL/1 und Assembler zugreifbar. Hierfür bietet IMS entsprechende Unterprogrammschnittstellen an. Ein Beispiel für einen solchen Zugriff aus PL/1 heraus ist im Folgenden angegeben:

```
CALL PLITDLI (SIX, GU, BLDGPCP, STUDENT_FELD, KSSA, ASSA, SSSA) ;
```

In PL/1 wird mit dem Bezeichner *Call* ein Unterprogrammaufruf eingeleitet. Die Unterprogrammschnittstelle von PL/1 zu IMS lautet *Plitdli*. Diese Prozedur besitzt in unserem Falle sieben Parameter, wobei gleich der erste die Zahl der folgenden Parameter angibt. Diese beinhalten nacheinander die Zugriffsmethode (*GU* = Get Unique), einen Zeiger (*Bldgpcp*) auf die aktuelle Position, ein Rückgabefeld für das Ergebnis und sogenannte SSA-Felder. SSA steht für *Segment Search Arguments* und gibt die Zugriffshierarchie aus (K*urs*SSA, A*ngebot*SSA, S*tudent*SSA). Mit diesem Aufruf wird ein bestimmter Student gesucht, und die Daten werden im Feld *Student_Feld* zurückgegeben.

Die Zugriffsmethoden selbst sind natürlich auf die Baumstruktur zugeschnitten. Einen Ausschnitt aus den Möglichkeiten liefert Tab. 51. Alle diese

Befehle lassen die Baumstruktur erkennen. Soll etwa mit ISRT ein neuer Student eingegeben werden, so ist anzugeben, zu welchem Kurs und welchem Angebot dieser Student gehören soll.

Tab. 51 Zugriffsmethoden in IMS

GU	Get Unique (Direktzugriff)
GN	Get Next (sequentielle Suche entsprechend der Ordnung)
GNP	Get Next within Parent (sequentielle Suche nur unter dem aktuellen Vater)
GHU, GHN, GHNP	wie GU, GN, GNP, wobei zusätzlich nachfolgende Lösch- und Update-Funktionen möglich sind
ISRT	Neues Segment (Knoten) einfügen
DLET	Segment entfernen
REPL	Update eines Segments

Zuletzt gehen wir noch kurz auf die physische Speicherung der Bäume in IMS ein. Hier erlaubt IMS fast alle nur denkbaren Wünsche, von einer einfachen sequentiellen Speicherung auf Band bis zu komplexen ISAM- oder Hash-Strukturen. Der Datenbankadministrator wird beim Erzeugen der Datenbank diese physischen Strukturen endgültig festlegen. Diese Strukturen wirken sich enorm auf die Laufzeiten beim Zugriff aus, Änderungen sind nur über komplette, und damit enorm aufwendige Reorganisationen möglich. Wir wollen nur einige Möglichkeiten der physischen Speicherungstechniken in IMS ansprechen.

① Sequentielle Speicherung gemäß der Ordnung: Diese einfachste Speicherung wird vorzugsweise für Bänder verwendet. Sie erlaubt kaum Änderungen der Daten und erfordert immer eine sequentielle Suche.

② Jedes Wurzel-Element besitzt einen Index, ansonsten sequentielle Speicherung gemäß der Ordnung: Diese teils adressierte, teils sequentielle Speicherung bietet vor allem bei nicht allzu tief gefächerten Bäumen schon deutlich schnellere Zugriffe als ①.

③ Verkettung gemäß der Ordnung: Diese geordnete verkettete Liste besitzt genau die Vor- und Nachteile, die wir bereits in Kapitel 1 zu Verkettungen erwähnten: aufwendige sequentielle Suche, aber einfaches Ein- und Ausfügen von Segmenten.

④ Verkettung entlang der Baumstruktur, zusätzlich Verkettung aller Knotenelemente: Die Verkettung des Baumes garantiert kurze Such-

zeiten entlang des Baumpfades. Sind allerdings viele Knotenelemente (etwa viele Angebote) vorhanden, so muss dort wieder sequentiell gesucht werden. Trotzdem ist diese Lösung schon deutlich laufzeitfreundlicher als alle bisherigen.

⑤ Verkettung entlang der Baumstruktur, zusätzlich Hashverfahren oder Indizes zur Suche der Knotenelemente: Gegenüber ④ wurde auch die Suche einzelner Knotenelemente optimiert. Dies ist ein aufwendiges, von der Laufzeit her aber fast unschlagbares Verfahren.

Natürlich können wir jetzt die gleichen (oder andere) Verkettungen auch für den Sekundärschlüssel verwenden. Die Gesamtstruktur ähnelt damit schon mehr einem Netz als einem Baum (ein Sohn kann zwei Väter besitzen, einen über den Originalbaum, einen über den Sekundärbaum). Wir wollen es dabei bewenden lassen und ziehen ein kurzes Fazit:

- Hierarchische Datenbanken sind von der Zugriffszeit her immer dann extrem schnell, wenn sich die reale Welt als Hierarchie anordnen lässt. Nur dann sind optimale Baumstrukturen erstellbar, denn die Zugriffe orientieren sich immer nach dieser Struktur.
- Der Anwender muss die Zugriffswege angeben, also genauestens kennen. Die physische Struktur bemerkt der Endbenutzer an Hand der Laufzeiten bestimmter Abfragen.
- IMS ist sehr komplex, es erfüllt als Großdatenbank jedoch alle Anforderungen an Integrität, Sicherheit, Recovery und Concurrency. Dies erfordert intern viele Einschränkungen, was zu einer ausufernden Anzahl von Regeln führt, sowohl bei der Datenbeschreibung als auch bei den Zugriffen. Es existiert auch keine mathematisch fundierte Theorie für diese Strukturen wie bei den Relationen. Die Folge ist, dass ein Arbeiten mit IMS eine lange Einarbeitungszeit und viel Erfahrung erfordert, sei es als Programmierer oder insbesondere als Datenbankadministrator.

Die knappe Schlussfolgerung ist, dass bei hierarchischen Datenbanken die reale Welt an die Datenbankstruktur angepasst wird, wofür wir eine hohe Leistung erhalten, allerdings zum Preis einer enorm inflexiblen und komplexen Datenbank. Dass sich IBM der Problematik dieses Datenbanktyps wohlbewusst ist, beweist vielleicht folgendes: Ende der 60er Jahre erhielt ein gewisser E. F. Codd von IBM den Auftrag, sich Gedanken über eine neue Datenbankgeneration zu machen. Er erfand die relationalen Datenbanken!

10.4 Netzwerkartige Systeme

Schon bei der Entwicklung der hierarchischen Systeme in den 60er Jahren wurden die Schwierigkeiten erkannt, die sich bei der Umsetzung der realen Welt auf Baumstrukturen ergeben. Aus dieser Zeit stammt auch die Erweiterung dieser Hierarchien zu Netzstrukturen. Die starre Vater-Sohn-Hierarchie wurde aufgebrochen. Während es in Hierarchien zu jedem Sohn immer genau einen Vater gibt, werden jetzt zusätzlich beliebig viele Väter zugelassen. Wir können dadurch quasi beliebige Vernetzungen zwischen den einzelnen Tabellen herstellen, wir sprechen deshalb von netzwerkartigen Datenbanken. Die Söhne heißen jetzt Mitglieder (engl.: *member*) und die Väter Eigentümer (engl.: *owner*). Die Verbindungen werden als Set bezeichnet. Ein einfaches Netzwerk mit Eigentümern, Mitgliedern und Sets ist in Abb. 43 angegeben.

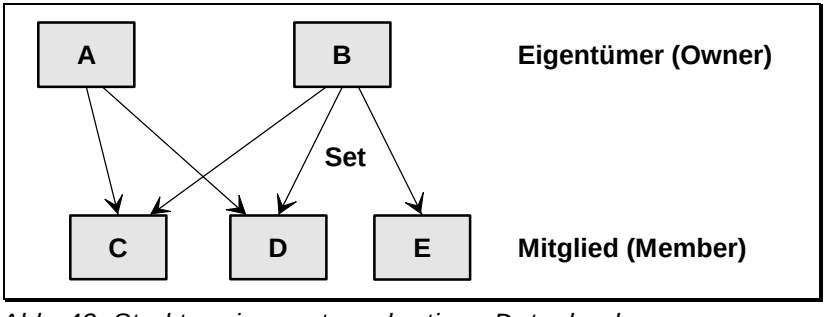


Abb. 43 Struktur einer netzwerkartigen Datenbank

Aus der Abbildung können wir erkennen, dass jetzt jedes Mitglied mehreren Eigentümern gehören kann. Umgekehrt kann jeder Eigentümer mehrere Mitglieder besitzen. Damit sind die Möglichkeiten, eine Datenbank zu konstruieren, wesentlich vielfältiger als in Hierarchien. Im Prinzip entsprechen die Sets den aus den relationalen Datenbanken bekannten Fremdschlüsseln.

Dieser Aufbau netzwerkartiger Strukturen wurde in allen Einzelheiten in einer Norm festgelegt. Ein internationales Programmiersprachen-Komitee erarbeitete die sogenannte CODASYL-Norm. CODASYL steht für Conference on Data Systems Language. Das gleiche Komitee entwarf übrigens Jahre zuvor auch die Programmiersprache COBOL. Der CODASYL-Abschlussbericht erschien im Jahre 1971.

Die beiden in Deutschland wichtigsten Vertreter netzwerkartiger Datenbanken sind IDMS (Computer Associates) und UDS (Siemens-Nixdorf). Beide sind CODASYL-Datenbanken. UDS werden wir als Beispiel einer netzwerk-

artigen Datenbank im nächsten Abschnitt noch genauer kennenlernen. Zunächst wollen wir uns aber mit grundlegenden Eigenschaften netzwerkartiger Datenbanken beschäftigen. Wie bei praktisch allen Datenbanken lassen sich die Befehle in Zugriffs- und Definitionsbefehle unterteilen. In CODASYL-Datenbanken unterscheiden wir zwischen drei Datenbankbefehlsarten:

Schema DDL:	Beschreibt den internen Aufbau einschließlich physischer Zugriffswege
Subschema DDL:	Beschreibt ausschließlich die externe Benutzersicht
DML:	Besitzt eine umfangreiche Programmiersprachensyntax und wird meist in Cobol eingebettet

Beginnen wir mit der Datenbankbeschreibung. Netzwerke bestehen ganz allgemein aus einer Menge von Tabellen (Records) und einer Menge von Verbindungen (Sets). Die Records entsprechen den Relationen der relationalen Datenbanken. Neu sind die Verbindungen, wobei diese immer von einem Owner zum Member gerichtet sind. Im Prinzip sind beliebige Verbindungen denkbar: Eine Tabelle kann bezüglich der einen Verbindung Eigentümer, bezüglich einer anderen Mitglied sein. Sie kann insgesamt Eigentümer mehrerer Mitglieder, aber auch Mitglied mehrerer Eigentümer sein. Es ist auch mehr als eine Verbindung zwischen zwei Tabellen denkbar. Es kann also eine Tabelle gleich mehrfach Eigentümer eines Mitglieds sein und umgekehrt. Aber auch wechselseitige Verbindungen sind möglich: Eine Tabelle kann Mitglied und gleichzeitig Eigentümer zu ein und derselben Tabelle sein. Auch Selbstverweise auf sich selbst sind bei einigen Netzen erlaubt. Beispiele sind in Abb. 44 schematisch angegeben.

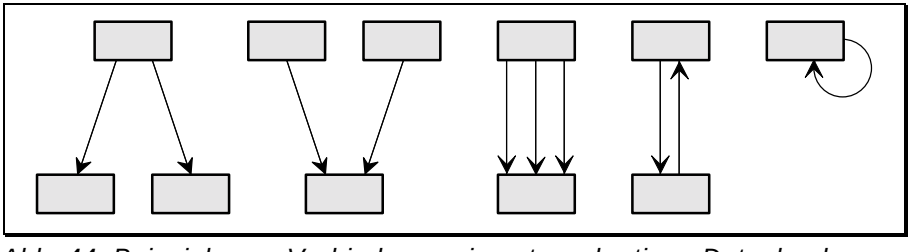


Abb. 44 Beispiele von Verbindungen in netzwerkartigen Datenbanken

Jeder einzelne Set, für sich betrachtet, verbindet genau zwei Records miteinander, nämlich einen Eigentümer mit einem Mitglied. Wie schon bei den hierarchischen Datenbanken müssen wir zwischen der Typdefinition und den einzelnen Elementen eines Typs unterscheiden. In Abb. 43 und Abb. 44 wur-

den jeweils die Tabellentypen, also Eigentümer- und Mitgliedstypen, angegeben. Hinter diesen Datenstrukturen verbergen sich natürlich in der Praxis eine Fülle von Einträgen.

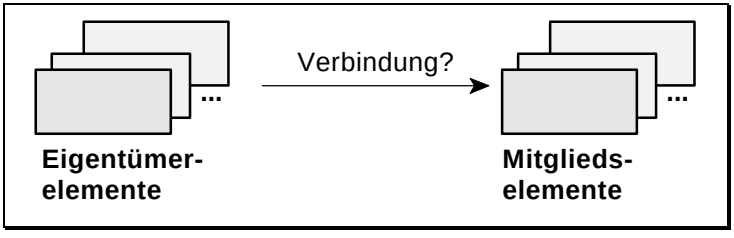


Abb. 45 Verbindung zwischen den einzelnen Elementen

Jeder Set zwischen einem Eigentümer- und einem Mitgliedselement enthält eine Vielzahl von Einträgen, wobei zu jedem Eigentümerelement genau ein Eintrag existiert. Dieser jeweilige Eintrag im Set verknüpft das Eigentümerelement mit allen seinen Mitgliedselementen. Es liegt demnach eine 1 zu m Beziehung vor. Besitzt ein Eigentümerelement momentan keine Verbindung, so ist im Set ein entsprechender Verweis vorhanden. Umgekehrt kann jedes Mitgliedselement bezüglich dieses einen Sets nur höchstens einem Eigentümerelement zugeordnet sein.

Wir wollen diesen Sachverhalt am Beispiel der Relationen *Verkaeufer* und *Produkt* aus Tab. 29 und Tab. 30 aufzeigen. Die bei den relationalen Datenbanken verwendete Relation *Verknuepfung* benötigen wir hier nur in einer abgespeckten Form, denn die Funktion der Fremdschlüssel wird von den Sets übernommen. Das Netz dieser einfachen Datenbank ist in Abb. 46 angegeben.

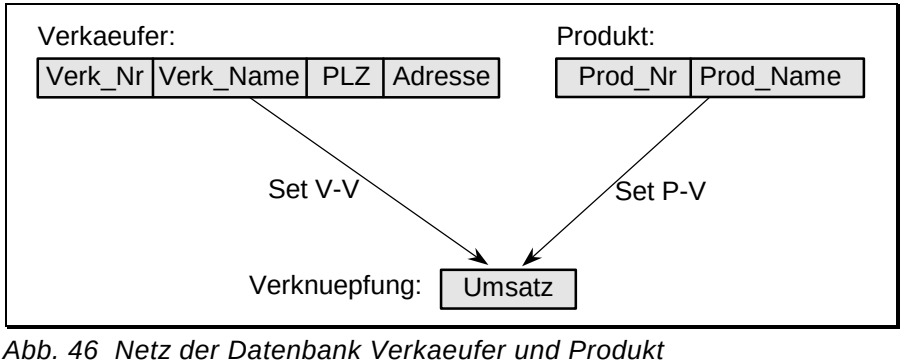


Abb. 46 Netz der Datenbank Verkaeufer und Produkt

Der Aufbau der Records *Verkaeufer*, *Produkt* und *Verknuepfung* ist ähnlich dem der Relationen in relationalen Datenbanken. Von Interesse sind die bei-

den Sets *V-V* und *P-V*. Diese stellen die Verbindungen zwischen den einzelnen Verkäufern und dem Umsatz bzw. zwischen den Produkten und dem Umsatz her. Der Aufbau dieser Verbindungen kann über verkettete Listen erfolgen. Abb. 47 zeigt eine solche Realisierung.

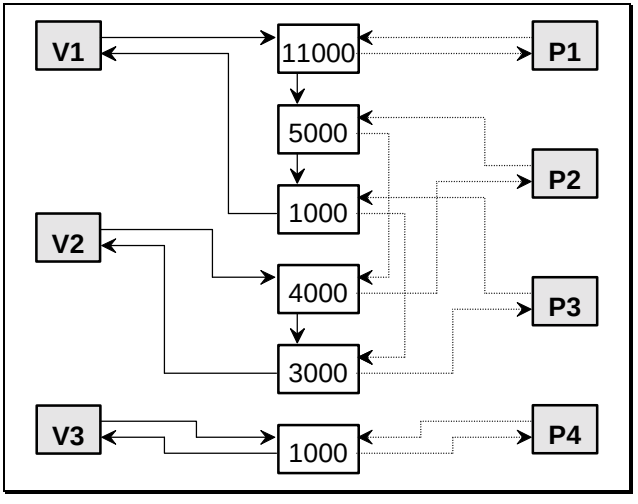


Abb. 47 Realisierung der Sets über verkettete Listen

Jeder Verkäufer ist Ausgangspunkt einer Liste, in der alle dazugehörigen Umsätze eingekettet sind. Die Liste endet wieder beim Verkäufer. Analog gilt dies auch für die einzelnen Produkte. Sollte ein Produkt noch nicht verkauft worden sein oder ein Verkäufer noch nichts verkauft haben, so besteht diese Kette nur aus einem Selbstverweis. In Abb. 47 ist die Verkettung der Einträge des Sets *V-V* mit durchgezogenen Strichen gekennzeichnet, die Verkettung der Einträge des Sets *P-V* mit gestrichelten Linien.

Dieses Beispiel zeigt deutlich, dass der Aufwand bei netzwerkartigen Datenbanken im Aufbau der Sets steckt. Dieser Aufwand ist hoch. Doch es gibt zusätzliche Schwierigkeiten, die es zu überwinden gilt:

- Das Netz besitzt keine Wurzel, wir haben also keinen definierten Einstieg wie bei hierarchischen Datenbanken.
- Nicht von jedem Record gelangen wir (via Sets) zu allen anderen Records.
- Es fehlt eine Anordnung wie bei hierarchischen Datenbanken. Die Daten müssen aber trotzdem so gespeichert werden, dass sie jederzeit wieder auffindbar sind.

Ehrlichkeitshalber dürfen wir nicht verschweigen, dass auch bei relationalen Datenbanken auf der physischen Ebene ähnliche Probleme existieren. In Netzwerken behilft man sich meist damit, dass eine sogenannte *System-Root* eingeführt wird. Diese enthält all diejenigen Records (und weitere) als Mitglied, die selbst nirgends Mitglied sind. Eine Lösung für unser einfaches Netzwerk von Verkäufern und Produkten könnte dann wie in Abb. 48 aussehen.

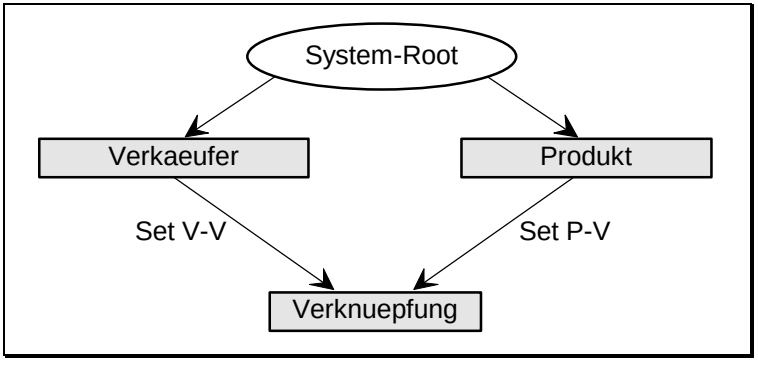


Abb. 48 Netz der Datenbank Verkaeufer und Produkt mit System-Root

Beenden wir damit unseren Überblick über die Datenbeschreibung und wenden uns noch kurz der Datenmanipulation zu. Das Netzwerk gibt die hier möglichen Bewegungen innerhalb der Daten vor. Einige typische Operatoren in Netzen sind:

- Bewegung von einem Eigentümer zum 1. Mitglied entlang eines vorgegebenen Sets (z.B. von *V1* nach 11000 in Abb. 47).
- Bewegung von einem Mitglied zum nächsten entlang eines Sets (z.B. von 5000 nach 1000 entlang *V-V* oder von 5000 nach 4000 entlang *P-V* in Abb. 47).
- Bewegung von einem Mitglied zum Eigentümer entlang eines Sets (z.B. von 4000 nach *P2* in Abb. 47).
- Bewegung zu einem bestimmten Record, um ein bestimmtes Feld zu lokalisieren.
- Erzeugen, Löschen und Ändern von Recordeinträgen.
- Aufbauen, Löschen und Ändern von Verbindungseinträgen, d.h. Einketten, Ausketten und Umketten innerhalb der einzelnen Zeigerketten und Erzeugen neuer Verkettungen.

Diese Befehle lassen einen ganz wichtigen Unterschied zu relationalen Datenbanken erkennen: CODASYL-Datenbanken liegt eine prozedurale Zugriffssprache zugrunde. Wir beschreiben den Weg, um ein bestimmtes Datum zu finden. Wir sind uns also der Sets genau bewusst und beschreiben die Suche bestimmter Einträge entlang dieser Sets (gehe zum ersten Mitglied und dann entlang eines bestimmten Sets zum nächsten usw.). Ganz anders verhielt sich SQL. Dort mussten wir nur angeben, was wir haben wollten, und nicht wie wir zu suchen haben.

Auch in netzwerkartigen Datenbanken gibt es Integritätsregeln. Allerdings sind sie weitergespannt als bei hierarchischen Systemen. Ein Ändern eines Owner-Elements führt zwar auch hier zum Ändern der Mitglieder. Wegen der recht allgemeinen Verkettung sind jedoch beim Löschen alle Regeln denkbar. Ebenso dürfen Elemente Nulleinträge besitzen. Wir erhalten zusammenfassend (in der Terminologie relationaler Datenbanken):

```
NULL / NOT NULL
ON DELETE  RESTRICT / CASCADE / SET NULL
ON UPDATE CASCADE
```

Im Folgenden wollen wir noch etwas näher die Beschreibungs- und Zugriffssprache am Beispiel der Datenbank UDS betrachten. Doch schon jetzt ist erkennbar, dass auch Netze sehr komplex sind, allerdings wesentlich flexibler in den Änderungs- und Erweiterungsmöglichkeiten als reine Baumstrukturen. Die Performance ist derjenigen relationaler Datenbanken meist deutlich überlegen. Doch die Komplexität wird schon dadurch erkennbar, dass wir zwei völlig verschiedene Strukturen verwalten müssen, Records und Sets!

10.5 CODASYL Datenbank UDS

Die Datenbank UDS (Universelles Datenbanksystem) ist eine leistungsfähige Großrechnerdatenbank der Firma Siemens-Nixdorf. Diese Datenbank ist auf dem Großrechnerbetriebssystem BS2000 verfügbar. Zusätzlich existiert das Produkt UDS-SQL, das dieser Datenbank eine komplette SQL Oberfläche verleiht. Aus Sicht unserer Definition von relationalen Datenbanken können wir UDS-SQL daher auch als relationale Datenbank bezeichnen. Dies gilt jedoch nicht für den internen Aufbau von UDS. Wir wollen am Beispiel der Datenbank UDS den Aufbau einer CODASYL-Datenbank etwas näher kennenlernen. Beginnen wir dazu zunächst mit der Beschreibungssprache (DDL).

Die Definition der Datenbank besteht aus Record- und Set-Beschreibungen (Tabellen und Beziehungen). Die Records ähneln sehr den Relationen aus relationalen Datenbanken. In Tab. 52 ist ein Record mit einigen Einträgen angegeben.

Tab. 52 Beispiel des Records UDS-Kunde

KUNDEN-NR	KUNDEN-NAME	STRASSE	PLZ	ORT
01020	Alte Leipziger	Alte Leipziger-Platz	61440	Oberursel
12149	Black & Decker	Black & Decker - Str.	65510	Idstein/TS
01306	Gerling-Konzern	Gereonshof 14-16	50670	Köln

Die Deklaration dieses Records geschieht in UDS wie folgt:

```
RECORD NAME IS   UDS-KUNDE
   LOCATION MODE IS CALC USING KUNDEN-NR
   WITHIN        KUNDENREALM
   KUNDEN-NR     PICTURE IS 9 (6).
   KUNDEN-NAME   PICTURE IS X (30).
   STRASSE       PICTURE IS X (30).
   PLZ           PICTURE IS X (6).
   ORT           PICTURE IS X (30).
```

Dieser Record enthält in der ersten Zeile den Namen des Records, in der zweiten wird *Kunden-Nr* als Primärschlüssel angegeben, nach dem die Sätze sortiert sind. In der Folgezeile steht die Bezeichnung *Kundenrealm*. Eine UDS-Datenbank ist in einzelne *Realms* unterteilt. Dies sind Verwaltungseinheiten der Datenbank und dienen sowohl dem Datenschutz als auch einer optimalen Aufteilung der Daten auf mehrere Dateien und damit auch auf mehrere Magnetplatten. Die Definition der Spalten erfolgt in den Folgezeilen. Dabei steht die Zahl 9 für numerische Eingabebereiche, der Buchstabe *X* für alphanumerische Eingabefelder.

Für uns ganz neu sind die Set-Beschreibungen. Die Typbeschreibung besteht aus einem gerichteten Pfeil vom Owner zum Member. Das Beispiel eines Sets *Equipment* ist im linken Teil von Abb. 49 zu sehen. Wir wissen bereits, dass in diesem Set zu jedem Element des Owners ein Eintrag existiert. Schematisch ist im rechten Teil von Abb. 49 ein einzelner Eintrag, der auch als Set-Occurrence bezeichnet wird, eingezeichnet. Dieser Set zeigt also auf, welche Rechneranlagen die Kunden im Einsatz haben.

Die Typdefinition dieses Sets *Equipment* hat in UDS folgende Gestalt:

```
SET NAME IS      EQUIPMENT
   OWNER IS      UDS-KUNDE.
   MEMBER IS     ZENTRALEINHEIT    OPTIONAL MANUAL
      SEARCH KEY IS ZE-TYP
                USING INDEX
                DUPLICATES ARE ALLOWED
      SEARCH KEY IS AUSBAU
                USING INDEX
                DUPLICATES ARE ALLOWED
   SET OCCURRENCE SELECTION IS THRU CURRENT OF SET.
```

In dieser Definition werden der Name des Sets, der Eigentümer und das Mitglied festgelegt. Es werden die Suchschlüssel, die Verwendung von Indizes und die Erlaubnis von Mehrfachnennungen angegeben. Die letzte Zeile gibt darüberhinaus Auskunft über die Reihenfolge der Speicherung der Set-Occurrences.

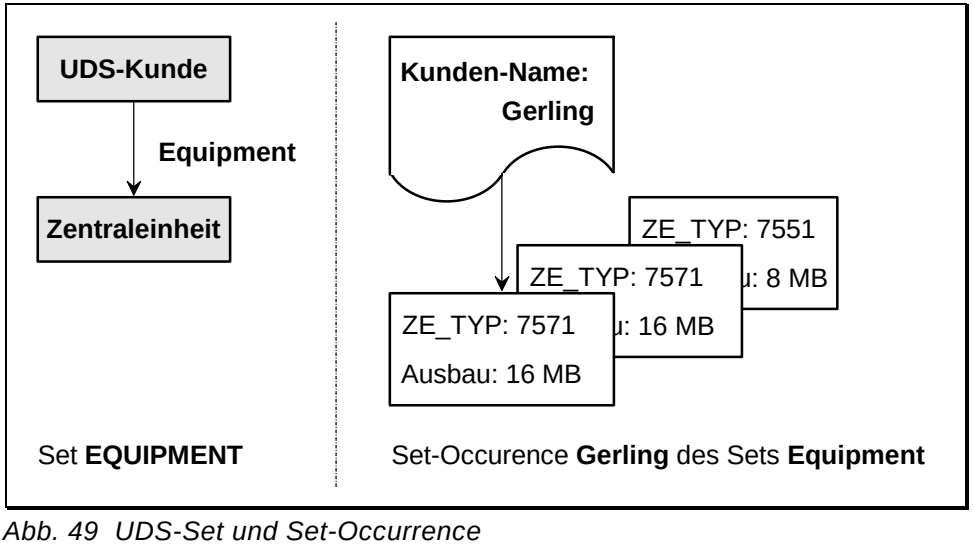


Abb. 49 UDS-Set und Set-Occurrence

Diese Beschreibung eines Sets gibt die logische Darstellung an, allerdings verknüpft mit einigen physischen Eigenschaften (z.B. Index). Wir haben im vorherigen Abschnitt in Abb. 47 bereits gesehen, dass wir uns die Darstellung der Sets immer als Kette von einem Eigentümereintrag zu jedem seiner Mitgliedseinträge vorstellen können. Die Möglichkeiten der tatsächlichen physischen Speicherung sind in UDS ähnlich vielfältig wie bei IMS. Einige Möglichkeiten sind im Folgenden angegeben:

- Die Mitgliedseinträge stehen physisch direkt hinter den Eigentümereinträgen.
- Die Einträge sind miteinander verkettet (analog Abb. 47).
- Zeiger auf die Mitgliedseinträge stehen physisch direkt hinter den Eigentümereinträgen.

Eine mögliche physische Realisierung des Set-Occurrence *V1* aus Abb. 47 ist in Abb. 50 dargestellt. Hier sind die einzelnen Mitglieder über ein nach dem Suchschlüssel sortiertes Zeigerfeld direkt erreichbar. Die Mitglieder enthalten in unserem Fall allerdings nur den einzigen Feldeintrag *Umsatz*. Würden demhingegen die Mitglieder aus mehreren Feldern bestehen, so wäre eines dieser Felder der Suchbegriff, nach dem die Zeiger sortiert sind. Bei vielen Einträgen wird ein Index auf diese Suchbegriffe das Suchen noch weiter beschleunigen.

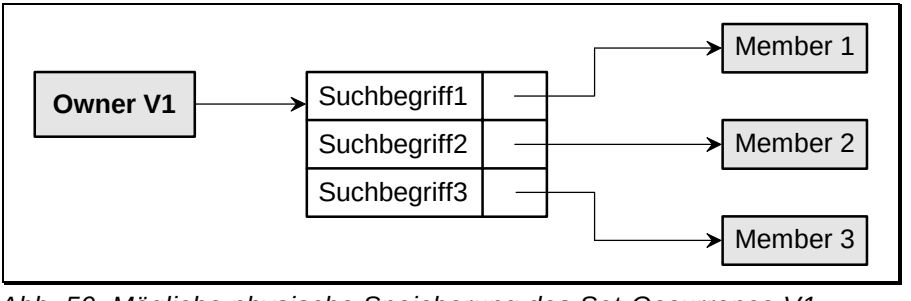


Abb. 50 Mögliche physische Speicherung des Set-Occurrence V1

Zur Optimierung der Zugriffszeiten existieren in UDS wie in allen anderen größeren Datenbanksystemen viele zusätzliche Möglichkeiten. Erwähnt sei hier nur die Zwischenspeicherung von häufig benötigten Daten im Arbeitsspeicher, so dass die relativ langsamen Plattenzugriffe deutlich reduziert werden können. Selbstverständlich sorgt das System dafür, dass es bei Rechnerausfällen nicht zu Inkonsistenzen wegen dieser verlorengegangenen Daten im flüchtigen Speicher kommt.

Nach dieser kleinen Einführung in die Beschreibungssprache DDL wollen wir noch eine Übersicht über die DML-Befehle in UDS geben. Die wichtigsten Befehle sind in Tab. 53 zusammengefasst. Die erste Gruppe zur Befehlssteuerung befasst sich mit dem Anschließen an einen Datenbankbereich und dem Beenden. Ferner können Sätze (Tupel) für andere Benutzer gesperrt und diese Sperren wieder freigegeben werden (*Keep* und *Free*).

Tab. 53 Übersicht über die DML-Befehle in UDS

Befehle zur Steuerung und zum Satzschutz:

READY datenbankbereich
FINISH
IF set-bedingung
KEEP
FREE

Befehle zur Änderung der Informationen:

STORE satzname
ERASE satzname
CONNECT satzname **TO** setname
DISCONNECT satzname **FROM** setname
MODIFY satzname

Befehle zur Wiedergewinnung von Informationen:

ACCEPT
FETCH DATABASE-KEY
FETCH ANY satzname
FETCH DUPLICATE satzname
FETCH FIRST / **LAST** / **NEXT** / **PRIOR** satzname
FETCH CURRENT satzname
FETCH OWNER WITHIN setname
FETCH satzname **USING** feldnamensliste/suchausdruck

Zum Ändern von Informationen müssen wir beachten, dass wir Record-Einträge ändern können (*Store*, *Erase*, *Modify*), aber auch Set-Einträge (*Connect*, *Disconnect*). Auch beim lesenden Zugriff müssen wir uns über die Zugriffswege entlang der Sets im Klaren sein. In Abb. 51 haben wir einige der wichtigsten Suchbefehle (*Fetch*) anschaulich dargestellt.

Die Programmierung erfolgt überwiegend in COBOL, heute aber auch in C oder anderen höheren Programmiersprachen. Zu beachten ist immer, dass es zwei verschiedene Zugriffe gibt, die über Records und die über Sets. Dies verdoppelt die Befehlssyntax beispielsweise gegenüber SQL. Auch ist der Aufwand zum Definieren von Sets sehr hoch. Dafür erhalten wir eine gegenüber relationalen Datenbanken deutlich höhere Systemleistung und eine gegenüber hierarchischen Datenbanken wesentlich flexiblere Struktur. Als Nachteil gegenüber relationalen Datenbanken müssen wir eine komplexe Definitionssprache und eine relativ aufwendige Programmierung der CODASYL-Datenbanken in Kauf nehmen.

Diese Beispiele zeigen deutlich, dass die physische Struktur dem Benutzer nicht verborgen bleibt. Außerdem ist der prozedurale Aufbau der Programmie-

rung zu erkennen: Wir müssen programmieren, wie wir uns durch die Datenbank navigieren.

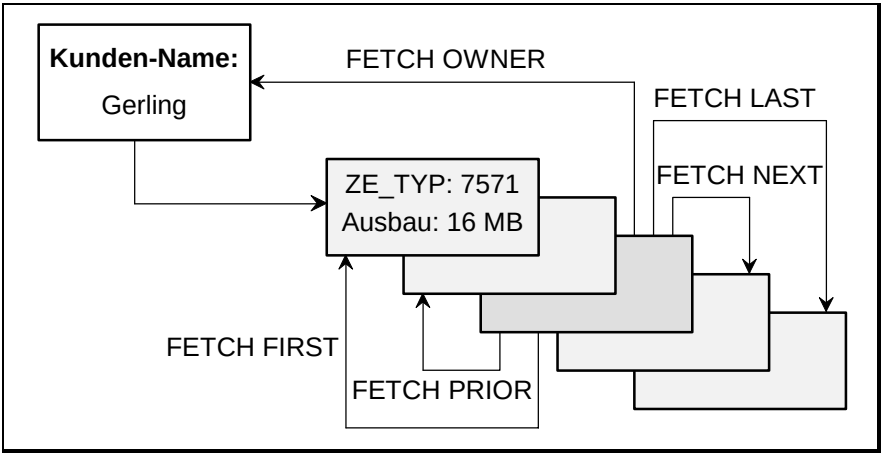


Abb. 51 Einige Zugriffe mit dem FETCH-Befehl in UDS

10.6 Übungsaufgaben

1) Praktisch jeder Datenpool kann mittels eines geeigneten Verwaltungssystems in eine Datenbank (invertierte Liste) überführt werden. Diese Überführung ist aber enorm komplex und umfangreich. Warum?
2) Primärschlüssel existieren in relationalen wie auch in hierarchischen und netzwerkartigen Datenbanken. Es gibt neben Gemeinsamkeiten auch Unterschiede in der Definition und Verwendung dieser Schlüssel in den einzelnen Datenbanksystemen. Beschreiben Sie die Gemeinsamkeiten und die Unterschiede.
3) Eigene Sekundärschlüssel wie in IMS gibt es in relationalen Datenbanken nicht. Welche weiterreichenden Möglichkeiten bieten stattdessen relationale Datenbanken?
4) Sekundärschlüssel haben wir auch in der Datenorganisation kennengelernt. Vergleichen Sie diese mit dem Sekundärschlüssel in IMS. Geben Sie Gemeinsamkeiten und Unterschiede an.
5) Fremdschlüssel sind der „Kit“ zwischen den Entitäten in relationalen Datenbanken. Wie wird in hierarchischen, wie in netzwerkartigen Systemen „gekittet“?
6) CODASYL-Datenbanken besitzen wesentlich mehr DML- und DDL-Befehle als relationale Datenbanken. Woran liegt dies?

7) Die Entitäten in relationalen Datenbanken sind die Relationen, in netzwerkartigen die Records. Beschreiben Sie Gemeinsamkeiten und Unterschiede zwischen Relationen und Records.
8) Kann eine CODASYL-Datenbank in eine relationale Datenbank überführt werden? Gilt auch die Umkehrung?
9) Worin liegen die Stärken, worin die Schwächen hierarchischer Datenbanken?
10) Woran mag es liegen, dass viele Anwender von netzwerkartigen Datenbanken auf relationale umsteigen möchten? Woran scheitert diese Umstellung häufig?

11 Moderne Datenbankkonzepte

In den letzten Jahren stieg das Interesse sowohl für verteilte als auch für objektorientierte Datenbanken erheblich. Zum Einen werden die Aktivitäten von Firmen auch im Bereich der Datenverarbeitung immer weiter gestreut, und zum Anderen stellt sich die nicht unberechtigte Frage, ob denn relationale Datenbanken für alle Bereiche die optimalen Speichermodelle sind. Wir wollen daher dieses Kapitel den beiden Themen Verteilung und Objektorientierung widmen. Das Ziel dieses Kapitels ist, die Idee dieser beiden modernen Datenbankkonzepte kennenzulernen. Mit diesen Basisinformationen wird der Leser die Anwendungsmöglichkeiten abschätzen können, vor allem wird er auch mit den Problemen dieser beiden Konzepte konfrontiert.

In verteilten Datenbanken werden die Daten nicht wie bisher zentral gespeichert, sondern auf mindestens zwei Rechner verteilt. Es versteht sich fast von selbst, dass dadurch eine weitere Komplexitätsstufe hinzukommt, nämlich die Koordination der Rechner untereinander. Gleichzeitig steigen aber auch die Einsatzmöglichkeiten, da in der Praxis oft Daten auf vielen unterschiedlichen Rechnern gehalten werden, denken wir nur an eine Firma mit mehreren Filialen.

Hinter objektorientierten Datenbanken steht ein weiteres Datenbankmodell und damit eine neue Technik, Daten zu verwalten. Während in den 80er Jahren viele hierarchische und netzwerkartige durch relationale Systeme verdrängt wurden, scheinen die 90er Jahre, zumindest in Teilgebieten, den objektorientierten Datenbanken zu gehören. Einige Überlegungen zu objektorientierten Ansätzen und Vergleiche zu relationalen Datenbanken werden in diesem Kapitel besprochen. Neuerungen in Oracle V8.0 werden vorgestellt.

11.1 Verteilte Datenbanken

Bei der Überlegung zwecks Verteilung von Daten auf mehrere Rechner werden wir hauptsächlich mit zwei Fragen konfrontiert:

- Worin liegt der Vorteil der getrennten Datenhaltung gegenüber der zentralen Speicherung?

- Wie funktioniert die Verwaltung verteilter Datenbanken bzw. inwieweit muss der Datenbankbenutzer beim Umstieg auf verteilte Systeme umdenken?

Wir werden im Folgenden auf diese beiden wichtigen Fragen eingehen. Bei der Beantwortung der zweiten Frage steht in der Praxis die Forderung im Vordergrund, dass der Benutzer möglichst wenig von der Verteilung bemerkt. Dies führte zur Aufstellung von 12 Regeln, die in einem verteilten System erfüllt sein sollten. Wir stellen diese im übernächsten Unterabschnitt vor und gehen danach auf die Schwierigkeiten ein, diese Regeln zu erfüllen.

11.1.1 Vorteile der verteilten Datenhaltung

Der Hauptgrund für die Verteilung liegt darin begründet, dass die meisten Unternehmen nicht ausschließlich zentral verwaltet werden. Es existieren meist mehrere Fabriken, Werke oder Zweigstellen. Aber auch bei kleineren Firmen gibt es Unternehmensbereiche, Abteilungen oder zumindest verschiedene Projekte. Daraus folgt direkt, dass bereits die anfallenden Daten verteilt sind. Es ist quasi ein Schritt zurück, diese verteilten Daten zentral zusammenzufassen. Und wir fragen uns unwillkürlich, ob dies im Zeitalter der Vernetzung überhaupt noch notwendig ist.

Ein Vorteil dezentraler Datenhaltung liegt nämlich darin, dass häufig lokal zugegriffen wird. Denken wir nur etwa an ein Geldinstitut mit vielen Zweigstellen. Die meisten Nachfragen und Geldbewegungen geschehen innerhalb der einzelnen Zweigstellen. Nur wenige Buchungen erfolgen extern.

Ein weiterer Vorteil der Verteilung von Daten liegt in der Ausfallsicherheit. Setzen wir voraus, dass das verteilte System keinen zentralen Server besitzt, so wird der Ausfall eines beliebigen Rechners das Netz nicht vollständig lahmlegen. Im Gegenteil, bis auf die Daten des einen Rechners bleiben alle weiteren zugreifbar. Damit wird eine hohe Ausfallsicherheit garantiert. Hier liegt andererseits die Achillesferse zentraler Systeme. Der Ausfall des zentralen Datenbankservers legt das gesamte System lahm.

11.1.2 Die zwölf Regeln zur verteilten Datenhaltung

Bereits die ersten Veröffentlichungen zu verteilten Datenbanken beschäftigten sich mit der Frage, wie sichergestellt werden kann, dass der Benutzer mit

den Problemen der Verteilung nicht konfrontiert wird. Wir bezeichnen dies als das fundamentale Prinzip verteilter Datenbanken:

Fundamentales Prinzip verteilter Datenbanken

☞Ein verteiltes System sollte sich dem Anwender gegenüber genauso wie ein nichtverteiltes verhalten.

Um dieses Prinzip zu garantieren, stellte J. F. Date einen Forderungskatalog für verteilte Datenbanken auf. Dieser Katalog (siehe [Date90]) umfasst 12 Regeln, die idealerweise erfüllt sein sollten. Sie sind in Tab. 54 aufgelistet.

Tab. 54 Die zwölf Regeln von Date

1.	Lokale Eigenständigkeit jedes Rechners
2.	Keine zentrale Verwaltungsinstanz
3.	Ständige Verfügbarkeit
4.	Lokale Unabhängigkeit
5.	Unabhängigkeit gegenüber Fragmentierung
6.	Unabhängigkeit gegenüber Datenreplikation
7.	Optimierte verteilte Zugriffe
8.	Verteilte Transaktionsverwaltung
9.	Unabhängigkeit von der Hardware
10.	Unabhängigkeit von Betriebssystemen
11.	Unabhängigkeit vom Netz
12.	Unabhängigkeit von den Datenverwaltungssystemen

Diese Regeln sind in einem verteilten System nicht alle zwingend notwendig, aus dem einen oder anderen Grund aber doch wünschenswert. Wir gehen auf diese 12 Regeln im Folgenden kurz ein:

1) Lokale Eigenständigkeit jedes einzelnen Rechners
   Jeder einzelne Rechner im verteilten System besitzt eine maximal mögliche Autonomie. Dies bedeutet insbesondere, dass ein lokaler Zugriff auf lokal gespeicherte Daten nicht misslingen sollte, weil ein anderer Rechner momentan nicht zugreifbar ist. Dieses Verhalten garantiert Ausfallsicherheit, verlangt aber, dass alle lokal gespeicherten Daten auch lokal verwaltet werden. Dies impliziert auch die lokale Garantie von Integrität, Sicherheit und Transaktionsmechanismen.

2) Keine zentrale Instanz, die das System leitet und verwaltet
Aus der lokalen Eigenständigkeit folgt direkt, dass es keine zentrale Instanz geben darf, die die verteilte Datenbank verwaltet. Doch auch wenn die Regel 1 nicht voll erfüllt sein sollte, ist eine zentrale Verwaltung nicht wünschenswert: Das Gesamtsystem wird verwundbar, sei es, dass die zentrale Instanz ausfällt oder nur, dass dieser zentrale Rechner zum Engpass wird.

3) Ständige Verfügbarkeit
In einem verteilten System sollte es nie erforderlich sein, aus Gründen der Datenbankverwaltung das gesamte System oder auch nur Teile davon gelegentlich abzuschalten.

4) Lokale Unabhängigkeit
Mit lokaler Unabhängigkeit ist gemeint, dass der Benutzer nicht wissen muss, wo die einzelnen Daten gespeichert sind. Diese Regel erleichtert die Programmierung erheblich, da die gleichen Programme auf allen Rechnern der verteilten Datenbank ohne Anpassung ablaufen können. Diese Regel impliziert weiterhin, dass alle gewünschten Daten jederzeit auf jeden beliebigen Rechner zwecks weiterer Bearbeitung geholt werden können.

5) Unabhängigkeit gegenüber Fragmentierung
Fragmentierung heißt, dass auch vorgegebene Dateneinheiten, etwa Relationen, auf mehrere Rechner verteilt sein können. Denken wir nur an eine große Personaltabelle, in der alle Mitarbeiter einer Firma geführt werden. Aus Performancegründen ist es empfehlenswert, die Mitarbeiterdaten der einzelnen Werke lokal zu halten, die Relation *Personal* also auf mehrere Rechner zu verteilen. Genau diese Möglichkeit fordert diese fünfte Regel. Das Wort „Unabhängigkeit“ bedeutet, dass der Benutzer nicht merken soll, ob Daten fragmentiert sind oder nicht.

6) Unabhängigkeit gegenüber Datenreplikation
Unter Datenreplikation verstehen wir, dass Kopien von Daten auf mehreren Rechnern gleichzeitig gehalten werden dürfen. Dies verbessert sowohl die Performance, falls häufig auf die gleichen Daten lokal zugegriffen wird, als auch die Verfügbarkeit. Die Replikation besitzt aber auch einen schwerwiegenden Nachteil: Werden die Daten in ei-

ner Kopie geändert, so müssen auch alle anderen Kopien angepasst oder zumindest invalidiert werden. Die Verwaltung dieser Replikate und deren Invalidierung bedingen einen hohen Aufwand. Wieder ist diese Verwaltung alleinige Aufgabe des verteilten Datenbanksystems, das sich nach Aussage dieser Regel für den Benutzer genauso zu verhalten hat wie ein System, das Replikate nicht unterstützt.

7) Optimierte verteilte Zugriffe
Zugriffsoptimierung ist in einem verteilten System viel wichtiger als in einem zentralen, da die Daten über (langsame) Netze transportiert werden müssen. Optimierung heißt, dass jeder einzelne Zugriff möglichst ohne Umwege erfolgen sollte, und dass die Anzahl der Zugriffe für eine Anfrage minimiert wird. Die Minimierung der Zugriffe erfolgt besonders einfach mit relationalen Datenbanken. Wollen wir etwa in München alle Artikel wissen, die im Lager in Hamburg vorrätig sind, so genügt ein Befehl. Dies impliziert nur 2 Bewegungen: Die Anfrage von München nach Hamburg und die Rückgabe der Ergebnisrelation in umgekehrter Richtung. Die meisten nichtrelationalen Systeme benötigen $2n$ Bewegungen, wenn n die Anzahl der unterschiedlichen Artikel ist.

8) Verteilte Transaktionsverwaltung
Auch in verteilten Systemen sind Transaktionen atomare Einheiten. Recovery und Concurrency müssen ebenso unterstützt werden. Wir haben dazu in Abschnitt 7.2 bereits den Zwei-Phasen-Commit kennengelernt. Er basiert darauf, dass zunächst alle betroffenen Rechner des verteilten Systems eine lokale Transaktion ausführen und erst dann, wenn alle Teilsysteme mit einem Commit enden, auch global ein Commit durchgeführt wird. Dies erfordert eine globale Kontrolle und widerspricht damit der zweiten Regel. Wir diskutieren dies ausführlich im nächsten Unterabschnitt.

9) Unabhängigkeit von der verwendeten Hardware
Diese Forderung ist heute bereits Alltag. PCs kommunizieren mit UNIX-Rechnern, UNIX-Rechner mit Großrechnern und wiederum Großrechner mit PCs. Alle diese Rechner besitzen eine unterschiedliche Hardware-Plattform.

10) Unabhängigkeit von den verwendeten Betriebssystemen
Diese Regel ist im Wesentlichen nur ein Unterpunkt der vorhergehenden. Windows, UNIX, MVS oder BS2000, um nur einige zu nennen, arbeiten einträchtig miteinander.

11) Unabhängigkeit vom verwendeten Netzwerk
Auch dem ist nichts hinzuzufügen.

12) Unabhängigkeit von den Datenbankverwaltungssystemen
Diese Regel ist noch nicht ganz so selbstverständlich wie die vorhergehenden. Doch auch hier haben dank einheitlicher Protokolle die einzelnen Datenbankhersteller zueinandergefunden. Als Standard etablieren sich dabei immer mehr die Schnittstellen CORBA, ODBC und JDBC. Die Kommunikationssprache zwischen den einzelnen Datenbankverwaltungssystemen ist hier SQL!

Leider ist es alles andere als einfach, alle 12 Regeln gleichzeitig zu erfüllen. In der Praxis werden daher meist die eine oder andere oder gleich mehrere Regeln fallengelassen. Einige Details werden im nächsten Unterabschnitt besprochen.

Doch vorher überlegen wir uns, welche Datenbanksysteme die verteilte Welt am besten unterstützen. Sicherlich sind die Regeln 9 bis 11 von allen Datenbanksystemen gleich gut zu erfüllen, auch die Regeln 1 bis 4 und 6 spielen für diese Überlegung nur eine untergeordnete Rolle. Um jedoch die Regeln 5, 7 und 12 optimal zu unterstützen, benötigen wir relationale Datenbanken. Auch die im nächsten Abschnitt behandelten objektorientierten Datenbanken zeigen hier Schwächen.

Begründung: Zum Einen ist es heute Standard, mittels SQL zwischen unterschiedlichen Datenbanksystem zu kommunizieren (Regel 12). Aber auch die Fragmentierung (Regel 5) verlangt, dass Relationen mittels Projektionen und Restriktionen beliebig zerlegt und auf verschiedene Rechner verteilt werden können. Dies ermöglichen in einfachster Weise relationale Datenbanken dank ihrer „flachen“ Struktur. Und wie bereits in Regel 7 direkt angesprochen unterstützen SQL und damit alle relationalen Datenbanken diese Regel praktisch automatisch. Wir folgern:

Wichtig

☞ Als verteilte Datenbanken im Sinne der obigen 12 Regeln kommen in der Praxis nur relationale Datenbanken in Frage.

11.1.3 Probleme verteilter Datenbanken

Wir haben im letzten Unterabschnitt bereits einige Probleme in verteilten Datenbanken angesprochen. All diese Probleme haben gemeinsam, dass zum Einen die Übertragungsgeschwindigkeit im Netz erheblich niedriger ist als der Zugriff auf lokale Speichermedien. Und zum Anderen müssen sich die einzelnen Rechner koordinieren, was das Netz weiter belastet. Wir müssen leistungsfähige verteilte Datenbanken daher immer unter dem Gesichtspunkt sehen, dass sowohl die Menge der über das Netz übertragenen Daten als auch der Nachrichtenaustausch zwischen den Rechnern zu minimieren ist. Betrachten wir dazu drei besonders problematische Fälle in verteilten Datenbanken:

Problem der Datenverwaltung
In verteilten Datenbanken können gesuchte Daten über mehrere Rechner gestreut sein. Die Regel 4 verlangt, dass Programme nicht von vornherein davon ausgehen dürfen, auf welchen Rechnern die gewünschten Daten zu finden sind. Es muss daher eine Datenverwaltung existieren, die darüber Auskunft gibt. Diese Datenverwaltung unterliegt allerdings Zwängen:

- Die Datenverwaltung sollte nicht auf einem fest vorgegebenen Rechner liegen, da sonst Regel 2 verletzt wäre. Würde dieser eine Rechner ausfallen, so wäre das gesamte System lahmgelegt.
- Die Datenverwaltung sollte nicht als Kopie auf jedem Rechner vorliegen, da dies den Nachrichtenaustausch enorm erhöhen würde; denn jede lokale Änderung im Datenaufbau müsste auch in allen anderen Rechnern nachvollzogen werden.
- Die lokalen Daten jedes einzelnen Rechners sollten nicht ausschließlich lokal verwaltet werden; denn jeder Zugriff auf entfernte Daten würde eine Suche in gesamten Rechnerverbund erfordern, um diese Daten zu lokalisieren.

Wir sehen, dass jede dieser drei naheliegenden Verwaltungen neben wenigen Stärken erhebliche Schwächen aufweist. In verteilten Systemen, die diesen Namen auch verdienen, müssen daher neue Algorithmen zur Datenverwaltung gesucht werden. Eine auch in der Praxis angewandte Lösung ist:

Jedes Datum wird dem Rechner zugeordnet, wo es erstellt wurde. Diese Information ist in jedem angeschlossenen Rechner lokal verfügbar und wird erst beim Löschen dieses Datums wieder entfernt. Wird dieses Datum von irgend-

einem Rechner angefordert, so wendet sich dieser nach der lokalen Recherche an diesen angegebenen Rechner. Sollte das Datum inzwischen auf einen dritten Rechner migriert sein, so wird dieser Rechner darüber immer Bescheid wissen. In der Praxis sind dann ein, höchstens aber zwei Zugriffe auf entfernte Rechner erforderlich. Und beim Migrieren eines Datums muss immer nur der „Heimatrechner“ informiert werden.

Diese beschriebene Verwaltung verstößt nicht gegen Regel 2. Schließlich existiert kein zentraler „Heimatrechner“, denn Daten können auf allen Rechnern erstellt werden.

Übrigens lässt sich mit diesem Algorithmus auch das Problem der Replikation halbwegs performant lösen. Sind Replikate zugelassen, so wird im „Heimatrechner“ vermerkt, wo Replikate zu finden sind. Soll nun ein Replikat geändert werden, so wird vom „Heimatrechner“ die Erlaubnis geholt. Dieser wird vermerken, dass ein Replikat geändert wurde, und alle anderen für ungültig erklären. Der Zugriff auf ein anderes Replikat wird dann dazu führen, dass der „Heimatrechner“ zunächst eine gültige Kopie an den entsprechenden Rechner schicken muss. Dieser „Heimatrechner“ übernimmt auch die Sperrverwaltung zur Vermeidung unerlaubter paralleler Zugriffe.

Problem des globalen Transaktionsbetriebs

Bereits in Abschnitt 7.2 haben wir gesehen, wie mittels des Zwei-Phasen-Commits eine globale Transaktion durchgeführt werden kann. Der dort behandelte Algorithmus ging von einem zentralen Koordinator aus, der die globale Transaktion (Phase 2) steuert. Dies widerspricht jedoch der Regel 2! Wir sollten daher den globalen Transaktionskoordinator fallen lassen. Stattdessen bietet es sich an, die globale Transaktion von dem Rechner durchführen zu lassen, der die Transaktion startete.

Leider löst dies nicht alle Probleme. Erstens ist weiterhin die lokale Eigenständigkeit (Regel 1) jedes an der Transaktion beteiligten Rechners verletzt, da diese vom Koordinator abhängen. Zweitens gibt es kein endliches Protokoll, das die korrekte Beendigung einer globalen Transaktion (*Commit* oder *Rollback*) in jedem nur denkbaren Fehlerfall garantiert. Dies ist leicht beweisbar: Nehmen wir dazu an, dass ein korrektes Zwei-Phasen-Commit-Protokoll existiert, und dass dazu genau n Nachrichten zwischen den Rechnern ausgetauscht werden müssen. Geht nun gerade diese n-te Nachricht verloren, so war diese entweder nicht notwendig, was der Annahme widerspricht, oder das Protokoll ist nicht vollständig und damit fehlerhaft.

Wir erkennen, dass die Fehlerfreiheit des globalen Transaktionsbetriebs nicht unter allen Umständen garantiert werden kann. Ein gewisses Unbehagen

bleibt. Wir sehen aber auch, dass die Regel 1 nicht immer vollständig erfüllt werden kann. Wir haben daher auch nur von einer maximal möglichen Autonomie gesprochen.

Problem der Minimierung des Netzverkehrs

Wie bereits besprochen müssen die übertragenen Daten und Nachrichten minimiert werden. Dies erfordert eine geschickte Abfrage der Daten, eine optimale Verteilung der Daten und optimierte Protokolle.

Bei der Abfrage von Daten ist es immer sinnvoll, eine Aktion dort auszuführen, wo die meisten der für diese Abfrage benötigten Daten zu finden sind. In diesem Fall müssen dann nur noch die restlichen Daten dorthin transportiert werden. Um aber zu erfahren, wo die meisten Daten stehen, müssen Nachrichten ausgetauscht werden. Es ist daher meist sehr schwer, vorab zu entscheiden, auf welchem Rechner die gewünschte Abfrage die höchste Performance aufweist.

Eine optimale Verteilung der Daten ist dann gegeben, wenn die lokalen Zugriffe maximal sind. Hier empfiehlt sich die Selbstverwaltung des Systems mittels Replika. Die Daten werden immer dorthin transportiert, wo sie benötigt werden. Viele Daten werden häufig mehr als einmal hintereinander gelesen oder geschrieben, was eine gewisse Lokalität garantiert. Die auftretenden Probleme der Haltung von Replikaten haben wir bei dem Problem der Datenhaltung bereits angesprochen.

Optimierte Protokolle reduzieren die Nachrichtenströme. Wir haben bereits am Beispiel des Problems der Datenverwaltung gesehen, wie solche Protokolle aussehen können. Betrachten wir noch ein Beispiel: das Setzen von Locks. Wir benötigen Locks, um Concurrency zu ermöglichen. Zum Setzen eines Locks benötigen wir nur zwei Nachrichten, vorausgesetzt wir arbeiten mit Replikaten und dem beim Problem der Datenverwaltung besprochenen Algorithmus. Diese beiden Nachrichten sind das Anfordern des Locks beim „Heimatrechner“ und das Gewähren dieses Locks. Gäbe es diese zentrale Instanz nicht, so wären je Replikat zwei Nachrichten erforderlich: Anfordern und Gewähren des Locks.

Natürlich gibt es in verteilten Datenbanken auch rechnerübergreifende Deadlocks. Betrachten wir dazu Abb. 52. Dort arbeiten zwei Transaktionen *TA1* und *TA2* rechnerübergreifend auf den beiden Rechnern *A* und *B*. Jede der beiden Transaktionen benötigt Lock L_A auf Rechner *A* und Lock L_B auf Rechner *B*. In der Situation in Abb. 52 kommt es dann zum globalen Deadlock. Wir sehen insbesondere, dass eine lokale Instanz diesen Deadlock nicht erkennen kann. Um nicht gegen Regel 2 zu verstoßen, sind zentralisierte globale Lock-

verwalter nicht erwünscht. Wir benötigen recht komplexe Algorithmen, siehe etwa [Date83] und [Kudl92].

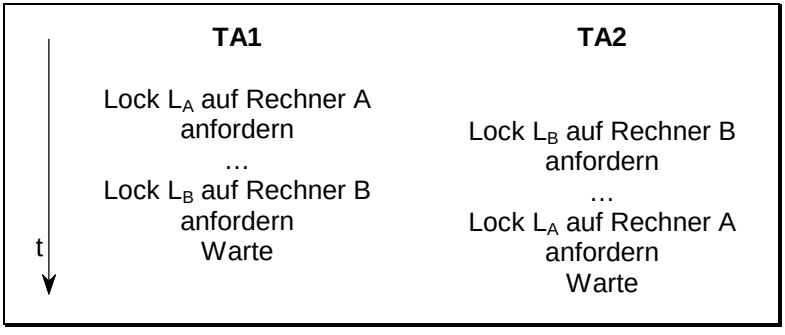


Abb. 52 Globaler Deadlock in verteilten Datenbanken

11.1.4 Zusammenfassung

Wir haben gesehen, dass eine verteilte Datenhaltung den realen Gegebenheiten wesentlich besser entspricht als eine zentrale Verwaltung. Auch die Ausfallsicherheit wird dadurch deutlich verbessert. Allerdings ist der Aufwand zur Verwaltung einer verteilten Datenbank enorm hoch. Als Richtlinie, welche Eigenschaften eine verteilte Datenbank besitzen sollte, dienen die zwölf Regeln von Date. Diese sind zwar nicht zwingend vorgeschrieben, ihre Einhaltung garantiert jedoch die Einhaltung des fundamentalen Prinzips verteilter Datenbanken: Der einzelne Benutzer erkennt nicht, ob eine zentrale oder verteilte Datenverwaltung vorliegt. Unter Benutzer verstehen wir dabei sowohl den Endbenutzer als auch den Anwendungsprogrammierer. Es bleibt also den einzelnen Datenbankverwaltungssystemen vorbehalten, die Verteilung intern durchzuführen, ohne diese extern sichtbar werden zu lassen.

In der Praxis sind wir allerdings noch weit von diesem Ideal entfernt. Wir haben an den wichtigsten Beispielen (Datenverwaltung, Transaktionsbetrieb, Nachrichtenaustausch) gesehen, wie komplex und zeitaufwendig ein ideales verteiltes System ist. Durch das Fallenlassen einzelner Regeln wird der Nachrichtenverkehr im Netz erheblich reduziert. Wir können etwa die Datenmigration unterdrücken, die Replikation einschränken oder einen zentralen Verwaltungsserver einrichten. In diesen Fällen wären die Regeln 1, 2, 4 oder 6 verletzt. Die dadurch entstehenden Nachteile werden meist in Kauf genommen.

Unter einigen dieser Einschränkungen arbeiten verteilte Datenbanken in der Praxis bereits beachtlich gut. Die marktführenden verteilten Datenbanken sind dB2 von IBM, SQL*Star von Oracle und INGRES/Star von Relational Technology. Diese Datenbanken erfüllen zwar nicht alle zwölf Regeln, doch sie sind auf verschiedenen Rechnerplattformen verfügbar, und sie können auch miteinander über Gateways Daten austauschen. Mit immer leistungsfähigeren Netzen kann sich auch die Performance dieser Datenbanken langsam sehen lassen. Zu Details sei auf die Informationen und Handbücher der einzelnen Hersteller verwiesen.

11.2 Objektorientierte Datenbanken

Unter objektorientierten Datenbanken verstehen wir Datenbankmodelle, denen objektorientierte Konzepte zugrunde liegen. Mit den objektorientierten Programmiersprachen entstanden in den 80er Jahren auch diese neuen Datenbanken. Sie sind seit über 10 Jahren im praktischen Einsatz. Wir unterscheiden heute zwei Arten objektorientierter Datenbanken: Zum Einen die Datenbanken, die aus objektorientierten Programmiersprachen entstanden, und zum Anderen diejenigen, die als Erweiterung relationaler Datenbanken in Richtung Objektorientierung anzusehen sind. Wir wollen beide Arten vorstellen, die erfolgversprechendere zweite am Beispiel von Oracle V8.0 etwas ausführlicher.

Es ist in diesem Abschnitt von Vorteil, wenn Grundkenntnisse über objektorientierte Programmierung vorhanden sind, und Begriffe wie Objekte, Klassen, Methoden, Datenkapselung und Vererbung geläufig sind.

11.2.1 Definition objektorientierter Datenbanken

Der Begriff *objektorientiert* ist in der Literatur nicht genau festgelegt. Entsprechend schwer fällt es zu sagen, wann eine Datenbank objektorientiert ist und wann nicht. Wir geben daher eine sehr allgemeine Definition an.

Definition (objektorientierte Datenbanken)

☞Eine Datenbank heißt objektorientiert, wenn sie grundlegende objektorientierte Konzepte wie Objekte, Klassen, Methoden, Kapselung und Vererbung enthält.

Die in der Definition aufgezählten Begriffe sind der objektorientierten Programmierung entnommen. Bei Objekten handelt es sich um einzelne Gegenstände, die gegebenenfalls sehr komplex sein können. Beispielsweise wäre ein bestimmtes Flugzeug ein Objekt, das aus Rumpf, Tragflächen und Motoren bis hin zu einzelnen Anzeigeinstrumenten und Schrauben zusammengesetzt ist. Eine Klasse ist ein Objekttyp, gibt also den Aufbau von Objekten an. Objekte sind folglich Elemente einer bestimmten Klasse. Eine Klasse besteht nicht nur aus Variablen, sondern gibt auch die Methoden an, die auf die Klassenobjekte angewendet werden. Beim Flugzeug kommen unter anderem die Methoden *Flugzeug_starten*, *Flugzeug_warten* oder *Flugzeug_landen* in Frage.

Die Kapselung sorgt dafür, dass nicht wahllos die Daten eines Objekts manipuliert werden dürfen. Vielmehr muss über die Methoden zugegriffen werden. Interessieren etwa alle durchgeführten Wartungen, so ist die entsprechende Methode aufzurufen, etwa *Lies_Wartungen*. Dies garantiert, dass die interne Struktur einer Klasse jederzeit geändert werden darf. Es müssen nur jeweils die Methoden dieser neuen Struktur angepasst werden, nicht aber die Programme, die auf diese Klassen zugreifen.

Die Vererbung ermöglicht die Spezialisierung von Klassen. Haben wir etwa eine allgemeine Klasse *Flugzeug* definiert, so können wir die Grundeigenschaften einschließlich deren Methoden auch in die Klassen *Motorflugzeug* und *Segelflugzeug* übernehmen, sie werden also weitervererbt. Es müssen nur noch die zusätzlichen neuen Eigenschaften und Methoden hinzugefügt werden.

Tab. 55 Nachteile relationaler Datenbanken

Nachteile	Auswirkung
Flache Struktur	Komplexe Objekte werden „flachgeklopft“; ihr Aufbau ist aus dem Design nicht mehr direkt ersichtlich
Keine Rekursion	Der Aufbau komplexer Objekte kann nur schwer nachvollzogen werden (Stücklistenproblem)
Vielzahl von Relationen	Häufige Joins auf zusammengehörige Relationen verlängern die Laufzeit

Diese Eigenschaften der Objektorientierung sind auch in Datenbanken willkommen. Denn relationale Datenbanken besitzen auch einige Schwächen. Diese sind in Tab. 55 kurz zusammengefasst: In relationalen Datenbanken fällt es schwer, die nicht weiter zerlegbaren Einzelteile eines komplexen Bauteils zu ermitteln, seien es die Anzahl der Tretlager des Herren- oder Damenrads aus der Beispieldatenbank *Radl* aus Anhang A oder die Anzahl der Bord-

instrumente beim Flugzeug. In objektorientierten Datenbanken liefern entsprechende Methoden, rekursiv programmiert, zuverlässig die gewünschten Ergebnisse.

Um hohe Normalformen und damit einfache Zugriffe und eine redundanzfreie Datenhaltung zu erhalten, müssen komplexe Strukturen in relationalen Datenbanken „flachgeklopft“ werden. Wegen dieser flachen Struktur müssen komplexe Objekte im Relationenentwurf meist völlig umstrukturiert werden. Die dabei entstehenden Relationen haben dann nur wenig mit den zugrundeliegenden Strukturen gemeinsam, müssen aber bei den einzelnen Zugriffe bekannt sein. In der Regel bauen dann zahlreiche Joins die ursprüngliche Struktur wieder auf, was allerdings zu erheblichen Leistungseinbußen führen kann.

Ganz anders sieht dies bei objektorientierten Datenbanken aus. Auch komplexeste Datenobjekte behalten in der Datenbank ihre Struktur. Entsprechende Zugriffe sind daher schnell, auch werden erforderliche rekursive Zugriffe in objektorientierten Systemen direkt unterstützt.

Betrachten wir beispielsweise die Datenbank *Radl* aus Anhang A und dort die Relationen *Teilestamm* und *Teilestruktur*. Die Relation *Teilestamm* enthält alle Teile und die Relation *Teilestruktur* gibt an, welches komplexe Teil aus welchen Einzelteilen besteht. Ein solches Einzelteil kann aber aus weiteren einfacheren Einzelteilen zusammengesetzt sein und so fort. Es liegt ein rekursives Problem zum Bestimmen aller nicht weiter zusammengesetzten Einzelteile eines Fahrrads vor. Rekursion wird in relationalen Datenbanken aber nicht unterstützt, was eine recht trickreiche Programmierung erfordert.

In objektorientierten Systemen können wir dagegen die Struktur eines Objektes *Fahrrad* direkt angeben (siehe Abb. 53). Die Originalstruktur wird also unverändert übernommen. Wir können uns mit entsprechenden Befehlen direkt entlang dieser Strukturen bewegen und erhalten die gewünschten Informationen direkt und schnell.

Wir wollen aber nicht verschweigen, dass objektorientierte Datenbanken auch Nachteile gegenüber relationalen besitzen. Die komplexere Struktur erfordert eine umfangreichere Verwaltung. Vor allem steigt auch die Komplexität der Zugriffsbefehle. Die direkte Ableitung des Aufbaus einer objektorientierten Datenbank aus dem Entity-Relationship-Modell bereitet gewisse Schwierigkeiten. Der Aufbau von Objekten aus Unterobjekten ist hierarchisch. Wir müssen die gegebene Welt in dieses Schema einordnen. Auch einige Nachteile hierarchischer Datenbanken bekommen wir hier zu spüren: viele Zugriffsbefehle, keine zugrundeliegende umfassende mathematische Theorie.

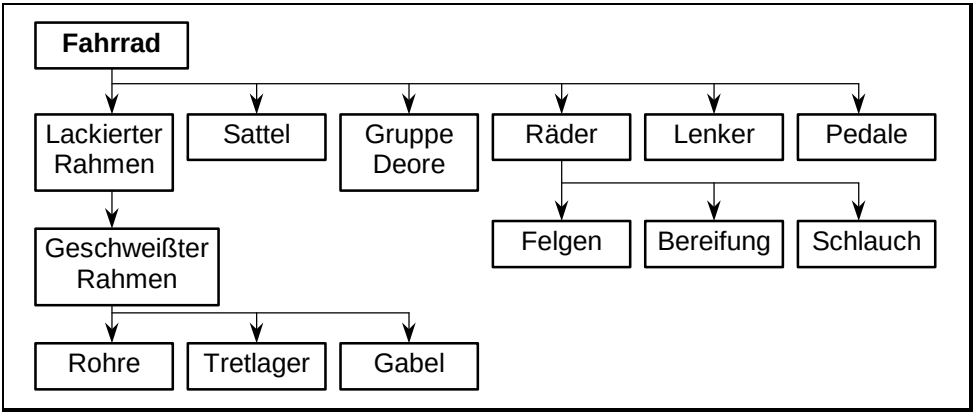


Abb. 53 Objekt Fahrrad

Aus diesen genannten Gründen werden objektorientierte Datenbanken die relationalen wohl nur dort verdrängen können, wo komplexe Objekte zugrundeliegen und gefragt sind. Dies ist vor allem im technisch wissenschaftlichen Bereich der Fall, etwa in den Bereichen CAD (Computer Aided Design), CIM (Computer Integrated Manufacturing), Büroautomatisation, CASE (Computer Aided Software Engineering) und Multimedia.

In praktisch allen anderen Fällen lohnt der Umstieg nicht: zum Einen wegen des hohen Umstellungsaufwands, zum Anderen wegen der oben genannten Nachteile objektorientierter Datenbanken. Gerade wegen des hohen Umstellungsaufwands von relationalen zu objektorientierten Systemen gibt es erfolgreiche Versuche, bestehende relationale Systeme in Richtung Objektorientierung zu erweitern. Der wichtigste Ansatz in dieser Richtung ist die Sprache SQL-3 (siehe [SQL3]), mit deren abschließenden Normierung demnächst gerechnet wird. Wir gehen im nächsten Unterabschnitt kurz darauf ein, um dann im übernächsten einige Neuerungen in Oracle V8.0 ausführlich zu behandeln. In Unterabschnitt 11.2.3 beschäftigen wir uns dann mit Datenbanken, die aus objektorientierten Programmiersprachen durch Hinzufügen von Datenbankeigenschaften entstanden.

11.2.2 Objektrelationale Datenbanken

Die Erweiterung relationaler Datenbanken durch Hinzufügen einiger objektorientierter Strukturen und Methoden ist ein interessanter Ansatz, den wir hier vorstellen wollen. Dieser Ansatz ist darauf ausgerichtet, die in Tab. 55 aufgeführten Nachteile zu beseitigen oder zumindest zu verringern. Es ist dabei

nicht das Ziel, möglichst viele objektorientierte Konstrukte zu übernehmen. Vielmehr bleiben Vererbung oder Datenkapselung teilweise ausgespart. Wir sprechen hier auch von objektrelationalen Datenbanken.

Dieser Ansatz der Erweiterung ist deshalb von großem Interesse, da Migrationen von einem in ein anderes System sehr aufwendig sind. Eine Erweiterung hingegen würde die bisherigen rein relationalen Systeme weiter unterstützen, so dass mit der Zeit objektorientierte Methoden hinzugefügt werden könnten.

Ein großer Vorteil relationaler Datenbanken ist ihre Einfachheit. Diese wird unter anderem dadurch erreicht, dass die einzelnen Attribute nur atomare Einträge enthalten dürfen (erste Normalform). Wir haben aber bereits in Kapitel 3 gesehen, dass dies auch Nachteile mit sich bringt. Die Relation *Verkaeufer-Produkt* aus Tab. 13 auf Seite 65 war in erster Normalform, besitzt aber eine nicht akzeptable hohe Redundanz. Wird die Atomarität nicht mehr gefordert, so kann diese Redundanz leicht beseitigt werden. Wir verweisen auf Tab. 15. Genauso ließe sich auch leicht eine redundanzfreie Relation *Fahrrad* erstellen, in der in einem Attribut *Einzelteile* einfach die Einzelteile aufgelistet werden.

Dieser Ansatz führt zu den NF^2-Relationen (NF^2 = NFNF = Non-First-Normal-Form). Wir definieren:

Definition (NF^2-Relationen)

☞ Eine Relation ist in der n-ten NF^2-Form (n>1), falls sie in der n-ten Normalform ist, wobei aber die Bedingung der ersten Normalform (Atomarität der Attribute) nicht erfüllt sein muss.

Diese Definition ist nicht exakt. Genaugenommen müssten wir erst angeben, was wir unter einer Determinante oder der funktionalen Abhängigkeit bei zusammengesetzten Attributen verstehen. Es soll hier jedoch nur angedeutet werden, dass die wichtigen Bedingungen der höheren Normalformen erhalten bleiben.

Bereits die NF^2-Relationen erlauben die Definition komplexer Objekte. Um auch Rekursion zu ermöglichen, muss die Anwendungssprache zu einer vollständigen Zugriffssprache ausgebaut werden. Weitergehende objektorientierte Methoden (Vererbung, Datenkapselung) erfordern weitere umfangreiche Spracherweiterungen.

Genau in diese Richtung zielt SQL-3. Die vorliegenden Arbeitspapiere (siehe [SQL3]) lassen den Funktionsumfang dieser zukünftigen Sprache bereits erkennen. SQL-3 wird Klassen, Objekte, Methoden, Einkapselung und

Vererbung kennen. Unterstützt werden auch das Überladen von Funktionen und parametrisierte Typen (*Templates*). Mittels des neuen Operators *Recursive Union* wird auch das Stücklistenproblem gelöst.

Dies alles erfordert eine umfangreiche Erweiterung der SQL-2 Norm. Sprechen wir diese hier nur kurz an: Der neue Befehl *Create Type* definiert Datentypen. Diese entsprechen dem aus der objektorientierten Programmierung bekannten Klassenbegriff: Es können sowohl Variablen, Konstanten als auch Memberfunktionen inklusive Konstruktoren und Destruktor definiert werden. Die Datenkapselung geschieht analog C++ mit den Bezeichnern *Public*, *Private* und *Protected*.

Um Funktionen innerhalb eines SQL-Befehls definieren zu können, benötigen wir Kontrollstrukturen. Folglich finden wir in der neuen SQL-3 Norm auch *If-*, *While-* und *For*-Anweisungen. Funktionen außerhalb des *Create-Type*-Befehls dürfen die SQL-eigenen Sprachkonstrukte oder eine Wirtssprache, etwa C++, verwenden. Die Funktionen dürfen überladen werden und können auch rekursiv aufgerufen werden. Der *Create-Type*-Befehl ermöglicht ferner eine Klassenhierarchie, indem die hier definierten Datentypen von anderen Klassen abgeleitet werden können. Mittels des erweiterten *Create-Table*-Befehls können nun Objekte (Relationen) erzeugt werden, die von einem durch den *Create-Type*-Befehl definierten Typ sind.

Mit der endgültigen Verabschiedung der neuen SQL-3 Norm wird eine weltweit eindeutige Norm existieren, die wesentliche objektorientierte Eigenschaften enthält. Die neue Norm ist bewusst so aufgebaut, dass die SQL-2 Norm, im Intermediate Level, voll enthalten ist. Ein behutsamer Umstieg bestehender Datenbanken ist daher möglich. Inwieweit sich diese neue Norm durchsetzen wird, wird die Zukunft zeigen. Seit einiger Zeit steht beispielsweise mit Oracle V8.0 eine weit verbreitete Datenbank mit objektrelationalen Erweiterungen zur Verfügung. Einige dieser Erweiterungen und neue Möglichkeiten werden im nächsten Unterabschnitt vorgestellt.

11.2.3 Objektrelationale Erweiterungen in Oracle V8.0

Der Sprachumfang der Datenbank Oracle wurde in der Version V8.0 erheblich erweitert. Mit dieser Version steht eine leistungsfähige objektrelationale Datenbank zur Verfügung, die natürlich weiterhin alle Möglichkeiten relationaler Datenbanken bietet. Da die neue SQL-3 Norm noch nicht verabschiedet ist, werden in diesem Abschnitt an Hand von Oracle V8.0 einige der neuen

objektorientierten Möglichkeiten, die bereits im letzten Unterabschnitt angesprochen wurden, im Detail vorgestellt.

Es sei nicht verschwiegen, dass Oracle nicht alle neuen Ansätze der neuen Norm in seine Version 8.0 aufgenommen hat. Oracle besitzt bereits seit längerem eine leistungsfähige Programmiersprache, genannt PL/SQL. In diese Sprache werden die neuen Erweiterungen eingebettet, so dass Oracle auf die Definition neuer Schleifen- oder Verzweigungsstrukturen verzichtet hat.

In Oracle V8.0 gibt es mehrere Möglichkeiten, bestehende oder neue Datenbanken mit objektorientierten Strukturen zu versehen. Diese Möglichkeiten sind:

- Objekt: Neben Relationen können jetzt auch Objekte definiert werden. Attribute von Relationen bestehen wie bisher aus Standarddatentypen oder aus anderen bereits definierten Objekten.
- Objekt-Sichten: Mit diesen Sichten können bereits bestehende Relationen mit einer objektorientierten Sicht versehen werden. Dies ist die einfachste Möglichkeit, existierende relationale Datenbanken mit objektorientierten Strukturen auszustatten.
- Variable Felder: Mit diesen Feldern wird die Atomarität der Attribute einer Relation aufgegeben. Ein Attribut, das als variables Feld definiert wird, kann eine begrenzte Anzahl von Werten aufnehmen.
- Eingebettete Relationen (engl.: nested table): Hier liegen Relationen vor, die komplett in einer anderen Relation enthalten sind. Dies ist die konsequente Weiterführung der variablen Felder. Ein Attribut besteht nicht mehr nur aus einem Feld, sondern jetzt sogar aus einer kompletten, beliebig großen Relation.

Wir wollen im Folgenden diese vier Möglichkeiten an einfachen Beispielen kurz vorstellen. Beginnen wir mit der erweiterten Syntax von Oracle. Einen wesentlichen Platz nimmt der *Create-Type*-Befehl ein. Hier gibt es drei Varianten:

```
CREATE [ OR REPLACE ] TYPE Typname AS OBJECT
( Spalte Datentyp [ , ... ] ,
  [ { MEMBER { Prozedurname | Funktionsname } } [ , ... ] ] )

CREATE TYPE Typname AS
  { VARRAY | VARRYING ARRAY } ( Anzahl ) OF Datentyp

CREATE TYPE Typname AS TABLE OF Datentyp
```

Der erste dieser drei Befehle erzeugt ein neues Objekt. Ein eventuell bereits existierendes wird durch die Option *Or Replace* ersetzt. In diesem Fall geben wir, durch Kommata getrennt, die einzelnen Spaltennamen an. Als Datentyp sind alle Standarddatentypen, aber auch bereits definierte Objekte erlaubt. Wir erkennen weiter, dass in einer Objektdefinition auch sogenannte Memberfunktionen und –prozeduren deklariert werden können. Die Definition dieser Memberfunktionen und –prozeduren geschieht mittels des hier nicht weiter behandelten *Create-Type-Body*-Befehls.

Mit dem zweiten *Create-Type*-Befehl werden variable Felder definiert, mit dem dritten eingebettete Relationen. Betrachten wir jetzt einige kleine Beispiele:

In der Beispieldatenbank *Radl* fällt auf, dass in den drei Relationen *Kunde*, *Lieferant* und *Personal* als Adresse die Attribute *Strasse*, *PLZ* und *Ort* vorkommen. Natürlich ist es sinnvoll, dass alle drei Relationen die gleichen Bezeichnungen verwenden. Auch ist es sinnvoll, dass entweder alle diese Relationen im Attribut *Strasse* die Hausnummer mit aufnehmen, oder dass in allen diesen Relationen beide Daten auf zwei Attribute aufgeteilt werden. Garantieren können wir diese Eigenschaft bisher aber nicht. Mit Oracle V8.0 definieren wir jetzt ein Objekt *TAdresse*:

```
CREATE OR REPLACE TYPE TAdresse AS OBJECT
(   Strasse     CHARACTER ( 30 ) ,
    PLZ         CHARACTER ( 5 ) ,
    Ort         CHARACTER ( 20 )     ) ;
```

Gleichzeitig ändern wir jetzt die Definition unserer drei Relationen. Als Beispiel wird die Relation *LieferantNeu* angegeben:

```
CREATE TABLE LieferantNeu
(   Nr          INTEGER              PRIMARY KEY ,
    Name        CHARACTER ( 20 )     NOT NULL ,
    Adresse     TAdresse ,
    Sperre      CHARACTER      ) ;
```

Die beiden anderen Relationen werden entsprechend geändert. Immer wenn wir Adressdaten in eine neue Relation einfügen müssen, greifen wir nun auf unser neues Objekt zu. Die Datenbank bleibt insgesamt homogen.

Der Zugriff auf Relationen, die Objekte enthalten, ist relativ einfach und entspricht dem auch in Programmiersprachen gewohnten Muster: Die Objektelemente werden zusammen mit dem Objektnamen, getrennt durch einen

Punkt, angegeben. Wollen wir beispielsweise den Namen und den Wohnort aller Lieferanten ausgeben, so lautet jetzt der *Select*-Befehl:

```
SELECT Name, L.Adresse.Ort FROM LieferantNeu L;
```

Es ist zu beachten, dass in Oracle das Objekt *Adresse* mittels des Aliasnamen *L* qualifiziert werden muss. Existiert in einer Datenbanken schon eine Relation *Lieferant*, so ist das Löschen dieser Relation und das anschließende Erzeugen der neuen Relation *LieferantNeu* nicht empfehlenswert. In diesem Fall ist es vorzuziehen, der existierenden Relation ein objektrelationales Aussehen mittels einer Sicht zu geben. Wir erzeugen daher folgende Objekt-Sicht aus der Original-Relation *Lieferant*:

```
CREATE VIEW SLieferant ( Nr, Name, Adresse, Sperre )  AS
    SELECT L.Nr, L.Name, TAdresse (L.Strasse, L.PLZ, L.Ort), L.Sperre
     FROM Lieferant L;
```

Auf diese Art und Weise können wir neu definierte Objekte auch auf bestehende Relationen anwenden. Der Aliasname *L* zusammen mit der Qualifikation aller Attributsnamen ist nicht notwendig. Es soll damit nur betont werden, dass sich die Namen in der *Select*-Klausel auf die Attribute der Relation *Lieferant* und nicht auf die Eigenschaften des Objekts *TAdresse* beziehen. Die Ausgabe aller Lieferantennamen und Wohnorte mittels dieser Sicht lautet analog:

```
SELECT Name, L.Adresse.Ort FROM SLieferant L;
```

Diese Erweiterungen sind wünschenswert. Noch tiefer dringen wir in die objektorientierte Programmierung durch die zusätzliche Definition von Memberfunktionen vor. Wir verzichten hier darauf, da wir dazu Kenntnisse in der Oracle-spezifischen Programmiersprache PL/SQL benötigen würden. Stattdessen gehen wir gleich auf die Definition von variablen Feldern und eingebetteten Relationen ein.

Erinnern wir uns dazu an die nicht atomare Relation *VerkaeuferProdukt* in Tab. 15 auf Seite 70. Gehen wir davon aus, dass die Firma nicht mehr als 20 unterschiedliche Produkte vertreibt, so könnte die Definition dieser Objekt-Relation wie folgt aussehen:

```
CREATE TYPE TProdukt AS VARRAY ( 20 ) OF CHARACTER ( 30 ) ;
```

```
CREATE TABLE VerkaeuferProdukt
(   Verk_Nr       CHARACTER ( 4)    PRIMARY KEY ,
    Verk_Name     CHARACTER (20)    NOT NULL ,
    Adresse       TAdresse ,
    Produkt       TProdukt ,
    Umsatz        NUMERIC (10, 2)   ) ;
```

Mit dem ersten Befehl wird ein variables Feld erzeugt, das maximal 20 Elemente vom Datentyp *Character(30)* aufnehmen kann. Mit dem zweiten Befehl erzeugen wir die nicht-atomare Relation *VerkaeuferProdukt*. Wir haben uns nicht ganz an die Vorlage aus Tab. 15 gehalten haben: Da wir inzwischen den Objekttyp *TAdresse* kennen, wurde dieser auch hier gewinnbringend eingesetzt. Das Hinzufügen der ersten Zeile aus Tab. 15 geschieht auch in objektrelationalen Datenbanken mit einem *Insert*-Befehl:

```
INSERT INTO VerkaeuferProdukt VALUES
(   'V1', 'Meier', TAdresse ('Postfach 100111', '80075', 'München'),
    TProdukt ('Waschmaschine', 'Herd', 'Kühlschrank'), 17000     );
```

Mit dem Insert-Befehl werden genau 5 Attribute eingefügt, genauso viele wie im obigen *Create-Table*-Befehl definiert wurden. Zwei dieser Attribute sind jedoch spezielle Attribute: ein Objekt und ein variables Feld. Damit dies im *Insert*-Befehl korrekt erkannt wird, muss zur Eingabe ein *Cast*-Operator unter Angabe des Datentyps verwendet werden. Die Eingabe der Adresse ist leicht nachvollziehbar. Es werden die drei Attribute des Adressenobjekts angegeben. Analog werden die Eingaben für das variable Feld aufgelistet. Oracle erlaubt entsprechend der Definition die Eingabe von maximal 20 Zeichenketten.

Wir können jetzt mit zwei weiteren *Insert*-Befehlen die Relation *VerkaeuferProdukt* gemäß Tab. 15 füllen. Dass diese Daten auch gespeichert wurden, können wir dann leicht durch folgenden Select-Befehl nachprüfen:

```
SELECT * FROM VerkaeuferProdukt;
```

Es werden genau drei Zeilen ausgegeben, wobei das vierte Attribut jeder Zeile eine Aufzählung enthält. Die Daten eines variablen Feldes verwaltet Oracle intern. Oracle führt beispielsweise auch einen Zähler, der die Anzahl der momentan im variablen Feld gespeicherten Elemente enthält. Es ist nicht möglich, direkt mit einem SQL-Befehl gezielt auf Teile von Daten eines variablen Feldes zuzugreifen. Wir benötigen dazu die Oracle-spezifische Spra-

che PL/SQL. Wir gehen daher nicht weiter auf die Möglichkeiten variabler Felder ein und behandeln gleich eingebettete Relationen.

Eingebettete Relationen sind, grob gesprochen, eine Erweiterung von variablen Feldern. Während wir in variablen Feldern nur eine fest vorgegebene Anzahl von Daten eines bestimmten Objekttyps speichern konnten, können wir in eingebetteten Relationen beliebig viele Objekte ablegen. Betrachten wir einige grundlegende Möglichkeiten an einem Beispiel.

In der Beispieldatenbank *Radl* werden alle Aufträge in der Relation *Auftrag* und die Einzelpositionen in der Relation *Auftragsposten* verwaltet. Wir können jetzt die Einzelpositionen, die im eigentlichen Sinne ja keine eigene Entität darstellen, auch in einer eingebetteten Relation abspeichern. Definieren wir dazu zunächst ein entsprechendes Objekt:

```
CREATE OR REPLACE TYPE TEinzelposten AS OBJECT
(   Teilenr         INTEGER,
    Anzahl          INTEGER,
    Gesamtpreis     NUMERIC(10,2)       ) ;
```

Um dieses Objekt in einer Relation einbetten zu können, müssen wir es entsprechend kennzeichnen:

```
CREATE TYPE  ET_Einzelposten  AS TABLE OF  TEinzelposten ;
```

Der Datentyp *ET_Einzelposten* ist jetzt die Basis für eine eingebettete Relation. Wir definieren unsere neue Relation *AuftragNeu*:

```
CREATE TABLE  AuftragNeu
(   AuftrNr         INTEGER    PRIMARY KEY,
    Datum           DATE,
    Einzelposten    ET_Einzelposten,
    Kundnr          INTEGER    REFERENCES Kunde
                          ON DELETE NO ACTION   ON UPDATE CASCADE,
    Persnr          INTEGER    REFERENCES Personal
                          ON DELETE SET NULL     ON UPDATE CASCADE   )
 NESTED TABLE Einzelposten  STORE AS  ET_Einzelposten_TAB ;
```

Mit der letzten Zeile der Definition wird der Datenbank mitgeteilt, wo diese eingebettete Relation (*nested table*) abgelegt werden soll.

Der Zugriff auf diese neue Relation ist ähnlich komplex wie bei variablen Feldern. Beginnen wir wieder mit dem Hinzufügen von Datensätzen. Wir

wollen den zweiten Auftrag aus der Originalrelation in der neuen Relation abspeichern. Der Auftrag enthält zwei Einzelpositionen:

```
INSERT INTO AuftragNeu VALUES
(   2,  '06-Sep-98',
    ET_Einzelposten (    TEinzelposten (100002, 3, 3750),
                         TEinzelposten (200001, 1, 700)   ),     3,  5     ) ;
```

Das Einfügen in Relationen mit eingebetteter Relation und in Relationen mit variablen Feldern ist vergleichbar. Da hier jedes einzufügende Element ein Objekt ist, gibt ein weiterer *Cast*-Operator den korrekten Datentyp an. Die eingebettete Relation *ET_Einzelposten* enthält also eine beliebige Anzahl von Objekten vom Datentyp *TEinzelposten*. Hierin liegt der wesentliche Unterschied zwischen variablen Feldern und eingebetteten Relationen: Erstere speichern eine vorgegebene maximale Anzahl von Objekten, letztere eine beliebige, nur durch die Speicherkapazität beschränkte Anzahl.

Somit unterscheidet sich auch die Speichertechnik. Oracle speichert variable Felder direkt in einer Relation, wobei Speicher für die maximale Anzahl von Elementen reserviert wird. Eingebettete Relationen werden hingegen als eigene Relationen verwaltet. Wir verstehen damit die letzte Zeile der Definition der Relation *AuftragNeu*. Hier wird mitgeteilt, in welcher Relation diese Daten abgelegt werden sollen. Auch wenn die eingebettete Relation *ET_Einzelposten_TAB* von Oracle als eigenständige Relation verwaltet wird, so unterscheidet sich diese doch erheblich von „normalen“ Relationen. Insbesondere ist ein direkter Zugriff auf diese Relation nicht möglich. Originalrelation und eingebettete Relation werden intern mittels Zeigern verbunden, der Zugriff erfolgt ausschließlich über die Relation *AuftragNeu*.

Und damit wären wir auch schon beim Vorteil dieser neuen Möglichkeiten. Wir haben nicht zwei Relationen definiert, die mittels eines Fremdschlüssels miteinander verbunden sind, sondern nur eine, die interne Verknüpfungen enthält. Fügen wir jetzt mittels vier weiterer *Insert*-Befehle noch die Daten analog dem Inhalt der beiden Relationen *Auftrag* und *Auftragsposten* aus Tab. 62 und Tab. 63 hinzu, so erhalten wir einen Gesamtüberblick über den Inhalt durch den einfachen Befehl:

```
SELECT * FROM AuftragNeu ;
```

Der Zugriff ist nicht nur einfacher, sondern auch schneller als bei relationalen Datenbanken, da keine Joins erforderlich sind. Wollen wir jetzt beispiels-

weise alle Einzelpositionen ausgeben, die zu Auftrag 2 gehören, so geben wir folgenden Befehl ein:

```
SELECT  Einzelposten
 FROM    AuftragNeu
 WHERE AuftrNr = 2 ;
```

Bei genauer Betrachtung dieses *Select*-Befehls erkennen wir, dass wir nur ein Tupel und ein Attribut ausgewählt haben. Dieses Attribut ist ein Objekt, das wiederum eine Aufzählung in Form einer eingebetteten Relation enthält. Das Abfrageergebnis ist daher recht unübersichtlich (siehe Tab. 56).

Tab. 56 Direkte Ausgabe einer eingebetteten Relation

Einzelposten(Teilenr, Anzahl, Gesamtpreis)
ET_Einzelposten(TEinzelposten(100002, 3, 3750), TEinzelposten(200001, 1, 700), TEinzelposten(500013, 2, 100))

Es ist daher nicht verwunderlich, dass die Ausgabe nur der Einzelposten, die mehr als 200 DM kosten, nicht direkt möglich ist. Wir müssen dazu die ab Oracle V8.0 zur Verfügung stehende Funktion *The* benutzen. Diese Funktion *The* wandelt eine einzelne eingebettete Relation in eine „gewöhnliche“ Relation um. Dies nutzen wir im folgenden *Select*-Befehl aus, wo das Funktionsergebnis einer *The*-Funktion in der *From*-Klausel verwendet wird:

```
SELECT *
 FROM   THE (  SELECT  Einzelposten
                FROM    AuftragNeu
                WHERE AuftrNr = 2 ) ;
```

Dieser *Select*-Befehl gibt eine Relation aus. Das Abfrageergebnis unterscheidet sich somit erheblich von dem des vorherigen Befehls. Es ist in Tab. 57 wiedergegeben. Die drei Ergebnisattribute heißen *Teilenr*, *Anzahl* und *Gesamtpreis*. Diese Bezeichner können daher in diesem *Select*-Befehl weiter benutzt werden, etwa in einer *Where*-Klausel (siehe auch den nächsten *Select*-Befehl).

Diese *The*-Funktion lässt sich auch im *Insert*-Befehl gewinnbringend einsetzen. Fügen wir zu Auftrag 2 beispielsweise noch 2 Tretlager à 50 DM hinzu, so lautet der dazugehörige Befehl:

```
INSERT INTO THE ( SELECT Einzelposten FROM AuftragNeu
                  WHERE AuftrNr = 2 )
VALUES ( TEinzelposten ( 500013, 2, 100 ) ) ;
```

Auch komplexere Zugriffe auf die Relation *AuftragNeu* sind möglich. Wir müssen dabei immer beachten, dass diese Relation nicht insgesamt eine einzige eingebettete Relation enthält, sondern genaugenommen eine je Zeile. Wollen wir also auf die Einzelpositionen mehrerer Aufträge zugreifen, so müssen wir diese einzelnen eingebetteten Relationen erst zusammenfassen und dann in einen geeigneten Datentyp umwandeln.

Tab. 57 Ausgabe einer eingebetteten Relation als Relation

Teilenr	Anzahl	Gesamtpreis
100002	3	3750
200001	1	700
500013	2	100

Für das Zusammenfassen benötigen wir die *Multiset*-Funktion. Sie ist im Wesentlichen eine Erweiterung der *The*-Funktion. Das Umwandeln in einen passenden Datentyp besorgt schließlich die *Cast*-Funktion. Oracle hält sich mit der Einführung der *Cast*-Funktion in Version 8.0 an die SQL-Norm (siehe auch Abschnitt 6.1). An einem letzten Beispiel wenden wir diese beiden Funktionen an. Wir geben hier alle Einzelpositionen aller Aufträge aus, die einen Gesamtwert von über 1000 DM besitzen:

```
SELECT A1.AuftrNr,
       CAST (MULTISET (SELECT * FROM
                       THE (SELECT Einzelposten FROM AuftragNeu A2
                        WHERE A2.AuftrNr = A1.AuftrNr ) T
                        WHERE T.Gesamtpreis > 1000
             ) AS ET_Einzelposten)
 FROM AuftragNeu A1;
```

Betrachten wir zunächst den inneren *Select*-Befehl. Hier handelt es sich nur um eine leichte Modifikation des letzten Select-Befehls: Statt der fixen Auftragsnummer 2 steht hier die Variable *A1.AuftrNr*. Zusätzlich wurde eine *Where*-Klausel hinzugefügt. Dieser innere *Select*-Befehl liefert also alle Einzelpositionen zu einem vorgegebenen Auftrag, der den Gesamtwert von 1000 DM übersteigt. In einer *Select*-Klausel sind *Select*-Befehle nicht direkt erlaubt. Unser innerer *Select*-Befehl muss daher in einen geeigneten Datentyp umgewandelt werden. Dies geschieht mit der *Cast*-Funktion, die diesen *Select*-Be-

fehl in den Datentyp *ET_Einzelposten* umwandelt. Leider kann die *Cast*-Funktion nicht direkt auf eine Abfrage ausgeführt werden. Diese Abfrage muss erst in eine Relation umgewandelt werden, entweder mittels der *The*- oder der *Multiset*-Funktion. Die *The*-Funktion ist hier nicht anwendbar, da diese nur ein Attribut in der *Select*-Klausel erlaubt. Somit greifen wir zur ähnlich arbeitenden, aber flexibleren *Multiset*-Funktion. Die *Multiset*-Funktion ist nur als Operand des *Cast*-Operators zugelassen, was uns hier aber vollauf genügt.

Der äußere *Select*-Befehl gibt also je Tupel eine Auftragsnummer und eine eingebettete Relation aus, die nur Positionen mit einem Wert über 1000 DM enthält. Dieses Beispiel zeigt deutlich einen Nachteil der objektrelationalen Datenbanken auf: Zusätzliche Befehle erweitern die Syntax erheblich, wodurch auch die DML-Befehle durchwegs komplexer werden. Dies sollte aber nicht abschrecken. In großen Datenbanken mit komplexen Strukturen kann ein objektrelationales Modell vielleicht gerade deshalb erhebliche Vorteile mit sich bringen. Vereint mit dem gezielten Einsatz von Member- und Triggerfunktionen können Datenbank so generiert werden, dass damit die eigentlichen Zugriffe relativ einfach sind. Beispielsweise bietet die Sprache PL/SQL in Oracle auf einfachste Weise an, die Anzahl der Tupel einer eingebetteten Relation zu ermitteln. Wir wollen die Einführung in objektrelationale Datenbanken an dieser Stelle aber beenden. Im nächsten Unterabschnitt werden wir stattdessen noch einen Überblick über „echte“ objektorientierte Datenbanken geben. Als weiterführende Literatur seien [KoLo97], die (Online-)Handbücher von Oracle und die zahlreichen weiteren Bücher über Oracle8 empfohlen.

11.2.4 Weitere objektorientierte Datenbankansätze

Seit mehr als 10 Jahren existieren Prototypen von rein objektorientierten Datenbanken. Bei deren Entwicklung ging man einen von SQL unabhängigen Weg: Objektorientierte Programmiersprachen wurden um Datenbankkonstrukte erweitert. Meist liegen die Sprachen Smalltalk oder C++ zugrunde. Hier sind die objektorientierten Spezifika bereits vorhanden. Diese müssen nun um die Fähigkeiten einer Datenbank erweitert werden. Die wichtigsten sind Persistenz (dauerhafte Datenhaltung), Abfrage, Transaktion, Recovery, Concurrency, Integrität und Sicherheit. Auch die Performance spielt eine wichtige Rolle.

Es gibt hier die verschiedensten Ansätze. Zum Einen können wir wieder versuchen, erweiterte Relationen einzuführen. Wir kommen dann aber wieder

auf den in den letzten Unterabschnitten besprochenen Weg. Weiter können wir vom Entity-Relationship-Modell ausgehen, oder wir bauen eigene, voll auf die objektorientierte Programmierung basierende Datenbanken auf. Wir wollen hier auf keine Einzelheiten eingehen, sondern nur einige der bekannteren und kommerziell verfügbaren Systeme kurz ansprechen:

- **Gemstone**: Dieses System basiert auf der Sprache Smalltalk und gehört zur Gruppe der objektorientierten Datenbank-Programmiersprachen. Gemstone ist eine der ältesten objektorientierten Datenbanken. Sowohl die Datenbeschreibung als auch die Datenmanipulation erfolgen mittels der von Smalltalk abgeleiteten Sprache Opal. Während Smalltalk keine Datentypen kennt, wurden diese in Opal hinzugenommen. Eine ausführliche Beschreibung von Gemstone einschließlich Opal finden wir in [Voss91].
- **Ontos**: Dieses System gehört zur Gruppe der Datenbanken mit objektorientierten Erweiterungen. Die Ontos-Architektur ist auf C++ zugeschnitten. Der Ontos-Präprozessor bearbeitet die über C++ hinausgehenden Konstrukte und erzeugt reinen C++ Code. Einen Überblick zu Ontos liefert [Gopa92].
- **O_2**: Dieses System ist ähnlich zu Ontos, ist aber nicht so sehr auf C++ fixiert. Ontos unterstützt neben C und C++ auch die eigene Datenbanksprache O_2C und eine SQL-ähnliche Abfragesprache namens O_2SQL. Diese Datenbank wurde in Europa entwickelt (Inria, Siemens-Nixdorf, Universität Paris) und ist vor allem hier verbreitet. Der sprachliche Aufbau von O_2 (O_2C und O_2SLQ) ähnelt dem von SQL-3. Einzelheiten zu O_2 finden wir beispielsweise in [Deux91].
- **Postgres**: Diese Datenbank ist eine Erweiterung relationaler Datenbanken. Sie ähnelt noch mehr als O_2 dem SQL-3 Standard. Postgres baut auf der Datenbank Ingres auf und wurde an der Universität von Berkeley entwickelt und hat auch die Datenbank Ingres selbst beeinflusst. Eine gute Beschreibung finden wir in [StKe91].

11.2.5 Zusammenfassung

Wir haben gesehen, dass im wissenschaftlichen Bereich relationale Datenbanken nicht ausreichen. Hier bieten sich objektorientierte Systeme an. Diese Systeme lassen sich durch eine Erweiterung relationaler Datenbanken oder durch eine Erweiterung objektorientierter Programmiersprachen aufbauen.

Die Erweiterung relationaler Datenbanken geschieht unter anderem durch das Fallenlassen der Atomarität von Attributseinträgen. Wir nennen die so erhaltenen Formen NF^2-Relationen. Es sei nicht verschwiegen, dass mit dem Weglassen der Atomarität das Relationenmodell nicht mehr mathematisch fundiert ist. Implementierungen wie Postgres zeigen jedoch die Praxistauglichkeit. Auch der neue SQL-3 Standard wird sich bewähren. Hier wurde die Syntax von SQL-2 durch wichtige objektorientierte Konstrukte ergänzt.

Am Beispiel von Oracle V8.0 wurden einige objektrelationale Neuerungen konkret vorgestellt. Dies waren Datentypdefinitionen, variable Felder und insbesondere eingebettete Relationen. Zum Schluss haben wir noch einige weitere kommerzielle Datenbanken erwähnt, die auf Smalltalk und C++ basieren.

11.3 Übungsaufgaben

1) Was wird unter Unabhängigkeit gegenüber Fragmentierung und gegenüber Replikation verstanden?
2) Definieren Sie den Begriff *Verteilte Datenbanken*!
3) Wir haben globale Deadlocks in verteilten Datenbanken kennengelernt. Können in verteilten Datenbanken auch lokale Deadlocks auftreten?
4) Ist unter Verwendung eines Zwei-Phasen-Commit-Protokolls die Regel 2 erfüllbar, obwohl für dieses Protokoll ein zentraler Koordinator benötigt wird?
5) Kennen Sie eine verteilte Datenbank aus der Praxis? Wenn ja, welche der 12 Regeln verletzt sie? Welche Nachteile werden deshalb in Kauf genommen?
6) Versuchen Sie, den Begriff *Determinante* auf NF^2-Relationen zu erweitern.
7) Wie löst SQL-3 das rekursive Stücklistenproblem?
8) Wie werden in SQL-3 NF^2-Relationen definiert?
9) Kennen Sie objektorientierte Datenbanken aus der Praxis? Wenn ja, versuchen Sie diese einzuordnen (mehr relationale Datenbank mit objektorientierten Erweiterungen oder mehr objektorientierte Programmiersprache mit Datenbankkonstrukten). Beurteilen Sie auch die Zugriffssprache (SQL-ähnlich oder von den Sprachen C++ oder Smalltalk abgeleitet).
10) Schreiben Sie ein Objekt *TPerson*, das den Namen, die Adresse und das Geburtsdatum einer Person enthält. Greifen Sie auf das Objekt *TAdresse* zurück. Schreiben Sie eine Relation *PersonalNeu*, die das Objekt *TPerson* verwendet.
11) Schreiben Sie einen *Select*-Befehl, der aus der Relation *AuftragNeu* alle Einzelpositionen zu Auftrag 4 mit einem Gesamtpreis von über 100 DM ausgibt.

Anhang A Die Beispieldatenbank Radl

Zur Demonstration der Arbeitsweise von relationalen Datenbanken bieten sich einfache, leicht überschaubare Beispiele an. Dadurch kann auf das Wesentliche aufmerksam gemacht werden, ohne durch Details abzulenken. Doch erst an nichttrivialen Anwendungsbeispielen gewinnen wir einen Eindruck von der Leistungsfähigkeit relationaler Datenbanken, sowohl beim Datenbankentwurf als auch bei den Zugriffen auf die Datenbank. Wir stellen deshalb hier eine Datenbank *Radl* vor, die sowohl einfache als auch komplexer aufgebaute Relationen enthält. Sie ist der Praxis entnommen und voll funktionsfähig: Der hier betrachtete Fahrradhändler kann seine Aufgaben mit dieser Datenbank verwalten, auch wenn noch manche Wünsche offen bleiben.

Diese Datenbank *Radl* ist eine Übungsdatenbank. Aus diesem Grund wurden an einigen Stellen absichtlich kleinere Designschwächen eingebaut. Nicht jede Relation befindet sich folglich in der 3. Normalform. Es sei als Übung empfohlen, die noch enthaltenen Designfehler zu suchen und durch Überführung aller Relationen in die 3. Normalform zu beseitigen.

Die erforderliche Software zur Installation dieser Datenbank kann vom Autor über das Internet sowohl für Oracle ab V7.0 als auch für MS-Access97 bezogen werden. Zu Einzelheiten sei auf Anhang D verwiesen.

Im ersten kurzen Abschnitt beschreiben wir die Aufgabe der *Radl*-Datenbank. Die hier aufgezählten Stichpunkte könnten das Ergebnis einer Stoffsammlung zum gewünschten Leistungsumfang der Datenbank sein. Im zweiten Abschnitt wird das Entity-Relationship-Modell vorgestellt, und im dritten Abschnitt werden schließlich die Relationen im Detail behandelt und zur besseren Anschauung bereits mit Arbeitsdaten gefüllt. Im vierten Abschnitt geben wir die *Create-Table*-Befehle zum Erzeugen einiger dieser Basisrelationen an, und im letzten Abschnitt zeigen wir schließlich noch einige komplexere Programme, die auf diese Datenbank zugreifen.

A1 Die Idee der Radl-Datenbank

In diesem Abschnitt stellen wir einen Fahrradhändler vor, der sich überlegt, Warenwirtschaft und Verkauf in seiner Firma mittels einer Datenbank zu ver-

walten. Er verspricht sich davon eine Entlastung von Verwaltungstätigkeiten, insbesondere aber einen besseren Überblick über sein Vorratslager, seine Kunden und Lieferanten und seine Aufträge. Der Händler überlegt sich daher zunächst ausführlich, was seine Datenbank leisten soll. Als Ergebnis notiert er sich Stichpunkte zu seinem augenblicklichen Istzustand und seinen Wunschvorstellungen:

- Es existiert eine beschränkte Anzahl von Teilen, die sich im Laufe der Zeit ändern kann. Unter einem Teil werden sowohl Einzelteile (Schrauben, Fahrradschlauch, ...) als auch Zwischenteile (komplett montiertes Vorderrad, komplette Lichtanlage, ...) bis hin zu Endprodukten (verkaufsfertige Fahrräder) verstanden.
- Zwischenteile und Endprodukte sind aus einfacheren Teilen zusammengesetzt. Diese Struktur soll aus der Datenbank ablesbar sein.
- Nicht immer sind alle Teile vorrätig. Der Händler benötigt eine Lagerverwaltung, aus der ersichtlich ist, welche Teile in welcher Stückzahl vorrätig sind. Darüberhinaus sind die für fest eingeplante Arbeiten benötigten Teile als reserviert zu kennzeichnen.
- Zum Auffüllen des Lagers wendet sich der Händler an seine Lieferanten. Auch diese sind in der Datenbank zu führen. Weiter muss erkennbar sein, welches Teil von welchem Händler geliefert wird (einfachheitshalber nehmen wir an, dass jedes Teil jeweils nur von einem einzigen Händler geliefert wird).
- Der Händler lebt von Aufträgen für seine Kunden. Er benötigt daher auch eine Kunden- und eine Auftragsverwaltung. In der Auftragsverwaltung werden die Aufträge gesammelt, wobei insbesondere bei Reparaturarbeiten vermerkt wird, welche Arbeiten zu einem Auftrag anfallen und welcher Mitarbeiter diese erledigen wird. Auch besteht ein Zusammenhang zum Lager, da benötigte Teile reserviert werden müssen (siehe oben).
- Die Mitarbeiterdaten sind ebenfalls in der Datenbank zu führen, wobei auch vermerkt ist, welcher Mitarbeiter welche Aufträge entgegennahm.
- Um die Datenbank nicht übermäßig aufzublähen, wird zunächst auf ein Rechnungs- und Mahnwesen verzichtet.

Auf Basis dieser kleinen Stoffsammlung ist nun eine Datenbank zu erstellen. Dazu ist entsprechend den Ausführungen zum Datenbankdesign zunächst ein Entity-Relationship-Modell zu entwerfen. Wir empfehlen dringend, dass

sich der Leser selbst intensiv mit einem solchen Entwurf beschäftigt, bevor er mit dem nächsten Abschnitt fortfährt.

A2 Entity-Relationship-Modell der Radl-Datenbank

Auf Basis der Überlegungen des Fahrradhändlers aus dem vorigen Abschnitt erstellen wir ein Entity-Relationship-Modell. Der hier vorgestellte Entwurf ist nicht optimal. Zum Einen sind nicht alle Relationen in der dritten Normalform, und zum Anderen fehlt der Einkauf und ein Rechnungswesen. Es sei dem Leser überlassen, diesen Entwurf zu verbessern und zu erweitern.

Aus den Überlegungen des Händlers aus dem letzten Abschnitt erkennen wir schnell zwei zentrale Stellen in unserer neuen Datenbank. Dies sind die Verwaltung der Teile und das Auftragswesen. Darüberhinaus gibt es Lieferanten-, Kunden- und Personal-Entitäten.

Betrachten wir zunächst die Verwaltung der Teile eingehender: Wir erkennen eine Entität *Teilestamm*. Diese enthält alle Teile, die der Fahrradhändler bei seiner Tätigkeit benötigt, beginnend von Einzelteilen bis zum komplett montierten Fahrrad. Weiter benötigen wir eine Entität *Lager*, aus der der momentane Lagerbestand ersichtlich ist. Den Aufbau komplexer Teile aus Einzelteilen erhalten wir schließlich mittels einer Beziehungsentität, nennen wir sie *Teilestruktur*.

Etwas komplexer ist das Auftragswesen aufgebaut: Zunächst liegt eine Entität *Auftrag* vor, in der alle einkommenden Aufträge verwaltet werden. Sie besitzt Beziehungen zur *Kunden*- und *Personal*-Entität. Einzelheiten zu den einzelnen Aufträgen finden wir in der Entität *Auftragsposten*. Hier sind alle Einzelpositionen zu den einzelnen Aufträgen aufgeführt. Diese Entität besitzt drei Beziehungen zu anderen Entitäten, davon zwei zur Entität *Teilestamm*. Zum Einen wollen wir wissen, welche Teile in Auftrag gegeben wurden. Zum Anderen interessiert, welche Teile für diese Auftragsposition reserviert werden müssen. Im letzten Fall benötigen wir wegen der m:n-Beziehung eine eigene Beziehungsentität, die wir *Teilereservierung* nennen wollen. In ihr ist vermerkt, welche Teile für welchen Auftrag reserviert sind, um sie gegebenenfalls bei einer Stornierung wieder freizugeben. Als drittes existiert eine Beziehung zur Entität *Arbeitsvorgang*, in der die zu einem Auftrag erforderlichen Arbeiten vermerkt werden, etwa Schweißarbeiten am Rahmen eines Fahrrads. Diese Beziehung wird über die Beziehungsentität *AVO_Belegung* hergestellt.

Nochmals möchten wir darauf aufmerksam machen, nach diesen Überlegungen zunächst selbst ein Entity-Relationship-Modell zu erstellen. Vergleichen Sie Ihren Entwurf dann mit demjenigen aus Abb. 54. Alle Verbindungen sind dort als 1:m-, m:1- oder 1:1-Verbindung markiert. m:n-Verbindungen wurden mittels einer Beziehungsentität in m:1-Verbindungen aufgelöst. Die Entität *Lager* ist schwach bezüglich der Entität *Teilestamm*, da nicht oder nicht mehr existierende Teile auch nicht gelagert werden können.

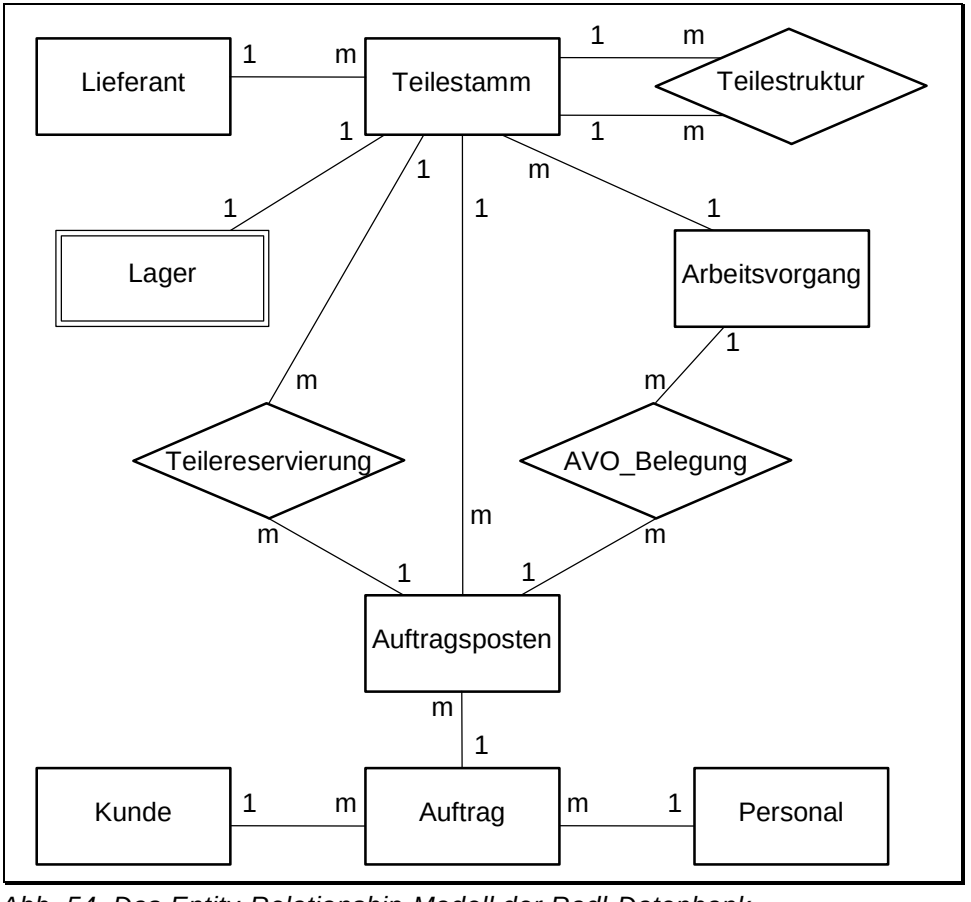


Abb. 54 Das Entity-Relationship-Modell der Radl-Datenbank

Ein besonderes Augenmerk sei auf die Entität *Teilestruktur* gerichtet. Sie ist in der Abbildung rechts oben als Beziehungsrelation eingezeichnet. Solche Beziehungsrelationen, die Entitäten intern über zwei Verbindungen verknüpfen, heißen auch Stücklisten und kommen in der Praxis häufig vor. Diese Stückliste *Teilestruktur* besitzt zwei Fremdschlüssel, die zusamammen den Primärschlüssel bilden. Beide Fremdschlüssel verweisen auf den Primär-

schlüssel der Relation *Teilestamm*. Der erste Fremdschlüssel identifiziert das komplexe zusammengesetzte Teil (*Oberteilnr*), der zweite Fremdschlüssel verweist auf ein Einzelteil (*Einzelteilnr*). Besteht ein zusammengesetztes Teil etwa aus zehn Einzelelementen, so werden dazu in der Stückliste zehn Einträge geführt, die als ersten Fremdschlüsselwert die Nummer dieses zusammengesetzten Teiles enthalten. Der zweite Fremdschlüsselwert verweist dann je Tupel auf die Nummer des jeweiligen Einzelteils. Ein weiteres Attribut gibt die Anzahl der benötigten Einzelteile an.

A3 Die Basisrelationen der Radl-Datenbank

Aus dem Entity-Relationship-Modell aus Abb. 54 erkennen wir, dass insgesamt 11 Entitäten, davon 3 Beziehungsentitäten, vorliegen. Diese 11 Entitäten sind die Basis unserer relationalen Datenbank und werden eins zu eins in Relationen umgesetzt. Die drei Beziehungsentitäten lösen m:n-Beziehungen auf, die in relationalen Datenbanken immer als eigenständige Relationen realisiert werden müssen. Jede der 11 Relationen (Entitäten) besitzt einen Primärschlüssel. Jede der 13 Verbindungslinien repräsentiert einen Fremdschlüssel. Dieser Fremdschlüssel befindet sich immer in der Relation, dem die m-Beziehung zugeordnet ist. Bei der einzigen 1:1 Beziehung zwischen den Relationen *Lager* und *Teilestamm* enthält die schwache Entität *Lager* den Fremdschlüssel. Wir erkennen an dieser Beispiel-Datenbank bereits eindrucksvoll, dass ein korrekt entworfenes Entity-Relationship-Modell (ERM) direkt und ohne Umwege zu den Relationen einer relationalen Datenbank führt. Wir stellen daher diese Relationen im Folgenden einzeln vor. Sie wurden zum leichteren Verständnis mit Beispieldaten gefüllt. Der Eintrag *NULL* in einem Attribut gibt an, dass hier der *Null*-Wert und nicht die Zeichenfolge „NULL“ abgespeichert ist.

- Lieferant: Diese Relation beinhaltet als Attribute (Eigenschaften) den Firmennamen, den Wohnort, die Postleitzahl, die Straße und die Hausnummer. Zusätzlich markiert das Attribut *Sperre*, ob in nächster Zeit noch Teile von diesem Lieferanten bestellt werden sollen, etwa wegen schlechter Erfahrungen in der Vergangenheit. In der Praxis werden häufig noch weitere Daten zu Statistik- oder Informationszwecken aufgenommen.

Tab. 58 Relation Lieferant der Beispieldatenbank

Nr	Name	Strasse	PLZ	Ort	Sperre
1	Firma Gerda Schmidt	Dr.Gesslerstr.59	93051	Regensburg	0
2	Sattler GmbH	Burgallee 23	90403	Nürnberg	0
3	Shimano GmbH	Rosengasse 122	51143	Köln	0
4	Suntour LTD	Meltonstreet 65	NULL	London	0
5	Mannesmann GmbH	St-Rotteneckstr.13	93047	Regensburg	0

- Kunde: Diese Relation entspricht der Relation *Lieferant*. Das *Sperre*-Attribut kennzeichnet hier, ob etwa mangels Liquidität an den Kunden noch etwas verkauft werden darf.

Tab. 59 Relation Kunde der Beispieldatenbank

Nr	Name	Strasse	PLZ	Ort	Sperre
1	Fahrrad Shop	Obere Regenstr. 4	93059	Regensburg	0
2	Zweirad-Center Staller	Kirschenstr. 20	01169	Dresden	0
3	Maier Ingrid	Universitätsstr. 33	93055	Regensburg	1
4	Rafa - Seger KG	Liebigstr. 10	10247	Berlin	0
5	Biker Ecke	Lessingstr. 37	22087	Hamburg	0
6	Fahrräder Hammerl	Schindlerplatz 7	81739	München	0

- Personal: Auch diese Relation ist ähnlich zu den Relationen *Lieferant* und *Kunde* aufgebaut. Neben Name, Vorname, Wohnort, Postleitzahl, Straße und Hausnummer sind aber noch das Geburtsdatum, der Familienstand, der Vorgesetzte, das Gehalt, eine persönliche Beurteilung und die Aufgabe in der Firma (Arbeiter, Vertreter, ...) aufgeführt.

Tab. 60 Relation Personal der Beispieldatenbank (Teil1)

Persnr	Name	Strasse	PLZ	Ort
1	Maria Forster	Ebertstr. 28	93051	Regensburg
2	Anna Kraus	Kramgasse 5	93047	Regensburg
3	Ursula Rank	Donaustr. 12	94315	Straubing
4	Heinz Rolle	In der Au 5	90455	Nürnberg
5	Johanna Köster	Wachtelstr. 7	90427	Nürnberg
6	Marianne Lambert	Fraunhofer Str. 3	92224	Landshut
7	Thomas Noster	Mahlergasse 10	93047	Regensburg
8	Renate Wolters	Schützenstr. 9	93309	Kelheim
9	Ernst Pach	Oberschlagweg 2	94036	Passau

Tab. 61 Relation Personal (Teil2)

Persnr	GebDatum	Stand	Vorgesetzt	Gehalt	Beurteilung	Aufgabe
1	05.07.47	verh	NULL	6800.00	2	Manager
2	09.07.61	led	1	3400.00	3	Vertreter
3	04.09.53	verh	6	4200.00	1	S-Arbeit
4	12.10.43	led	1	4600.00	3	Sekretär
5	07.02.70	gesch	1	3200.00	5	Vertreter
6	22.05.60	verh	NULL	5500.00	1	Meister
7	17.09.58	verh	6	3800.00	5	Arbeiter
8	14.07.65	led	1	4700.00	4	Sachbearb
9	29.03.78	led	6	800.00	NULL	Azubi

In der Praxis werden meist noch weitere Daten aufgenommen, denken wir nur an das Einstelldatum oder die schulische und berufliche Vorbildung. Verbale Beurteilungen oder bisheriger Einsatz in der Firma sind weitere denkbare Einträge.

- Auftrag: Diese Relation wurde gegenüber den Anforderungen in der Praxis erheblich vereinfacht. Es existiert kein Rechnungswesen, so dass zu jedem Auftrag (identifiziert durch das Attribut *AuftrNr*) nur folgende Daten gespeichert werden: das Datum, die Kundennummer und die Vertreternummer (*Persnr*). Einzelheiten zu den einzelnen Positionen des Auftrags finden wir in der folgenden Relation *Auftragsposten*.

Tab. 62 Relation Auftrag der Beispieldatenbank

AuftrNr	Datum	Kundnr	Persnr
1	04.08.98	1	2
2	06.09.98	3	5
3	07.10.98	4	2
4	18.10.98	6	5
5	06.11.98	1	2

- Auftragsposten: Diese Relation enthält alle wichtigen Auftragsdaten. Sie nimmt in der Datenbank eine zentrale Rolle ein, da sie mit den Relationen *Auftrag*, *Teilestamm* und *AVO_Belegung* in Beziehung steht. Mit der Relation *Teilestamm* bestehen sogar zwei Beziehungen, zum Einen direkt und zum Anderen über die Beziehungsrelation *Teilereservierung*. Im ersten Fall wird das in Auftrag gegebene Teil gemerkt, im zweiten Fall die dafür zu reservierenden Einzelteile.

Tab. 63 Relation Auftragsposten der Beispieldatenbank

PosNr	AuftrNr	Teilenr	Anzahl	Gesamtpreis
11	1	200002	2	1.400,00
21	2	100002	3	3.750,00
22	2	200001	1	700,00
31	3	100001	1	1.350,00
32	3	500002	2	220,00
41	4	100001	1	1.350,00
42	4	500001	4	50,00
43	4	500008	1	173,00
51	5	500010	1	75,00
52	5	500013	1	50,00

- Arbeitsvorgang: Diese Relation enthält alle (Reparatur)-Arbeiten, die in der Firma anfallen. Die Relation enthält neben einer Arbeitsvorgangsnummer die Arbeitsbezeichnung, die benötigte Arbeitszeit für den Arbeitsvorgang, den erforderlichen Arbeitsplatz (z.B. Lackiererei) und den Stundensatz.

Tab. 64 Relation Arbeitsvorgang der Beispieldatenbank

AVOnr	Bezeichnung	Arbzeit	Platznr	PlatzBez	Kosten
1	He-Rahmen schweissen	2.0	1	Schweißerei	125.00
2	Da-Rahmen schweissen	2.5	1	Schweißerei	125.00
3	He-Rahmen lackieren	2.0	2	Lackiererei	110.00
4	Da-Rahmen lackieren	2.0	2	Lackiererei	110.00
5	He-Rad montieren	3.5	3	Werkstatt	90.00
6	Da-Rad montieren	3.5	3	Werkstatt	90.00
7	Rädersatz montieren	0.3	3	Werkstatt	90.00

- Teilestamm: In dieser Relation wird jedes Teil durch eine Bezeichnung, einen Verkaufspreis und gegebenenfalls durch eine Maßangabe (Abmessung) und die Maßeinheit (z.B. *ST* für Stück oder *CM* für Zentimeter) beschrieben. Ein weiteres Attribut *Typ* spezifiziert, ob ein Endprodukt (*E*), ein zusammengesetztes Teil (*Z*) oder ein Fremdteil (*F*) vorliegt. Fremdteile sind Einzelteile, die geliefert werden. Zu diesen Teilen wird der Lieferant angegeben und die Lieferzeit vermerkt (einfachheitshalber hatten wir angenommen, dass ein Teil nur von einem Lieferanten geliefert wird). Bei den Teilen vom Typ *E* oder *Z* ist noch der Arbeitsvorgang aufgeführt, in dem diese Teile gefertigt werden.

Tab. 65 Relation Teilestamm der Beispieldatenbank

Teilnr	Bezeichnung	Preis	Mass	Einh.	Typ	Liefernr	Zeit	AVOnr
100001	Herren-City-Rad	1350	25 Zoll	ST	E	NULL	NULL	5
100002	Damen-City-Rad	1250	25 Zoll	ST	E	NULL	NULL	6
200001	He-Rahmen lack.	700	NULL	ST	Z	NULL	NULL	3
200002	Da-Rahmen lack.	700	NULL	ST	Z	NULL	NULL	4
200003	He-Rahmen geschw.	650	NULL	ST	Z	NULL	NULL	1
200004	Da-Rahmen geschw.	650	NULL	ST	Z	NULL	NULL	2
200005	Räder	105	25 Zoll	ST	Z	NULL	NULL	7
500001	Rohr 25CrMo4 9mm	12	9 mm	CM	F	5	1	NULL
500002	Sattel	110	NULL	ST	F	2	4	NULL
500003	Gruppe Deore LX	11	LX	ST	F	3	6	NULL
500004	Gruppe Deore XT	10	XT	ST	F	3	2	NULL
500005	Gruppe XC-LTD	14	Xc-Ltd	ST	F	4	5	NULL
500006	Felgensatz	70	25 Zoll	ST	F	1	1	NULL
500007	Bereifung Schwalbe	35	25 Zoll	ST	F	1	2	NULL
500008	Lenker + Vorbau	173	NULL	ST	F	1	4	NULL
500009	Sattelstütze	10	NULL	ST	F	1	2	NULL
500010	Pedalsatz	75	NULL	ST	F	1	3	NULL
500011	Rohr 34CrMo4 2.1mm	7	2,1 mm	CM	F	5	1	NULL
500012	Rohr 34CrMo3 2.4mm	7	2,4 mm	CM	F	5	1	NULL
500013	Tretlager	50	NULL	ST	F	1	4	NULL
500014	Gabelsatz	20	NULL	ST	F	1	5	NULL
500015	Schlauch	15	25 Zoll	ST	F	1	1	NULL

- Teilestruktur: Auf diese Relation wurde bereits im letzten Abschnitt hingewiesen. Sie enthält zu allen Teilen aus der Relation *Teilestamm*, die aus einfacheren Teilen zusammengesetzt sind, folgende Angaben: Ein Attribut gibt das komplexe Teil (*Oberteilnr*) an, ein weiteres Attribut beschreibt das darin enthaltene einfache Teil (*Einzelteilnr*). Das Attribut *Anzahl* gibt die Anzahl der benötigten Einzelteile an, und das Attribut *Einheit* bezieht sich auf die Maßeinheit des Einzelteils. Besteht ein komplexes Teil aus mehreren verschiedenen einfachen Teilen, so werden entsprechend viele Tupel in dieser Relation eingetragen. Diese Relation *Teilestruktur* entspricht einer einfachen Stückliste, die in der Praxis sehr häufig vorkommt.

Tab. 66 Relation Teilestruktur der Beispieldatenbank

Oberteilnr	Einzelteilnr	Anzahl	Einheit
100001	200001	1	ST
100001	500002	1	ST
100001	500003	1	ST
100001	200005	1	ST
100001	500008	1	ST
100001	500009	1	ST
100001	500010	1	ST
100002	200002	1	ST
100002	500002	1	ST
100002	500004	1	ST
100002	200005	1	ST
100002	500008	1	ST
100002	500009	1	ST
100002	500010	1	ST
200001	200003	1	ST
200002	200004	1	ST
200003	500001	180	CM
200003	500011	161	CM
200003	500012	20	CM
200003	500013	1	ST
200003	500014	1	ST
200004	500001	360	CM
200004	500011	106	CM
200004	500012	20	CM
200004	500013	1	ST
200004	500014	1	ST
200005	500007	2	ST
200005	500006	1	ST
200005	500015	2	ST

- Lager: Diese Relation enthält neben der Teileangabe den Lagerort und den Bestand. Weiter wird ein Mindestbestand, die Anzahl der reservierten und der bereits bestellten Stücke gespeichert. Nicht jedes in der Relation *Teilestamm* angegebene Teil muss in dieser Relation enthalten sein. Teile, die weder auf Lager, noch reserviert oder bestellt sind, brauchen hier nicht aufgeführt werden.

Tab. 67 Relation Lager der Beispieldatenbank

Teilenr	Lagerort	Bestand	Mindbest	Reserviert	Bestellt
100001	001002	3	0	2	0
100002	001001	6	0	3	0
200001	NULL	0	0	0	0
200002	004004	2	0	0	0
200003	NULL	0	0	0	0
200004	002001	7	0	2	0
200005	005001	1	0	0	0
500001	003005	8050	6000	184	0
500002	002002	19	20	2	10
500003	001003	15	10	0	0
500004	004001	18	10	0	0
500005	003002	2	0	0	0
500006	003004	21	20	0	0
500007	002003	62	40	0	0
500008	003003	39	20	1	0
500009	002007	23	20	0	0
500010	001006	27	20	1	0
500011	001007	3250	3000	161	0
500012	004002	720	600	20	0
500013	005002	20	20	2	0
500014	005003	27	20	1	0
500015	002004	55	40	0	0

- AVO_Belegung: Diese Relation speichert, welche Arbeitsvorgänge für welchen Auftragsposten benötigt werden. Diese Beziehungsrelation besitzt demnach je einen Fremschlüssel auf die Relationen *Arbeitsvorgang* und *Auftragsposten*. Das dritte Attribut nimmt die Anzahl der benötigten Arbeitsvorgänge auf.

Tab. 68 Relation AVO_Belegung der Beispieldatenbank

PosNr	AVOnr	Anzahl
11	4	2
22	1	1
22	3	1

- Teilereservierung: Diese Relation gibt Auskunft, welche Teile für welchen Auftragsposten reserviert wurden. Diese Beziehungsrelation besitzt je einen Fremdschlüssel auf die Relationen *Teilestamm* und *Auf-*

tragsposten, die zusammen den Primärschlüssel bilden. Ein weiteres Attribut speichert noch die Anzahl der einzelnen reservierten Teile.

Tab. 69 Relation Teilereservierung der Beispieldatenbank

Posnr	Teilenr	Anzahl
11	200004	2
21	100002	3
22	500001	180
22	500011	161
22	500012	20
22	500013	1
22	500014	1
31	100001	1
32	500002	2
41	100001	1
42	500001	4
43	500008	1
51	500010	1
52	500013	1

A4 Deklaration der Radl-Datenbank

Ausgehend vom obigen Entity-Relationship-Modell aus Abb. 54 kommen wir jetzt zur Realisierung dieser relationalen Datenbank mittels SQL-Befehlen. Wir haben dazu im letzten Abschnitt schon die Eigenschaften der elf erforderlichen Basisrelationen kennengelernt, die Attribute der einzelnen Relationen sind demnach bekannt. Wir müssen uns daher nur noch Gedanken über die Primär- und Fremdschlüssel der einzelnen Relationen machen, um die beiden wichtigen Integritätsregeln (Entität und Referenz) zu erfüllen. Wieder empfehlen wir, die Schlüssel zunächst selbst zu erarbeiten. Als Hinweis zu den Fremdschlüsseln sei angemerkt, dass es genauso viele Fremdschlüssel wie Verbindungen im Entity-Relationship-Modell geben muss. Die *Create-Table*-Befehle zum Erzeugen der Relationen sollten zunächst selbst erstellt werden, bevor sie mit den folgenden Angaben verglichen werden.

Im Folgenden sind nicht alle Basisrelationen aufgeführt, die weiteren meist einfacheren seien als Übung empfohlen (siehe auch die Lösungen zu Übung 1 aus Kapitel 6). Wir haben die SQL-Befehle zum Erzeugen der zentralen Relationen *Teilestamm*, *Auftrag* und *Auftragsposten* angegeben, ebenso diejenigen

zum Erzeugen der Beziehungsrelationen *Teilestruktur* (Stückliste) und *AVO_Belegung* zusammen mit der Relation *Arbeitsvorgang*. Aus Übersichtsgründen haben wir auf die Überprüfung der semantischen Integrität mittels zusätzlicher Tabellen- oder Spaltenbedingungen meist verzichtet. Diese Erweiterungen der *Create-Table*-Befehle seien wieder als Übung empfohlen. Beginnen wir mit der Relation *Teilestamm*:

```
CREATE TABLE  Teilestamm
(   Teilnr          INTEGER             PRIMARY KEY,
    Bezeichnung     CHARACTER (20),
    Preis           Numeric (10, 2)     CHECK ( Preis > 0),
    Mass            CHARACTER (15)      NOT NULL,
    Einheit         CHARACTER (2)       NOT NULL,
    Typ             CHARACTER (1)       CHECK ( Typ IN ( 'E', 'Z', 'F' ) ),
    Liefernr        INTEGER             REFERENCES Lieferant
                            ON DELETE SET NULL   ON UPDATE CASCADE,
    Zeit            CHARACTER (2),
    AVOnr           INTEGER             REFERENCES Arbeitsvorgang
                            ON DELETE SET NULL   ON UPDATE CASCADE
) ;
```

In dieser Relation ist das Attribut *Teilnr* der Primärschlüssel, die Fremdschlüssel verweisen auf die Relationen *Lieferant* und *Arbeitsvorgang*.

```
CREATE TABLE  Auftrag
(   AuftrNr  INTEGER   PRIMARY KEY,
    Datum    DATE,
    Kundnr   INTEGER   REFERENCES Kunde
                       ON DELETE NO ACTION   ON UPDATE CASCADE,
    Persnr   INTEGER   REFERENCES Personal
                       ON DELETE SET NULL    ON UPDATE CASCADE,
);
```

Die Relation *Auftrag* ist einfach aufgebaut. In den beiden Fremdschlüsseln wird allerdings beim Löschen des entsprechenden Kunden- bzw. Personaleintrags unterschiedlich reagiert. Ein Kunde ist so eng mit einem Auftrag verbunden, dass dieser Kunde nicht aus der Kundentabelle entfernt werden darf, so lange einer seiner Aufträge noch existiert. Scheidet hingegen ein Mitarbeiter aus, so wird der entsprechende Eintrag im Auftrag einfach auf NULL gesetzt.

```
CREATE TABLE Auftragsposten
(   PosNr    INTEGER   PRIMARY KEY,
    AuftrNr  INTEGER   REFERENCES Auftrag
                       ON DELETE CASCADE   ON UPDATE CASCADE,
```

```
    Teilenr   INTEGER   REFERENCES Teilestamm
                        ON DELETE NO ACTION   ON UPDATE CASCADE,
    Anzahl    SMALLINT,
    Gesamtpreis   NUMERIC(10,2)   CHECK (Gesamtpreis > 0),
    UNIQUE (AuftrNr, Teilenr)
);
```

Die Relation *Auftragsposten* besitzt zwei Fremdschlüssel, einen zur Relation *Auftrag*, um jede Auftragsposition einem Auftrag eindeutig zuordnen zu können. Weiter wird auf die Relation *Teilestamm* verwiesen, um die zum Auftrag gehörenden Teile zu identifizieren. Auch hier sind die Merkmale der Fremdschlüssel unterschiedlich: Wird ein Auftrag gelöscht, so sind auch die Einzelpositionen nicht mehr notwendig. Teile dürfen hingegen nicht entfernt werden, solange noch dazugehörige Aufträge existieren. Zu beachten ist ferner ein alternativer Schlüssel, der mit der Tabellenbedingung *Unique* gekennzeichnet wurde.

```
CREATE TABLE  Teilestruktur
(   Oberteilnr   INTEGER   REFERENCES Teilestamm
                           ON DELETE CASCADE     ON UPDATE CASCADE,
    Einzelteilnr INTEGER   REFERENCES Teilestamm
                           ON DELETE NO ACTION   ON UPDATE CASCADE,
    Anzahl       INTEGER,
    Einheit      CHARACTER (2),
    PRIMARY KEY (Oberteilnr, Einzelteilnr)
) ;
```

In der Relation *Teilestruktur* liegt der typische Fall einer Beziehungsrelation vor: Es existieren zwei Fremdschlüssel, die in unserem Fall (bei Stücklisten!) auf die gleiche Relation verweisen. Weiter besteht der Primärschlüssel aus eben diesen beiden Fremdschlüsselattributen. Beachten Sie deshalb, dass hier beim Löschen oder Ändern von Fremdschlüsselwerten die Option *Set Null* nicht erlaubt ist!

```
CREATE TABLE  Arbeitsvorgang
(   AVOnr          INTEGER              PRIMARY KEY,
    Bezeichnung    Character (25),
    Arbzeit        CHARACTER (3)        NOT NULL,
    Platznr        CHARACTER (2)        NOT NULL,
    PlatzBez       CHARACTER (15),
    Kosten         NUMERIC (7,2)
) ;
```

Die Relation *Arbeitsvorgang* besitzt eine Nummer (*AVOnr*), die jedes Tupel eindeutig identifiziert. Ansonsten ist sie sehr einfach aufgebaut.

```
CREATE TABLE  AVO_Belegung
(   PosNr       INTEGER   REFERENCES Auftragsposten
                          ON DELETE CASCADE     ON UPDATE CASCADE,
    AVOnr       INTEGER   REFERENCES Arbeitsvorgang
                          ON DELETE NO ACTION   ON UPDATE CASCADE,
    Anzahl      SMALLINT,
    PRIMARY KEY (PosNr, AVOnr),
) ;
```

Diese Beziehungsrelation *AVO_Belegung* ist trotz seines komplexen Aussehens einfach. Sie stellt eine Beziehung zwischen den Relationen *Auftragsposten* und *Arbeitsvorgang* her. Es gibt daher Fremdschlüssel zu beiden Beziehungen. Die beiden Fremdschlüssel ergeben zusammen den Primärschlüssel der Relation. Das noch verbleibende Attribut *Anzahl* gibt die Anzahl der erforderlichen Arbeitsschritte wieder.

Zuletzt sei noch ein Hinweis für Benutzer von Datenbanken angegeben, die nicht über den vollständigen SQL-2-Standard verfügen. In diesem Fall sind die *On-Delete*- und *On-Update*-Optionen bei den Fremdschlüsseln wegzulassen, so auch in Oracle (nur *On Delete Cascade* wird unterstützt) und MS-Access (*On Delete Cascade* wird unterstützt, jedoch nicht mittels SQL-Befehlen).

Zu beachten ist beim Erzeugen der Datenbank die Reihenfolge der DDL-Befehle, da bei Bezügen auf andere Relationen (Referenzangaben) diese anderen Relationen bereits existieren müssen. In ganz wenigen seltenen Fällen kann es in der Praxis vorkommen, dass Relationen direkt oder indirekt aufeinander gegenseitig verweisen. Dann müssen in den *Create-Table*-Befehlen einzelne Referenzen zunächst weggelassen und mit Hilfe des *Alter-Table*-Befehls erst später hinzugefügt werden. In unserem Beispiel sind zunächst die Relationen *Lieferant, Kunde, Personal* und *Arbeitsvorgang* zu erzeugen; denn diese Relationen haben gemeinsam, dass sie keine Fremdschlüssel besitzen.

Die Beispieldatenbank wird im Internet zur Verfügung gestellt. Beachten Sie hierzu Anhang D.

A5 Zugriffe auf die Radl-Datenbank

In den letzten Abschnitten haben wir die *Radl*-Datenbank spezifiziert und mittels des DDL-Befehls *Create Table* erstellt. Wir wollen nun mittels einiger ausgewählter Beispiele auf diese Datenbank zugreifen. Wir beginnen zunächst mit einem einfachen Beispiel und kommen zuletzt zu einer komplexeren Anwendung, wo wir auch auf Cursor und eingebettetes SQL zurückgreifen. Um auch den mit MS-Access vertrauten Leser von der Leistungsfähigkeit von SQL zu überzeugen, sind zwei Beispiele auch in der Sprache Visual Basic realisiert.

Beginnen wir mit dem einfachen Beispielprogramm, das eine Liste aller Teile ausgibt, deren Lagerbestand unter den Minimalbestand gesunken ist. Berücksichtigt werden dabei auch bereits reservierte und bereits bestellte Teile. Realisiert ist dieses kleine Programm in SQL*Plus, einer SQL-Umgebung von Oracle:

```
REM Ausgabetabelle formatieren und Titel festlegen:

column  teilnr          format  999999    heading  Teilenr
column  bezeichnung     format  A15       heading  TeilBezeichnung
column  zeit            format  A3        heading  LZ
column  name            format  A25       heading  Lieferant
column  bestand         format  99999     heading  Haben
column  mindbest        format  99999     heading  Min
column  reserviert      format  99999     heading  Res
column  bestellt        format  99999     heading  Best

ttitle center 'Lieferanten-Bestellungen' skip 2

REM Select-Befehl:

SELECT Teilnr, Bezeichnung, Zeit, Name, Bestand, Mindbest, Reserviert,
       Bestellt
 FROM  Teilestamm, Lieferant, Lager
 WHERE Bestand - Reserviert < Mindbest + Bestellt
   AND Teilestamm.Teilnr = Lager.Teilenr
   AND Lieferant.Nr = Teilestamm.Liefernr ;

REM ttitle off, da in Folgeseite obiger Titel nicht mehr gewuenscht wird
REM analoges gilt für die Spaltenangaben:
ttitle off
clear columns
```

Neben dem *Select*-Befehl finden wir noch einige weitere *SQL*Plus*-Befehle. Mittels des *Rem*-Befehls wird eine Kommentarzeile eingeleitet, *Column* setzt

Spaltenformate (*A15* steht für 15 alphanumerische Zeichen, *99999* für 5 Ziffern), *Ttitle* bestimmt einen Titel für eine Ausgabe und *Clear* löscht bestimmte Einstellungen. Der *Select*-Befehl selbst enthält einen Join auf die Relationen *Teilestamm*, *Lieferant* und *Lager*. Dieser Join ist erforderlich, um auch Angaben wie Lieferzeit, Lieferant oder Teilebezeichnung ausgeben zu können.

Typisch ist an diesem kleinen Programm der Aufwand für die Ausgabe. In der Praxis sind Ein- und Ausgaben benutzergerecht zu erstellen, was eines erheblichen Programmieraufwands bedarf, auch wenn Werkzeuge wie Ein- und Ausgabemasken und Trigger zur Verfügung stehen. Demgegenüber nehmen sich die Datenbankzugriffe, zumindest vom Programmumfang her, relativ bescheiden aus.

Betrachten wir jetzt ein auf den ersten Blick nur geringfügig komplexeres Programm, dieses Mal realisiert in MS-Access. Das Programm fügt einen neuen Artikel in die Relation *Teilestamm* ein. Je nachdem, ob es sich um ein Fremdteil handelt, oder dieses selbst hergestellt wird, werden auch Angaben zum Lieferanten oder zum erforderlichen Arbeitsvorgang und zum Aufbau dieses Teils angefordert. Allein die Ausgaben zusammen mit den notwendigen Eingaben blähen das Programm erheblich auf:

```
Option Compare Database          ' Zeichenketten normal vergleichen
Option Explicit                  ' Variablen muessen definiert werden

Function ArtikelEinfuegen()
Dim db As DATABASE               ' Datenbankvariable
Dim Transakt As Workspace        ' Arbeitsbereich fuer Transaktionen
Dim SelRec As Recordset          ' Select-Befehls-Bereich
Dim Ins As String                ' Zeichenkette fuer Insert-Befehl

Dim ATeileNr As String           ' Teilenummer
Dim ATeileBez As String          ' Teilebezeichnung
Dim ATeileDM As String           ' Preis
Dim ATeileMass As String         ' Teilemass
Dim ATeileEinh As String         ' Teileeinheit
Dim ATeileTyp As String          ' Teiletyp
Dim ALiefernr As String          ' Lieferantennummer
Dim ALiefZeit As String          ' Lieferzeit
Dim AAVOnr As String             ' AVO-Nummer
Dim AEinzel As String            ' Einzelteilnummer
Dim AAnzahl As String            ' Anzahl der Einzelteile
Dim weiter As Boolean            ' Hilfsschleifenvariable

On Error GoTo FehlerBehandlg     ' Bei Fehlern verzweigen

Set db = CurrentDb               ' aktuelle Datenbank oeffnen
Set Transakt = DBEngine.Workspaces(0)      'Transaktionsbetrieb einrichten
```

```
Transakt.BeginTrans                    ' Transaktion starten

ATeileNr = InputBox("Bitte neue Teilenummer eingeben:", "Teil einlesen")

Set SelRec = db.OpenRecordset(" Select Count(*) As Anz From Teilestamm " _
                              &  " Where Teilnr = " & ATeileNr & ";")

' Abbruch, falls Teil bereits vorhanden ist:

If SelRec!Anz <> 0 Then
    Transakt.Rollback
    msgBox "Teil existiert bereits", , "Abbruch"
    Exit Function
End If

' Teil existiert noch nicht, restliche Daten werden angefordert:

ATeileBez = InputBox("Bitte Bezeichnung eingeben:", "Teiledaten einlesen")
ATeileDM = InputBox("Bitte Preis eingeben:", "Teiledaten einlesen")
ATeileMass = InputBox("Bitte Mass eingeben:", "Teiledaten einlesen")
ATeileEinh = InputBox("Bitte Einheit eingeben:", "Teiledaten einlesen")
ATeileTyp = InputBox("Bitte Teiletyp (F/Z/E) eingeben:", "Teiledaten einlesen")

ATeileTyp = UCase(ATeileTyp)    ' in Grossbuchstaben umwandeln

' Abbruch, falls inkorrektes Teil:

If ATeileTyp <> "F" And ATeileTyp <> "Z" And ATeileTyp <> "E" Then
    Transakt.Rollback
    msgBox "Falscher Teiletyp eingegeben", , "Abbruch"
    Exit Function
End If

If ATeileTyp = "F" Then             ' Teil wird von Lieferanten geliefert

    ' Weitere Daten werden eingelesen:

    ALiefernr = InputBox("Bitte Lieferantennr eingeben:", "Teiledaten einlesen")
    ALiefZeit = InputBox("Bitte Lieferzeit eingeben:", "Teiledaten einlesen")
    ALiefZeit = "'" & ALiefZeit & "'"          ' Als SQL-String schreiben!
    AAVOnr = "NULL"

Else                                ' Teil wird selbst hergestellt

    ' Weitere Daten werden eingelesen:
    ALiefernr = "NULL"
    ALiefZeit = "NULL"
    AAVOnr = InputBox("Bitte AVO-Nummer eingeben:", "Teiledaten einlesen")

    ' Abbruch, falls Arbeitsvorgang nicht existiert:

    Set SelRec = db.OpenRecordset(" Select Count(*) As Anzahl " & _
                                  " From Arbeitsvorgang "  & _
                                  " Where AVONr = " & AAVOnr & ";")

    If SelRec!Anzahl = 0 Then       ' keine AVONr gefunden
        Transakt.Rollback
        msgBox "Arbeitsvorgangsnummer existiert nicht", , "Abbruch"
```

```
        Exit Function
    End If

End If

' Daten werden in Relation Teilestamm abgespeichert:

Ins = "Insert Into Teilestamm " & _
      "Values (" & ATeileNr & ",'" & ATeileBez & "'," & ATeileDM & ",'" & _
                 ATeileMass & "','" & ATeileEinh & "','" & ATeileTyp & "'," & _
                 ALiefernr & "," & ALiefZeit & "," & AAVOnr & ");"
db.Execute Ins

' Eingeben und Speichern des Aufbaus des Teils fuer die Rel. Teilestruktur:

If ATeileTyp <> "F" Then            ' Teilestruktur eingeben, falls kein Fremdteil:
    weiter = True
    While weiter
        AEinzel = InputBox("Bitte Einzelteilnr eingeben:", "Teilestruktur einlesen")
        AAnzahl = InputBox("Anzahl der Einzelteile:", "Teilestruktur einlesen")
        Set SelRec = db.OpenRecordset(" Select Einheit From Teilestamm " & _
                                      " Where Teilnr = " & AEinzel & ";")

        If not SelectRecord.EOF Then        ' Einzeilteil existiert
            Ins = "Insert Into Teilestruktur " & _
                  "Values (" & ATeileNr & "," & AEinzel & "," & AAnzahl & ",'" & _
                               SelRec!Einheit & "');"
            db.Execute Ins
        End If

        If msgBox("Weitere Daten?", vbYesNo, "Teilestruktur") = vbNo Then
            weiter = False
        End If
    Wend
End If

Transakt.CommitTrans        ' Transaktion erfolgreich beendet
msgBox "Daten in Datenbank komplett abgelegt", , "Erfolgreicher Abschluss"

Exit Function


FehlerBehandlg:
    Transakt.Rollback
    msgBox "Fehler in Funktion ArtikelEinfuegen aufgetreten", vbInformation

End Function
```

Dieses Programm ist allein wegen der vielen Abfragen so umfangreich. Es enthält keine besonderen technischen Schwierigkeiten. Einige Hinweise zur MS-Access-Programmierung wurden bereits in Abschnitt 9.5 ab Seite 237 gegeben. Ergänzend sei hier nochmals auf den Transaktionsmechanismus hingewiesen. MS-Access benötigt hierfür einen Arbeitsbereich (Variable *Trans-*

akt) und den expliziten Beginn der Transaktion (*Transakt.BeginTrans*). In diesem Programm wird zum Einlesen eine Inputbox und zur Ausgabe eine Messagebox verwendet. In der Praxis sind natürlich Eingabemasken vorzuziehen. Zur Demonstration der Arbeitsweise von MS-Access wird diese „schlechte“ Lösung jedoch vorgezogen.

Das Programm überprüft zunächst, ob die eingelesene Teilenummer existiert. Wenn ja, so wird das Programm beendet. Anschließend werden alle weiteren Daten zu dem neuen Teil eingelesen. Dabei wird bei Fremdteilen (Teiletyp *F*) die Liefernummer und -zeit angefordert und ansonsten die Arbeitsvorgangsnummer. Wieder wird das Programm vorzeitig beendet, wenn die Vorgangsnummer nicht existiert oder ein falscher Teiletyp angegeben wurde. Danach erfolgt das Abspeichern der Daten in der Relation *Teilestamm*. Lag kein Fremdteil vor, so muss in der Relation *Teilestruktur* noch der Aufbau des Teils eingetragen werden. Diese Angaben werden in einer Schleife angefordert und sofort in die Datenbank eingefügt. Zuletzt wird die Dateneingabe mit einem Commit abgeschlossen. Im Fehlerfall wird zur Marke *FehlerBehandlg* gesprungen und die Transaktion mittels eines Rollback zurückgesetzt.

Dieses Beispiel zeigt unter anderem auch eindrucksvoll die Wirkung des Transaktionsmechanismus. Wir schrieben Änderungen sofort in die Datenbank. Doch erst mit dem *Commit*-Befehl werden diese Änderungen endgültig. Wir konnten im Fehlerfall jederzeit mittels des *Rollback*-Befehl alle bis dahin vorgenommenen Datenbankeinträge wieder rückgängig machen.

Zum Schluss stellen wir noch ein umfangreiches Beispiel vor, in dem sowohl mit Cursorn als auch Transaktionen gearbeitet wird. In diesem Beispielprogramm wird ein bestehender Auftrag gelöscht. Dies erfordert einige Löschaktionen in mehreren Relationen. Wir verwenden wieder ein in C++ eingebettetes SQL-Programm:

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>

/* Definition der in SQL verwendeten Variablen */
EXEC SQL BEGIN DECLARE SECTION;
   char  username[20], ateilebez[21],  aname[31], aort[21];
   long  ateilenr,         aanzahl,        aposnr;
   int   aaufnr,           akdnr;
   char SQLSTATE [6];          // 5 Zeichen + \0 !
EXEC SQL END DECLARE SECTION;
```

```
main ( )          /* Programmbeginn */
{   char antwort;
    int i;

        // Einloggen in die Datenbank:
    cout << "Programm zum Löschen eines Auftrags.\n\n";
    cout << "Datenbanknamen eingeben: ";     cin >> username;
    EXEC SQL CONNECT TO :username;
    if (strcmp(SQLSTATE, "00000"))                          // Abbruch:
    {   cout << " Fehler beim Einloggen in die Datenbank ";  return 0;
    }
    cout << "Einloggen in die Datenbank ist erfolgt.\n";

    cout << "\nEinen Auftrag löschen (j/n) ?";   cin >> antwort;
    if ( toupper (antwort) != 'J' )
        return 0;                           // Loeschen abbrechen

    cout << "\nAuftragsnummer : ";    cin >> aaufnr;

    EXEC SQL  SELECT Kundnr  INTO :akdnr
                  FROM Auftrag
                  WHERE Auftrnr = :aaufnr;
        // SELECT ist nötig, um festzustellen, ob der Auftrag existiert

    if (strcmp(SQLSTATE, "00000"))
    {   cout << "Auftragsnummer existiert nicht oder Fehler, Abbruch\n"; return 0;
    }
        // Auftrag existiert, alle Daten zu diesem Auftrag suchen und loeschen

    EXEC SQL WHENEVER SQLERROR GOTO fehler;

        // Zeiger fuer die verschiedenen Positionen des Auftrags:
    EXEC SQL DECLARE zeiger1 CURSOR FOR
                    SELECT Posnr, Teilenr  FROM Auftragsposten
                    WHERE AuftrNr = :aaufnr
                    FOR UPDATE OF Posnr, Teilenr;

    EXEC SQL OPEN zeiger1;
    EXEC SQL FETCH zeiger1 INTO :aposnr, :ateilenr;

    while (strcmp(SQLSTATE, "02000"))     // noch nicht am Ende der Relation
    {   EXEC SQL SELECT Bezeichnung    INTO :ateilebez
                                       FROM Teilestamm
                                       WHERE Teilnr = :ateilenr;
        cout << "Position " << aposnr << ": " << ateilebez << endl;

        // Eintraege in AVO-Belegung loeschen, die sich auf den Auftrag beziehen
        EXEC SQL DELETE FROM AVO_Belegung
                        WHERE PosNr = :aposnr;

        // Zeiger fuer die verschiedenen Eintraege in Relation Teilereserv.:
        EXEC SQL DECLARE zeiger2 CURSOR FOR
                    SELECT Teilenr, Anzahl  FROM Teilereservierung
                    WHERE Posnr = :aposnr;
        EXEC SQL OPEN zeiger2;
```

```
        EXEC SQL FETCH zeiger2 INTO :ateilenr, :aanzahl;
        while (strcmp(SQLSTATE, "02000"))
        {   // Die Reservierungsdaten in Relation Lager anpassen:
            EXEC SQL UPDATE Lager
                        SET Reserviert = Reserviert - :aanzahl
                        WHERE Teilenr = :ateilenr;

            // Teilereservierungen loeschen, die sich auf Auftragsposten beziehen:
            EXEC SQL DELETE FROM Teilereservierung
                        WHERE CURRENT OF zeiger2;

            EXEC SQL FETCH zeiger2 INTO :ateilenr, :aanzahl;
        }  // end while (Loeschen der Reservierungen)

        EXEC SQL CLOSE zeiger2;

        // Aktuellen Auftragsposten loeschen:
        EXEC SQL DELETE FROM Auftragsposten
                    WHERE CURRENT OF zeiger1;

        EXEC SQL FETCH zeiger1 INTO :aposnr, :ateilenr;
    }   // end while ( Durchsuchen aller Auftragspositionen)

    EXEC SQL CLOSE zeiger1;

    cout << "\n\nDiesen Auftrag wirklich löschen (j/n)? ";    cin >> antwort;
    if ( toupper (antwort) == 'J' )
    {   // nur noch der Auftrag selbst muss geloescht werden:
        EXEC SQL DELETE FROM Auftrag
                    WHERE AuftrNr = :aaufnr;

        EXEC SQL COMMIT WORK;       // Aenderungen speichern
        cout << "Auftrag wurde gelöscht.\n\n";
    }           /* end if */
    else        // Alle Aenderungen sind zurueckzunehmen:
    {   EXEC SQL ROLLBACK WORK;
        cout << "Auftrag wurde nicht gelöscht.\n\n";
    }
    return 0;

fehler:         // Mindestens eine der obigen Aktionen war fehlerhaft

    cout << "\nFehler, daher wird Aenderungsvorgang zurueckgesetzt!\n";
    EXEC SQL ROLLBACK WORK ;
    return 0;
}
```

Dieses C++-Programm beinhaltet viele SQL-Befehle. Es beginnt mit einem *Declare*-Teil, in dem alle in SQL verwendeten Variablen aufgeführt sind. Im Hauptprogramm wird zunächst die Datenbank geöffnet. Bei Misserfolg wird das Programm abgebrochen. Andernfalls wird gefragt, ob ein Auftrag gelöscht werden soll. Wenn ja, so wird die Auftragsnummer abgefragt und überprüft,

ob diese existiert. Wenn nicht, so wird das Programm beendet. Ansonsten werden die Daten zu diesem Auftrag mit Hilfe eines Cursors aufgelistet. In dieser Cursorschleife, in der die Relation *Auftragsposten* durchsucht wird, werden gleichzeitig alle dazugehörigen Reservierungen in der Relation *Lager* zurückgenommen und die entsprechenden Einträge in den Relationen *Auftragsposten*, *Teilereservierung* und *AVO_Belegung* entfernt. Zum Durchsuchen der Angaben in der Relation *Teilereservierung* wird ein zweiter Cursor benötigt. Diese Angaben werden für die Änderungen in der Relation *Lager* benötigt.

Sollte eine dieser Aktionen misslingen, so können Inkonsistenzen auftreten. Aus diesem Grund werden dann mittels eines *Rollback Work* alle Änderungen zurückgesetzt. Dazu wird mit Hilfe der *Whenever*-Anweisung im Fehlerfall zur Marke *fehler* gesprungen. Bei erfolgreicher Ausführung aller Befehle wird ein *Commit Work* durchgeführt. Sicherheitshalber wird am Ende des Programms aber abgefragt, ob das Löschen des Auftrags tatsächlich durchgeführt werden soll. Gegebenenfalls werden mittels eines *Rollback Work* alle Änderungen wieder zurückgenommen.

Dieses Programm wollen wir zum Vergleich auch in MS-Access einbetten. Wir erkennen, dass hier die Cursor-Programmierung durch einen *Recordset* zusammen mit der Eigenschaft *EOF* und der Methode *MoveNext* durchgeführt wird. Die Cursordefinition entspricht dabei der Funktion *OpenRecordSet*. Ebenso steht zum Löschen des aktuellen Datensatzes die Methode *Delete* zur Verfügung. Wieder ist zu beachten, dass eine übersichtliche Bildschirmmaske mit Triggerprogrammierung (kleine Funktionsaufrufe bei Dateneingabe) die Aufgabe besser lösen würde. Doch hier soll gezeigt werden, dass sich die Inhalte gleichen, trotz großer syntaktischer Unterschiede in C++ mit SQL-Befehlen und Visual Basic:

```
Option Compare Database          ' Zeichenketten normal vergleichen
Option Explicit                  ' Variablen muessen definiert werden

Function AuftragLoeschen()
' Variablendeklaration
Dim db As DATABASE               ' Datenbankvariable
Dim Transakt As Workspace        ' Arbeitsbereich fuer Transaktionen
Dim sqlstr As String             ' Hilfsstring fuer SQL-Befehle
Dim Sel As Recordset             ' Select Bereich fuer Abfragen
Dim SelAuf As Recordset          ' Select Bereich fuer Auftragsposten
Dim SelRes As Recordset          ' Select Bereich fuer Teilereservierung
Dim AAufNr As String             ' Auftragsnummer

On Error GoTo FehlerBehandlg     ' Bei Fehlern verzweigen
```

```
Set db = CurrentDb                      ' aktuelle Datenbank oeffnen
Set Transakt = DBEngine.Workspaces(0)  'Transaktionsbetrieb einrichten

Transakt.BeginTrans                     ' Transaktion starten

If msgBox("Auftrag löschen?", vbYesNo, "Auftrag löschen") = vbNo Then
    msgBox "Ende des Programms"
    Transakt.Rollback
    Exit Function
End If

AAufNr = InputBox("Den zu löschenden Auftrag eingeben", "Auftrag löschen")

Set Sel = db.OpenRecordset(" Select Kundnr From Auftrag" & _
                           " Where Auftrnr = " & AAufNr & ";")

' SELECT ist nötig, um festzustellen, ob der Auftrag existiert

If Sel.EOF Then                         ' Auftrag existiert nicht
    msgBox "Auftrag existiert nicht, Ende des Programms"
    Transakt.Rollback
    Exit Function
End If

' Auftrag existiert, alle Daten zu diesem Auftrag suchen und loeschen

' Select Bereich fuer die verschiedenen Positionen des Auftrags:
Set SelAuf = db.OpenRecordset(" Select Posnr,Teilenr From Auftragsposten" _
                           &  " Where AuftrNr = " & AAufNr & ";")

Do Until SelAuf.EOF   'alle Datensaetze bis zum Ende lesen

    Set Sel = db.OpenRecordset(" Select Bezeichnung From Teilestamm" & _
                               " Where Teilnr = " & SelAuf!teilenr & ";")
    msgBox "Pos. " & SelAuf!Posnr & ": " & Sel!Bezeichnung, , "Pos. auflisten"

    ' Eintraege in AVO_Belegung loeschen, die sich auf den Auftrag beziehen
    sqlstr = "Delete From AVO_Belegung Where PosNr = " & SelAuf!Posnr & ";"
    db.Execute sqlstr                   ' Delete ausfuehren

    ' Lesen und Bearbeiten der Eintraege in Relation Teilereservierung:
    Set SelRes = db.OpenRecordset("Select Teilenr, Anzahl " & _
                                  "From Teilereservierung " & _
                                  "Where Posnr = " & SelAuf!Posnr & ";")

    Do Until SelRes.EOF
        ' Reservierungsdaten in Relation Lager anpassen:
        sqlstr = " Update Lager Set Reserviert = Reserviert - " & SelRes!Anzahl _
                & " Where Teilenr = " & SelReserv!teilenr & ";"
        db.Execute sqlstr               ' Update ausfuehren

        ' Teilereservierungen loeschen, die sich auf Auftragsposten beziehen:
        SelReserv.Delete      ' loescht aktuellen Eintrag aus Rel. Teilereservierung

        SelReserv.MoveNext  ' naechster Eintrag
    Loop
```

```
    SelReserv.Close

    ' Aktuellen Auftragsposten loeschen:
    SelAuftr.Delete

    SelAuftr.MoveNext    ' Zum nächsten Datensatz wechseln.
Loop

SelAuftr.Close

If msgBox("Auftrag wirklich löschen", vbYesNo, "Löschen") = vbYes Then
    ' Auftrag selbst muss noch geloescht werden:
    sqlstr = " Delete From Auftrag Where AuftrNr = " & AAufNr & ";"
    db.Execute sqlstr                    ' Loeschen durchfuehren

    Transakt.CommitTrans                 ' Transaktion beenden
    msgBox "Auftrag ist aus Datenbank entfernt", vbInformation
Else
    Transakt.Rollback                    ' alle Aenderungen zuruecknehmen
    msgBox "Transaktion wurde vollstaendig zurueckgesetzt", vbInformation
End If

Exit Function

FehlerBehandlg:
    Transakt.Rollback
    msgBox "Fehler in Funktion ArtikelEinfuegen aufgetreten", vbInformation

End Function
```

Sicherlich wurde mit diesen Beispielen ein kleiner Einblick in die Datenbankprogrammierung gewonnen. Eine Fülle von weiteren Programmieraufgaben sind denkbar: die Aufnahme eines neuen Auftrags, eines neuen Kunden, eines Lieferanten oder eines neuen Mitarbeiters. Auch das Entfernen von Personen oder die Suche nach bestimmten Aufträgen oder Teilen sind lohnenswerte Programmieraufgaben.

Anhang B SQL-Syntaxdiagramme

Im Folgenden ist eine Zusammenfassung der in diesem Buch verwendeten SQL-2 Syntax angegeben. Sie besitzt keinen Anspruch auf Vollständigkeit. Eine komplette Syntax finden wir im Original, siehe [SQL92], und einen sehr guten Querschnitt enthalten [DaDa98] und [MeSi93].

Die Syntax ist wie folgt aufgebaut: Worte in Großbuchstaben sind reservierte Bezeichner in SQL und müssen bis auf Groß- und Kleinschreibung genauso im Befehl auftauchen. Alle anderen Worte sind wahlfreie Bezeichner. Dies sind entweder Namen oder noch weiter zusammengesetzte Bezeichner. In diesem Fall sind diese dann noch angegeben.

In den Syntaxdiagrammen tauchen auch Sonderzeichen auf. Bis auf eckige Klammern, geschweifte Klammern, senkrechte Striche und Fortsetzungszeichen sind diese ebenfalls reservierte Bezeichner und müssen im Befehl genauso angegeben werden. Weiter werden Bindestriche bei längeren Bezeichnern gewählt. Die Bedeutung der genannten Sonderzeichen ist:

- Eckige Klammern (‚[', ‚]'): Alle zwischen eckigen Klammern stehenden Ausdrücke sind wahlfrei und können weggelassen werden.
- Geschweifte Klammern (‚{', ‚}'): Bei den zwischen den eckigen Klammern angegebenen Ausdrücken muss genau ein Ausdruck verwendet werden. Die einzelnen Ausdrücke sind durch senkrechte Striche voneinander getrennt.
- Senkrechte Striche (‚|'): Die senkrechten Striche trennen zwei oder mehr Ausdrücke voneinander. Sie treten im Zusammenhang mit geschweiften und eckigen Klammern auf. Bei geschweiften Klammern ist genau ein, bei eckigen Klammern höchstens ein Ausdruck auszuwählen.
- Fortsetzung (‚…'): Der vorherige Ausdruck darf ein oder mehrmals wiederholt werden. Häufig dienen reservierte Bezeichner als Wiederholungsoperatoren. Diese sind dann direkt vor dem Fortsetzungszeichen angegeben. Dieser Wiederholungsoperator steht dann immer zwischen dem vorherigen und dem nächsten Operator. Oft ist das Komma dieser Operator. In diesem Fall werden die einzelnen Ausdrücke also durch Kommata getrennt.

DDL Befehle:

Create-Schema-Befehl :
CREATE SCHEMA [Schemaname] [AUTHORIZATION Benutzer]
[Create-Domain-Befehl | Create-Table-Befehl | Create-View-Befehl
| Create-Assertion-Befehl | Grant-Befehl]

Create-Domain-Befehl :
CREATE DOMAIN Gebietsname [AS] Datentyp
[[CONSTRAINT Bedingungsname] CHECK (Bedingung)] [...]

Create-Table-Befehl :
CREATE [TEMPORARY] TABLE Tabellenname
({ Spalte { Datentyp | Gebietsname } [Spaltenbedingung [...]]
| Tabellenbedingung } [, ...])

Create-View-Befehl :
CREATE VIEW Sichtname [(Spalte [, ...])]
AS Tabellenausdruck
[WITH CHECK OPTION]

Create-Assertion-Befehl :
CREATE ASSERTION Bedingungsname CHECK (Bedingung)

Create-Index-Befehl (nur SQL-1) :
CREATE [UNIQUE] INDEX Name ON
Tabellenname ({ Spalte [ASC | DESC] } [, ...])

Alter-Domain-Befehl :
ALTER DOMAIN Gebietsname
{ ADD Tabellenbedingung | DROP CONSTRAINT Bedingungsname }

Alter-Table-Befehl :
ALTER TABLE Tabellenname
{ ADD [COLUMN] Spalte { Datentyp | Gebietsname }
[Spaltenbedingung [...]]
| DROP [COLUMN] Spalte { RESTRICT | CASCADE }
| ADD Tabellenbedingung
| DROP CONSTRAINT Bedingungsname { RESTRICT | CASCADE } }

Grant-Befehl :
GRANT { Zugriffsrecht [, ...] | ALL PRIVILEGES }
ON { DOMAIN Gebietsname | [TABLE] { Tabellenname | Sichtname } }
TO { Benutzer [, ...] | PUBLIC }
[WITH GRANT OPTION]

Zugriffsrecht :
SELECT | INSERT [(Spalte [, ...])] | UPDATE [(Spalte [, ...])]
| DELETE | REFERENCES [(Spalte [, ...])] | USAGE

Drop-Schema-Befehl :
DROP SCHEMA Schemaname { RESTRICT | CASCADE }

Drop-Domain-Befehl :
DROP DOMAIN Gebietsname { RESTRICT | CASCADE }

Drop-Table-Befehl :
DROP TABLE Tabellenname { RESTRICT | CASCADE }

Drop-View-Befehl :
DROP VIEW Sichtname { RESTRICT | CASCADE }

Drop-Assertion-Befehl :
DROP ASSERTION Bedingungsname

Drop-Index-Befehl (nur SQL-1) :
DROP INDEX Name

Revoke-Befehl :
REVOKE [GRANT OPTION FOR]
{ Zugriffsrecht [, ...] | ALL PRIVILEGES }
ON { DOMAIN Gebietsname | [TABLE] { Tabellenname | Sichtname } }
FROM { Benutzer [, ...] | PUBLIC } { RESTRICT | CASCADE }

Datentyp :
INTEGER | INT | SMALLINT | NUMERIC (Zahl [, Zahl])
| FLOAT [(Zahl)] | DECIMAL [(Zahl)] | DEC [(Zahl)]
| BIT (Zahl) | CHARACTER [(Zahl)] | CHAR [(Zahl)]
| CHARACTER VARYING [(Zahl)] | VARCHAR [(Zahl)]
| DATE | TIME

DML Befehle :

Select-Hauptteil :
SELECT [ALL | DISTINCT] Spaltenauswahlliste
FROM Tabellenliste
[WHERE Bedingung]
[GROUP BY Spaltenliste
[HAVING Bedingung]]

Insert-Befehl :
INSERT INTO { Tabellenname | Sichtname } [(Spaltenliste)]
Tabellenausdruck

```
Update-Befehl :
   UPDATE    { Tabellenname  |  Sichtname }
   SET       { Spalte  =  { Spaltenausdruck | NULL } }   [ , ... ]
   [ WHERE   { Bedingung  |  CURRENT OF Cursorname }  ]

Delete-Befehl :
   DELETE FROM   { Tabellenname  |  Sichtname }
   [ WHERE       { Bedingung  |  CURRENT OF Cursorname }  ]

Spaltenauswahlliste :
   {   Spaltenausdruck [ [ AS ]  Aliasname ]
   |   [  { Tabellenname  |  Sichtname }  .  ] *  }      [ , ... ]

Tabellenliste :
   Tabellenreferenz   [ , ... ]

Tabellenreferenz :
     { Tabellenname  |  Sichtname }  [ [ AS ]  Name ]
   | Tabellenreferenz   [  NATURAL  INNER
                        | NATURAL  { LEFT  |  RIGHT  |  FULL } [ OUTER ]
                        | UNION  ]                    JOIN  Tabellenreferenz
   | Tabellenreferenz   [  INNER
                        | { LEFT  |  RIGHT  |  FULL } [ OUTER ]
                        ]  JOIN  Tabellenreferenz  ON  Bedingung
   | (  Tabellenreferenz  )

Spaltenliste :
   Spaltenname   [ , ... ]
```

Eingebettete DML Befehle:

```
Einreihen-Select-Befehl :
   SELECT       [ ALL | DISTINCT ]    Spaltenauswahlliste
   INTO         Variablenliste
   FROM         Tabellenliste
   [ WHERE      Bedingung ]
   [ GROUP BY   Spaltenliste
      [ HAVING  Bedingung ]  ]

Declare-Cursor-Befehl :
   DECLARE  Cursorname  CURSOR FOR  Tabellenausdruck
   [ ORDER BY Ordnungsliste ]
   [ FOR    { READ ONLY  |  UPDATE [ OF  Spalte  [ , ... ]  ]  }   ]

Open-Befehl :
   OPEN  Cursorname
```

Fetch-Befehl :
FETCH [FROM] Cursorname INTO Variablenliste

Close-Befehl:
CLOSE Cursorname

Whenever-Befehl :
WHENEVER { SQLERROR | NOT FOUND }
{ CONTINUE | GOTO Labelname }

Variablenliste :
{ Variable [[INDICATOR] Variable] } [, ...]

Ordnungsliste :
{ { Spalte | Zahl } [ASC | DESC] } [, ...]

Bedingungen und Ausdrücke:

Tabellenbedingung :
[CONSTRAINT Bedingungsname]
{ PRIMARY KEY | UNIQUE } (Spalte [, ...])
| [CONSTRAINT Bedingungsname]
FOREIGN KEY (Spalte [, ...])
REFERENCES { Tabellenname | Sichtname } [(Spalte [, ...])]
[ON DELETE { NO ACTION | CASCADE | SET NULL }]
[ON UPDATE { NO ACTION | CASCADE | SET NULL }]
| [CONSTRAINT Bedingungsname]
CHECK (Bedingung)

Spaltenbedingung :
[CONSTRAINT Bedingungsname] NOT NULL
| [CONSTRAINT Bedingungsname] { PRIMARY KEY | UNIQUE }
| [CONSTRAINT Bedingungsname]
REFERENCES { Tabellenname | Sichtname } [(Spalte [, ...])]
[ON DELETE { NO ACTION | CASCADE | SET NULL }]
[ON UPDATE { NO ACTION | CASCADE | SET NULL }]
| [CONSTRAINT Bedingungsname] CHECK (Bedingung)

Bedingung :
Bedingungsterm [OR ...]

Bedingungsterm :
Bedingungsfaktor [AND ...]

```
Bedingungsfaktor :
   [ NOT ]
   {   ( Bedingung )
   |   Auswahlliste  { = | < | <= | > | >= | <> }  Auswahlliste
   |   Auswahlliste  [ NOT ]  BETWEEN  Auswahlliste  AND  Auswahlliste
   |   Zeichenkettenausdruck  [ NOT ]  LIKE  Zeichenkettenausdruck
   |   Auswahlliste  [ NOT ]  IN  (  Tabellenausdruck  )
   |   Auswahlliste  IS  [ NOT ]  NULL
   |   Auswahlliste  MATCH  (  Tabellenausdruck  )
   |   Spaltenausdruck  [ NOT ]  IN  (  Spaltenausdruck  [ , ... ]  )
   |   Auswahlliste  { = | < | <= | > | >= | <> }  { ALL | ANY | SOME }
                                                    (  Tabellenausdruck  )
   |   EXISTS  (  Tabellenausdruck  )
   |   UNIQUE  (  Tabellenausdruck  )          }

Tabellenausdruck :
   Tabellenterm   [ { UNION  |  EXCEPT }  [ ALL ]  ... ]

Tabellenterm :
   Tabellenfaktor [ INTERSECT [ ALL ]  ... ]

Tabellenfaktor :
       TABLE   { Tabellenname  |  Sichtname }
   |   VALUES   Auswahlliste
   |   Select-Hauptteil
   |   (  Tabellenausdruck  )

Auswahlliste :
   Spaltenausdruck   |   (  Spaltenausdruck  [ , ... ]  )   |   (  Tabellenausdruck  )

Spaltenausdruck :
       Numerischer-Ausdruck  |  Zeichenkettenausdruck
   |   Datumsausdruck         |  Intervallausdruck

Numerischer-Ausdruck :
   Numerischer-Term   [ { + | - }  ... ]

Numerischer-Term :
   [ + | - ]  Numerischer-Faktor   [ { * | / }  Numerischer-Faktor   [ { * | / }  ... ]   ]

Numerischer-Faktor :
       (  Numerischer-Ausdruck  )     |  Spaltenname     |  Numerische-Konstante
   |   Funktionsaufruf                 |  Statistikfunktionsaufruf
   |   Variable  [ [ INDICATOR ]  Variable ]

Funktionsaufruf :
   „Aufruf einer Funktion, die von SQL unterstützt wird“
```

```
Statistikfunktionsaufruf :
        COUNT ( * )
    |   { AVG | MAX | MIN | SUM | COUNT }
                                ( [ ALL | DISTINCT ] Spaltenausdruck )

Zeichenkettenausdruck :
        ( Zeichenkettenausdruck )   | Spaltenname   | Zeichenkettenkonstante
    |   Funktionsaufruf             | Statistikfunktionsaufruf
    |   Variable [ [ INDICATOR ] Variable ]

Datumsausdruck :
        Datumsterm [ { + | - } Intervallterm [ { + | - } ... ] ]
    |   Intervallausdruck + Datumsterm

Datumsterm :
        Spaltenname | Variable | CURRENT_DATE | Funktionsaufruf
    |   Statistikfunktionsaufruf | ( Datumsausdruck ) | Datumskonstante

Intervallausdruck :
        Intervallterm [ { + | - } ... ]
    |   Datumsausdruck - Datumsterm

Intervallterm :
        [ + | - ] Intervallfaktor [ { * | / } Numerischer-Faktor ] [ { * | / } ... ]
    |   Numerischer-Faktor * [ + | - ] Intervallfaktor

Intervallfaktor :
        Spaltenname | Variable | Funktionsaufruf | Statistikfunktionsaufruf
    |   ( Intervallausdruck ) | Intervallkonstante
```

Transaktionen und Rechnerverbindung :

```
Commit-Befehl :
    COMMIT [ WORK ]

Rollback-Befehl :
    ROLLBACK [ WORK ]

Set-Transaction-Befehl :
    SET TRANSACTION     [ READ ONLY | READ WRITE ]
    [ ISOLATION LEVEL   {  READ UNCOMMITTED | READ COMMITTED
                        | REPEATABLE READ | SERIALIZABLE  } ]

Connect-Befehl :
    CONNECT TO {  DEFAULT
                | { Zahl | Variable } [ AS { Zahl | Variable } ]
                                       [ USER { Zahl | Variable } ] }
```

Disconnect-Befehl :
DISCONNECT { DEFAULT | CURRENT | ALL | Zahl | Variable }

Namen, Variablen, Zahlen:

Katalogname :
Name

Schemaname :
[Katalogname .] Name

Gebietsname :
[Schemaname .] Name

Tabellenname :
[Schemaname .] Name

Sichtname :
[Schemaname .] Name

Bedingungsname :
[Schemaname .] Name

Spaltenname :
[{ Tabellenname | Sichtname | Aliasname } .] Spalte

Zeichenkettenkonstante :
' Zeichen [...] '

Numerische Konstante :
„Gleitpunkt- oder Ganzzahl“

Datumskonstante :
DATE ' Datumsformat '

Datumsformat :
„Datum im Format jjjj-mm-tt “

Intervallkonstante :
INTERVAL [+ | -] ' Jahreszahl - Monatszahl ' YEAR TO MONTH
| INTERVAL [+ | -] ' Zahl ' { YEAR | MONTH | DAY }

Variable :
: Variablenname

Spalte, Benutzer, Cursorname, Aliasname, Name :
Buchstabe [Buchstabe | Ziffer | _] [...]

Zeichen :
„abdruckbares Zeichen des Zeichensatzvorrats“

Buchstabe :
„Groß- oder Kleinbuchstabe des Alphabets ohne deutsche Sonderzeichen“

Jahreszahl, Monatszahl, Zahl :
Ziffer [...]

Ziffer :
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Variablenname, Labelname:
abhängig von der verwendeten Programmiersprache

Anhang C Lösungen zu den Übungsaufgaben

zu Kapitel 1:

1) a) Kalender, Karteien mit fester Nummerierung, nach aufeinanderfolgenden Nummern geordnete Daten (Hotelzimmer- oder Studentenkartei).

   b) Telefonbuch, Verzeichnisindex, Lexikon, alle nach Namen geordneten Daten.

2) Geordnete Listen. Das Einfügen und Entfernen ist zeitaufwendig.

3) Baum wird schieflastig, so dass sich die Zugriffszeiten deutlich erhöhen. Dies kann mittels ausgeglichener oder B-Bäume verhindert werden, bedeutet aber einen zusätzlichen Aufwand (Komplexität, Laufzeit) bei jedem Einfügen oder Entfernen.

4) Minimierung der Plattenzugriffe. Beispiel: ISAM.

5) ISAM splittet den Block, zum Splitting-Verfahren siehe Abschnitt 1.6.

6) 3 Dateizugriffe, davon 2 schreibend (1 Datenblock lesend, 1 Datenblock schreibend, 1 Indexblock schreibend).

7) ISAM, da optimal auf Plattenspeicherung abgestimmt. In ISAM werden die Indexdaten separat gespeichert, so dass diese ganz oder teilweise im Arbeitsspeicher gehalten werden können, was die Plattenzugriffe erheblich reduziert!

8) Viele Datenblöcke enthalten nur wenige Einträge, so dass der Speicherbedarf erheblich zunimmt. Eine komplette Reorganisation schafft Abhilfe.

9) In Anlehnung an die in Abschnitt 1.7 angegebene Hashfunktion 1. Ordnung könnten wir folgende Funktion wählen. Sie hat den Vorteil, dass zu jedem Namen ein anderer Versatz gewählt wird. Selbstverständlich gibt es noch Verbesserungsmöglichkeiten, wir verweisen auf [Mehl88] und [OtWi93]; doch nun zur Hashfunktion 2. Ordnung:

$$h_2(name) = 100 \cdot \left(\left(\left(\sum_{i=1}^{20} i \cdot \mathrm{ord}(name[i]) \right) \Big/ 1000 \right) \bmod 1000 \right)$$

10) Die Eindeutigkeit sowohl des Primärschlüssels als auch des Sekundärschlüssels wird in der Datenorganisation nicht gefordert. Trotzdem ist es zweckmäßig, den Primärschlüssel eindeutig zu wählen. Sekundärschlüssel sind meist Namen, manchmal auch das Geburtsdatum. Diese Schlüssel sind daher von Natur aus nicht eindeutig. Dies gilt auch, wenn der Name als Primärschlüssel gewählt wird. Meist sind jedoch Primärschlüssel eindeutige Nummern (z.B. KFZ-Kennzeichen, Personalnummern).

11) Schnelle Suche mit Suchbegriffen, die nicht der Primärschlüssel sind.

12) Abfragen nach einem eindeutigen Suchbegriff ergeben immer ein eindeutiges (einzelnes) Ergebnis. Andere Abfragen ergeben eine unbekannte Anzahl von Ergebnissen. Im ersten Fall kann das Ergebnis immer in einer Variable gespeichert werden. Im zweiten Fall muss ein entsprechend großes Feld zur Verfügung stehen, ein eventueller Überlauf muss abgefangen werden.
13) Betrachten wir die KFZ-Datei aus Tab. 7. Seien der Name und das Geburtsdatum die beiden Sekundärschlüssel. In jeder der beiden invertierten Dateien finden wir alle KFZ-Kennzeichen (Primärschlüssel) nochmals wieder. Weiter sind jeweils auch der Name und das Geburtsdatum vorhanden. In den insgesamt drei Indizes finden wir außerdem auch viele Daten zu KFZ-Kennzeichen, Namen und Geburtsdatum. Zusammenfassend gilt: Alle KFZ-Kennzeichen existieren mindestens dreimal und höchstens viermal; der Name und das Geburtsdatum existieren mindestens zweimal und höchstens dreimal; alle anderen Daten existieren einmal (in der Originaldatentabelle).
14) Die invertierte Datei! Die Originaldatei ist immer die Ausgangsdatei, alle anderen sind davon abgeleitet und werden im Zweifelsfalle an die Originaldatei wieder angepasst.

zu Kapitel 2:

1) Vorteile: Wissen über die physischen Strukturen ist nicht erforderlich, einfache Programmierung mittels mächtiger Schnittstellenfunktionen, hohe Datensicherheit (Zugriffsschutz, Integrität), Mehrfachzugriff und Recovery werden unterstützt.

   Nachteile: Datenbankverwaltung erforderlich, langsamere Zugriffe, höherer Speicherverbrauch.
2) Der Zugriff sieht in einer Pseudo-Programmiersprache wie folgt aus:

```
Öffne die ISAM-Datei (falls noch nicht geöffnet)
WHILE Dateiende noch nicht erreicht
BEGIN
    lies den nächsten Datensatz
    IF Sorte = 'Weißbier' THEN
        Gib Datensatz aus
END
```

   Wir durchsuchen sequentiell. Bei Verwendung eines ISAM-Indizes wäre die Programmierung noch komplexer!
3) Sammlung logisch verbundener Daten: Daten, die nicht miteinander in Verbindung stehen, werden in getrennten Datenbanken verwaltet.
   Speicherung der Daten mit möglichst wenig Redundanz: Je größer der Datenbestand ist, um so wichtiger wird eine geringe Redundanz, da Widersprüche in den Daten kaum noch erkannt werden können (Suche der Stecknadel im Heuhaufen!).

Abfragemöglichkeit und Änderbarkeit von Daten: wichtig, denn wozu speichern wir sonst Daten?
Logische Unabhängigkeit der Daten von der physischen Struktur: nicht zwingend erforderlich, aber es erleichtert den Zugriff und die Verwaltung der Daten ungemein.
Zugriffsschutz: zwingend, da beispielsweise ein Bankkunde nur seine eigenen Kontodaten lesen darf.
Integrität: zwingend, da beispielsweise das Buchen korrekt ablaufen muss oder die Versicherungsdaten korrekt gespeichert sein müssen.
Mehrfachzugriff: zwingend, da Bankkunden oder Versicherungssachbearbeiter gleichzeitig zugreifen wollen.
Zuverlässigkeit: zwingend, da ein unerlaubtes Eindringen von außen einen enormen Schaden verursachen kann.
Ausfallsicherheit: zwingend, da beispielsweise ein Rechnerabsturz nicht den dauerhaften Ausfall aller oder einzelner Daten nach sich ziehen darf.
Kontrolle: zwingend, um beispielsweise auf Überlast rechtzeitig reagieren zu können. Nichts ist schlimmer, als eine Großdatenbank sozusagen „blind“ laufen zu lassen.

4) Der Administrator ist der einzige, der den Datenbankaufbau verändern, also auch löschen, kann. Er vergibt außerdem Zugriffsrechte an alle Benutzer und übernimmt damit die Verantwortung über das Funktionieren der Datenbank. Diese Verantwortung kann und darf nicht breit über alle Benutzer gestreut werden.

5) Sind für eine Transaktion mehrere Mutationen erforderlich, so muss der Benutzer darauf achten, dass er immer alle diese Mutationen hintereinander ausführt. Dies erfordert eine hohe Disziplin, kann aber per Programm automatisiert werden. Werden aus Versehen eine oder mehrere Mutationen ausgelassen oder stürzt der Rechner zwischen diesen Mutationen ab, so müssen nachträglich die fehlenden Mutationen ermittelt und nachvollzogen werden. Dies kann einen enormen Aufwand bedeuten, im schlimmsten Fall müssen alle Datenbankdaten überprüft werden. Dies kann teilweise automatisiert werden.

   Viele kleine Datenbanken arbeiten ohne Transaktionsmechanismus, weil dieser entweder nicht unterstützt wird, oder weil die Programmierung aufwendiger wird, oder weil dadurch die Antwortzeiten erheblich anwachsen.

6) Relationale Datenbanken sind Datenbanken, die ausschließlich aus Tabellen bestehen und der Zugriff nur über diese Tabellen erfolgt. Ältere nichtrelationale Datenbanken bestehen zwar ebenfalls aus Tabellen (Knoten), die jedoch mittels spezieller Verknüpfungen miteinander verbunden werden. Oder einfacher gesagt: In relationalen Datenbanken werden auch die Verknüpfungen mittels Tabellen realisiert.

7)

a)

Sorte	Hersteller
Export	Löwenbräu
Hell	EKU
Pils	Bischofshof

b)

Sorte	Hersteller	Anzahl
Märzen	Hofbräu	3
Pils	Bischofshof	3

c)

Hersteller	Anzahl
Löwenbräu	22

8) a) Es kommt folgende Zeile zum Getränkelager hinzu:

Nr	Sorte	Hersteller	Typ	Anzahl
18	Export	EKU	6er Pack	8

b) Die Artikel mit den Nummern 24 und 28 werden gelöscht.

c) Von den Artikeln mit den Nummern 28 und 47 steht jetzt in der Spalte *Anzahl* jeweils die Zahl 5.

9)

```
a)  SELECT Sorte, Hersteller FROM Getränkelager WHERE Typ = '6er Pack' ;
b)  SELECT Sorte FROM Getränkelager WHERE Hersteller = 'Löwenbräu' ;
c)  DELETE FROM Getränkelager WHERE Hersteller = 'Kneitinger' ;
d)  UPDATE Getränkelager SET Anzahl = Anzahl - 10
WHERE Hersteller = 'Löwenbräu' AND Sorte = 'Pils' ;
e)  INSERT INTO Getränkelager
VALUES (10, 'Dunkles Weißbier', 'Schneider', 'Träger', 6) ;
```

zu Kapitel 3:

1) Siehe Tab. 14 und Abb. 16.
2) Unterschiede: eindeutig in relationalen Datenbanken, Sortierung nach dem Primärschlüssel in der Datenorganisation.
   Gemeinsamkeiten: dient zur Identifizierung von Einträgen.
3) *Verk_Nr* und *Produktname* zusammen
4) Alle Attribute, für die die Angabe *Not Null* gesetzt wurde. Dies gilt automatisch für den Primärschlüssel.
5) Primärschlüssel:
   Relation Lieferant: *Nr*
   Relation Kunde: *Nr*
   Relation Personal: *Nr*
   Relation Teilestamm: *Teilnr*
   Relation Lager: *Teilenr*
   Relation Arbeitsvorgang: *AVOnr*
   Relation Auftrag: *AuftrNr*
   Relation Auftragsposten: *PosNr*
   Relation Teilestruktur: *Oberteilnr* + *Einzelteilnr*
   Relation Teilereservierung: *Posnr* + *Teilenr*
   Relation AVO_Belegung: *PosNr* + *AVOnr*

   Alternative Schlüssel:
   Relation Auftragsposten: *AuftrNr* + *Teilenr* (je Auftrag erscheint ein Teil nur einmal)

6) Jede Verbindung in Abb. 54 repräsentiert genau einen Fremdschlüssel.

Fremdschlüssel (Relation.Attribut)	bezieht sich auf Relation
Teilestamm.Liefernr	Lieferant
Teilestamm.AVOnr	Arbeitsvorgang
Teilestruktur.Oberteilnr	Teilestamm
Teilestruktur.Einzelteilnr	Teilestamm
Lager.Teilenr	Teilestamm
Auftrag.Kundnr	Kunde
Auftrag.Persnr	Personal
Auftragsposten.AuftrNr	Auftrag
Auftragsposten.Teilenr	Teilestamm
Teilereservierung.Posnr	Auftragsposten
Teilereservierung.Teilenr	Teilestamm
AVO_Belegung.PosNr	Auftragsposten
AVO_Belegung.AVOnr	Arbeitsvorgang

7)

a)

Tupel (Relation: Primärschlüssel)
Personal: 5
Auftrag: 2
Auftrag: 4
Auftragsposten: 21
Auftragsposten: 22
Auftragsposten: 41
Auftragsposten: 42
Auftragsposten: 43
Teilereservierung: 21 + 100002
Teilereservierung: 22 + 500001
Teilereservierung: 22 + 500011
Teilereservierung: 22 + 500012
Teilereservierung: 22 + 500013
Teilereservierung: 22 + 500014
Teilereservierung: 41 + 100001
Teilereservierung: 41 + 500001
Teilereservierung: 41 + 500008
AVO_Belegung: 22 + 3
AVO_Belegung: 22 + 1

b)

Tupel (Relation: Primärschlüssel)
Teilestamm: 500001
Teilestruktur: 200003 + 500001
Teilestruktur: 200004 + 500001
Lager: 500001
Teilereservierung: 22 + 500001
Teilereservierung: 42 + 500001

8)

P.nr	Name	Ort	Vorg.	Geh.	Name2	Ort2	Geh.2
1	Maria Forster	Regensburg	NULL	6800	NULL	NULL	NULL
2	Anna Kraus	Regensburg	1	3400	Maria Forster	Regensbg.	6800
3	Ursula Rank	Sinzing	6	4200	M. Lambert	Barbing	5500
4	Heinz Rolle	Nittenau	1	4600	Maria Forster	Regensbg.	6800
5	Johanna Köster	Obertraubling	1	3200	Maria Forster	Regensbg.	6800
6	M. Lambert	Barbing	NULL	5500	NULL	NULL	NULL
7	Thomas Noster	Regensburg	6	3800	M. Lambert	Barbing	5500
8	Renate Wolters	Kelheim	1	4700	Maria Forster	Regensbg.	6800
9	Ernst Pach	Steinsberg	6	800	M. Lambert	Barbing	5500

9) Beim Join wird vorausgesetzt, dass beide Relationen ein gemeinsames Attribut besitzen.

	min. Kardinalität	max. Kardinalität
A UNION B	max (M, N)	M + N
A JOIN B	0	M * N
A MINUS B	0	M
A TIMES B	M * N	M * N
A INTERSECT B	0	min (M, N)

Bei UNION werden eventuell gleiche Einträge nur einmal gezählt!

zu Kapitel 4:

1) SELECT Name FROM Personal WHERE Gehalt > 4000.00 ;

2) SELECT SUM(Anzahl) FROM Teilereservierung ;

3) SELECT Teilenr, Bezeichnung FROM Lager, Teilestamm
   WHERE Teilnr = Teilenr AND Bestand – Mindbest - Reserviert < 3 ;

4) SELECT Oberteilnr,SUM(Anzahl) FROM Teilestruktur GROUP BY Oberteilnr;

5) SELECT TR.Teilenr, Bezeichnung, Sum (TR.Anzahl)
   FROM Auftragsposten A, Teilereservierung TR, Teilestamm TS
   WHERE A.PosNr = TR.Posnr AND Teilnr = TR.Teilenr AND AuftrNr = 2
   GROUP BY TR.Teilenr, Bezeichnung;

6) In SQL-1:

```
SELECT TR.Teilenr, Bezeichnung, Sum ( TR.Anzahl )
   FROM  Auftragsposten A, Teilereservierung TR, Teilestamm TS
   WHERE A.PosNr = TR.Posnr AND Teilnr = TR.Teilenr AND AuftrNr = 2
   GROUP BY TR.Teilenr, Bezeichnung
   UNION SELECT Teilnr, Bezeichnung, 0 FROM Teilestamm
   WHERE Teilnr NOT IN ( SELECT Teilenr FROM Teilereservierung
                         WHERE Posnr IN
                         ( SELECT PosNr FROM Auftragsposten
                           WHERE AuftrNr = 2 ) );
```

   In SQL-2:

```
SELECT Teilenr, Bezeichnung, Anzahl
   FROM Teilereservierung FULL OUTER JOIN Teilestamm ON Teilnr = Teilenr
   WHERE Posnr IN ( SELECT PosNr FROM Auftragsposten
                    WHERE AuftrNr = 2 ) ;
```

7) UPDATE Lager SET Bestand = Bestand+7 WHERE Teilenr IN
   (SELECT Teilnr FROM Teilestamm WHERE Bezeichnung = 'SATTEL') ;

8) INSERT INTO Teilestamm VALUES
   (100003, 'Damen-Mountainbike', 1300.00, '26 Zoll', 'ST', 'E', NULL, NULL, 6) ;

9) INSERT INTO Kunde SELECT 10, Name, Strasse, PLZ, Ort, Sperre
   FROM Lieferant WHERE Name = 'Firma Gerda Schmidt' ;

10) DELETE FROM Lager WHERE Bestand = 0 ;

11) UPDATE Personal SET Beurteilung = 2, Gehalt = Gehalt+100.00
    WHERE Beurteilung = 1 ;

12) Die Relation *A* bestehe aus den eventuell zusammengesetzten Attributen *A1* und *V*, die Relation *B* aus *B1* und *V*. Unter „*A.V = B.V*“ wird bei zusammengesetztem

Attribut *V* die Überprüfung auf komponentenweise Gleichheit (verknüpft mit dem *And*-Operator) verstanden. Bei der Division sei das Attribut *B1* leer.

	SQL-1	SQL-2
Schnitt	SELECT * FROM A WHERE EXISTS (SELECT * FROM B WHERE „A.* = B.*“)	SELECT * FROM A INTERSECT SELECT * FROM B
Join	SELECT A.A1, A.V, B.B1 FROM A, B WHERE „A.V = B.V“	SELECT * FROM A NATURAL JOIN B
Division	SELECT A.A1 FROM A, B WHERE „A.V = B.V“ GROUP BY A.A1 HAVING COUNT (*) = (SELECT COUNT (*) FROM B)	siehe SQL-1

zu Kapitel 5:

1) Der Primärschlüssel beschreibt ein Tupel eindeutig. Jedes Attribut einer Relation ist daher per definitionem funktional abhängig vom Primärschlüssel. Ist der Primärschlüssel nicht aus mehreren Einzelattributen zusammengesetzt, so sind alle Attribute sogar voll funktional abhängig vom Primärschlüssel. Dies ist aber gerade die Eigenschaft der zweiten Normalform.

2) a) VerkaeuferProdukt: Verk_Nr ⇒ (Verk_Name, PLZ, Verk_Adresse)
(Verk_Nr, Produktname) ⇒ Umsatz

Chemische Elemente: Protonen ⇔ Name ⇔ Symbol
Protonen ⇒ Atomgewicht
Name ⇒ Atomgewicht
Symbol ⇒ Atomgewicht

VerkaeuferProdukt_2NF: siehe Abb. 18 auf Seite 128

b) Lieferant und Kunde: Nr ⇒ (Name, Strasse, PLZ, Ort, Sperre)

Personal: PersNr ⇒ (Name, Strasse, PLZ, Ort, GebDatum, Vorgesetzt, Stand, Gehalt, Beurteilung, Aufgabe)

Teilestamm: Teilnr ⇒ (Bezeichnung, Mass, Einheit, Typ, Liefernr, Zeit, AVOnr)

Teilestruktur: (Oberteilnr, Einzelteilnr) ⇒ Anzahl
Einzelteilnr ⇒ Einheit

Lager: Teilenr ⇒ (Lagerort, Bestand, Mindbest, Reserviert, Bestellt)

Arbeitsvorgang: AVOnr ⇒ (Bezeichnung, Arbzeit, Platznr, PlatzBez, Kosten)
Platznr ⇒ (PlatzBez, Kosten)

Auftrag: AuftrNr ⇒ (Datum, Kundnr, Persnr)

Auftragsposten: PosNr ⇒ (AuftrNr, Teilenr, Anzahl, Gesamtpreis)
(AuftrNr,Teilenr) ⇒ (PosNr,Anzahl,Gesamtpreis)

Teilereservierung: (Posnr, Teilenr) ⇒ Anzahl

AVO_Belegung: (PosNr, AVOnr) ⇒ Anzahl

3) a) VerkaeuferProdukt: Verk_Nr, (Verk_Nr, Produktname)
Chemische Elemente: Protonen, Name, Symbol
VerkaeuferProdukt_2NF: Nr, Verk_Nr, (Produktname, Verk_Nr)

b) Lieferant und Kunde: Nr
Personal: PersNr
Teilestamm: Teilnr
Teilestruktur: (Oberteilnr, Einzelteilnr), Einzelteilnr
Lager: Teilenr
Arbeitsvorgang: AVOnr, Platznr
Auftrag: AuftrNr
Auftragsposten: PosNr, (AuftrNr, Teilenr)
Teilereservierung: (Posnr, Teilenr)
AVO_Belegung: (PosNr, AVOnr)

4) Verkaeufer: Verk_Nr; Produkt: Prod_Nr;
Verknuepfung: (Verk_Nummer, Produkt_Nr)

5) 3. NF: Lieferant, Kunde, Personal, Teilestamm, Lager, Auftrag, Auftragsposten, Teilereservierung, AVO_Belegung
2. NF: Arbeitsvorgang
1. NF: Teilestruktur

6) Betrachten wir die Relation *VerkaeuferProdukt_2NF* aus Tab. 28. Nehmen wir an, dass ein Verkäufer durch die drei Attribute *Verk_Name*, *PLZ* und *Verk_Adresse* eindeutig identifiziert wird, dann ist diese Relation in der 3. NF nach Codd, nicht aber in der 3. NF nach Boyce/Codd!

Begründung: *Nr*, (*Verk_Nr*, *Produktname*) und (*Verk_Name*, *PLZ*, *Verk_Adresse*, *Produktname*) sind Schlüsselkandidaten, wegen der gegenseitigen Abhängigkeit sind *Verk_Nr* und (*Verk_Name*, *PLZ*, *Verk_Adresse*) auch Determinanten, also keine Boyce/Codd-NF. Aber Codd-NF, da die Nichttransitivität nur von Attributen verlangt wird, die nicht Teile von Schlüsselkandidaten sind.

7) *Personal.Vorgesetzt* verweist auf *Personal.Persnr*: on delete set null, on update cascade
Auftrag.Persnr verweist auf *Personal.Persnr*: on delete set null, on update cascade
Auftrag.Kundnr verweist auf *Kunde.Nr*: on delete no action, on update cascade

8) *Lager* ist schwach gegenüber *Teilestamm*, Beziehungsrelationen sind: *Teilestruktur*, *Teilereservierung*, *AVO_Belegung*. *Lager* ist ein Subtyp von *Teilestamm*. *Teilestamm* ist ein Supertyp zu *Lager*.

9) Neue Relation *Rechnung* mit den Attributen *ReNr*, *Datum*, *Mahnung*, *Rabatt*, *Endpreis*, *bezahlt*

Primärschlüssel: *Rechnung.ReNr*
Fremdschlüssel: *Rechnung.ReNr* mit Delete No Action und Update Cascade

zu Kapitel 6:

1) Teilestamm, Auftrag, Auftragsposten, Teilestruktur, Arbeitsvorgang, AVO_Belegung: siehe Anhang A4

```
CREATE TABLE Lieferant ( Nr        INTEGER           PRIMARY KEY,
                         Name      CHARACTER (20) NOT NULL,
                         Strasse   CHARACTER (30),
                         PLZ       CHARACTER (5),
                         Ort       CHARACTER (20),
                         Sperre    CHARACTER   ) ;
```

Analog wird die Relation *Kunde* erzeugt.

```
CREATE TABLE Personal ( PersNr   INTEGER           PRIMARY KEY,
                        Name     CHARACTER (20) NOT NULL,
                        Strasse  CHARACTER (30),
                        PLZ      CHARACTER (5),
                        Ort      CHARACTER (20),
                        GebDatum DATE           NOT NULL,
                        Stand    CHARACTER (6),
                        Vorgesetzt INTEGER   REFERENCES Personal
                        ON DELETE SET NULL  ON UPDATE CASCADE,
                        Gehalt   NUMERIC (10, 2),
                        Beurteilung CHARACTER,
                        Aufgabe  CHARACTER (10) ) ;
CREATE TABLE Lager (    Teilenr  INTEGER   PRIMARY KEY
                        REFERENCES Teilestamm
                        ON DELETE CASCADE  ON UPDATE CASCADE,
                        Lagerort CHARACTER (6),
                        Bestand  SMALLINT  NOT NULL,
                        Mindbest SMALLINT,
                        Reserviert SMALLINT,
                        Bestellt SMALLINT  ) ;
CREATE TABLE Teilereservierung (
      Posnr      INTEGER   REFERENCES Auftrag
                 ON DELETE NO ACTION    ON UPDATE CASCADE,
      Teilenr    INTEGER   REFERENCES Teilestamm
                 ON DELETE NO ACTION    ON UPDATE CASCADE,
      Anzahl     SMALLINT,
      PRIMARY KEY (Posnr, Teilenr)    ) ;
```

2)
```
ALTER TABLE Auftragsposten
   ADD COLUMN Einzelpreis Numeric (10,2) ;
UPDATE Auftragsposten SET Einzelpreis = Gesamtpreis / Anzahl
      WHERE Anzahl IS NOT NULL AND Anzahl <> 0;
```

Die neue Relation ist in 2. NF.

3) Ein eindeutiger Index ist für Namen in der Regel zu einschränkend.

```
CREATE INDEX IBezeichng ON Teilestamm (Bezeichnung) ;
```

4)
```
CREATE INDEX Suche ON Personal (Einsatzort, Sachbearbeiter) ;
```

5)
```
CREATE VIEW VPers AS
  SELECT Nr, Name, Strasse, PLZ, Ort, GebDatum, Stand, Aufgabe
  FROM Personal WHERE Vorgesetzt IS NOT NULL ;
```

Manipulationsbefehle dürfen auf die Sicht angewendet werden.

6) CREATE VIEW VTeilestamm AS SELECT * FROM Teilestamm
   WHERE Einheit IN ('ST', 'CM') AND Typ IN ('E', 'Z', 'F') WITH CHECK OPTION;

7) CREATE VIEW VAuftragsPosten
   (PosNr, AuftrNr, Teilenr, Anzahl, Einzelpreis, Gesamtpreis)
   AS SELECT PosNr, AuftrNr, Teilenr, Anzahl, Gesamtpreis/Anzahl, Gesamtpreis
   FROM Auftragsposten ;

   Die Sicht ist nicht änderbar.

8) Das Hinzufügen von Schemaelementen zu einem Schema erfolgt einfach mittels Qualifizierung. Das Hinzufügen einer Sicht namens *VPers* zum Schema *Verkauf* geschieht beispielsweise wie folgt:

   CREATE VIEW Verkauf.VPers AS SELECT …

zu Kapitel 7:

1) Die Transaktion ist gültig, da nur die Information aus der Logdatei zählt.
2) Checkpoints sind Zeitpunkte, wo alle gültigen Datenbankdaten im Arbeitsspeicher auf das nicht flüchtige externe Medium (meist Magnetplatte) geschrieben werden. Ohne Checkpoints würden häufig angefasste Daten (Hot Spots) immer im Arbeitsspeicher stehenbleiben und nie auf dem externen Medium aktualisiert werden. Im Fehlerfall müsste dann zwecks Restaurierung immer die gesamte Logdatei durchsucht werden, was sehr zeitaufwendig wäre.
3) Recovery senkt wegen des zusätzlichen Overheads den Durchsatz und die mittleren Antwortzeiten des Datenbankbetriebs deutlich ab. Entsprechend schnellere Rechner und Speichermedium sind wesentlich teurer. Es muss daher abgewägt werden, was wichtiger ist, ein zuverlässiger Datenbankbetrieb oder niedrige Betriebskosten.
4) Rücksetzen dieser Transaktion mittels eines *Rollback Work* inklusive der Freigabe von gehaltenen Sperren. Fehlermeldung an den Benutzer. Empfohlen wird eine Protokollierung des Fehlers zwecks späterer Analyse.
5) Anhalten des gesamten Datenbankbetriebs; Rücksetzen aller noch nicht abgeschlossenen Transaktionen mittels *Rollback Work*; Abklemmen der Festplatte; Montieren einer neuen Festplatte; Kopieren der letzten Sicherungen auf die neue Festplatte; Nachfahren der Änderungen seit der letzten Sicherung mittels der Logdateiinformationen; Starten des Datenbankbetriebs.
6) Wenn sichergestellt ist, dass genau bei Transaktionsende die geänderten Daten dieser Transaktion auf Festplatte aktualisiert werden.
7) Nie! Ein Ausfall der Festplatte mit den gespeicherten Datenbankdaten würde sonst zu Datenverlusten führen.
8) Notwendig, sobald eine Transaktion auf mehr als eine Datenbank schreibend zugreift, für die kein gemeinsames Log geführt wird.
9) Nein!

10) Je Datenbank muss ein Log geführt werden, zusätzlich ist ein globales Log erforderlich. Der Hauptaufwand ist in der Regel jedoch der hohe Kommunikationsoverhead; siehe auch Kapitel 11.
11) In der Praxis werden Deadlocks meist mittels Wartegraphen erkannt. Eine der betroffenen Transaktionen wird zurückgesetzt.
12) Optimistisches Verfahren. Steigt die Konfliktwahrscheinlichkeit auf wenige Prozent, so schaukelt sich das System katastrophal auf.
13) Nein.
14) Nur teilweise: Transaktionen, die auf gemeinsame Daten zugreifen, werden serialisiert; eine Reihenfolge kann also angegeben werden. Für alle anderen Transaktionen, die gleichzeitig ablaufen, gilt dies nicht.
15) Nein. Auch ohne Sperrmechanismen wäre (zufälligerweise) keine Inkonsistenz aufgetreten.

zu Kapitel 8:

1) Ja, falls das DBMS keine internen Optimierungen (z.B. Ablegen und Merken der Rechte im Cache) vornimmt.
2) GRANT Select, Update (Bestand, Reserviert, Bestellt) ON TABLE Lager TO Gast ;
3) REVOKE Select, Update (Bestand, Reserviert, Bestellt) ON TABLE Lager FROM Gast ;
4) CREATE VIEW VLager AS SELECT Teilenr, Lagerort, Bestand FROM Lager
   WHERE Bestand >= Mindbest + Reserviert ;
   REVOKE ALL PRIVILEGES ON TABLE Lager FROM Gast ;
   GRANT Select ON TABLE VLager TO Gast WITH GRANT OPTION ;
5) CREATE VIEW VLieferant AS SELECT Nr, Name, Strasse, PLZ, Ort FROM Lieferant
   WHERE Ort = 'Regensburg' ;
   REVOKE ALL PRIVILEGES ON TABLE Lieferant FROM Gast ;
   GRANT Select ON TABLE VLieferant TO Gast ;
6) ALTER TABLE Personal ADD CONSTRAINT PersIntegr CHECK (
   GebDatum - CURRENT_DATE BETWEEN 365*15 AND 365*65 AND
   Stand IN ('led', 'verh', 'gesch', 'verw') AND Gehalt BETWEEN 800 AND 10000
   AND (Beurteilung IS NULL OR Beurteilung BETWEEN 1 AND 10)) ;
7) CREATE ASSERTION PersIntegr CHECK (NOT EXISTS
   (SELECT * FROM Personal WHERE GebDatum - CURRENT_DATE NOT
   BETWEEN 365*15 AND 365*65 OR Stand NOT IN ('led', 'verh', 'gesch', 'verw')
   OR Gehalt NOT BETWEEN 800 AND 10000 OR
   (Beurteilung IS NOT NULL AND Beurteilung NOT BETWEEN 1 AND 10))) ;
8) Es muss eine Sicht eingerichtet werden. Zugriffe direkt auf die Relation *Personal* dürfen nicht vergeben werden.

   CREATE VIEW VPers AS SELECT * FROM Personal WHERE
   GebDatum - CURRENT_DATE BETWEEN 365*15 AND 365*65 AND

```
        Stand IN ('led', 'verh', 'gesch', 'verw')  AND  Gehalt BETWEEN 800 AND 10000
        AND ( Beurteilung IS NULL OR Beurteilung BETWEEN 1 AND 10 )
        WITH CHECK OPTION ;
    REVOKE ALL PRIVILEGES ON TABLE Personal FROM PUBLIC ;
```

9) CREATE DOMAIN Berufsbezeichnung AS CHARACTER(12) CONSTRAINT Berufe1 CHECK (VALUE IN ('Manager', 'Vertreter', 'S-Arbeit', 'Sekretär', 'Meister', 'Arbeiter', 'Sachbearb', 'Azubi')) ;

10) ALTER DOMAIN Berufsbezeichnung ADD CONSTRAINT Berufe2 CHECK (VALUE IN ('Manager', 'Vertreter', 'S-Arbeit', 'Sekretär', 'Meister', 'Arbeiter', 'Sachbearb', 'Azubi', 'Lackierer', 'Schlosser')) ;
ALTER DOMAIN Berufsbezeichnung DROP CONSTRAINT Berufe1 ;

zu Kapitel 9:

1) auszugsweise, ohne Einloggen in die Datenbank:

```
EXEC SQL BEGIN DECLARE SECTION;
   char name[20], strasse[20], ort[15], plz[6];
   int nr, sperre;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT * INTO :nr, :name, :strasse, :plz, :ort, :sperre
            FROM lieferant WHERE Nr = 1;
cout  << "Name: " << name << ", Strasse: " << strasse << ", PLZ: " << plz
      << ", Ort: " << ort << ", Sperre: " << sperre << endl ;
```

2) auszugsweise, ohne Einloggen in die Datenbank:

```
EXEC SQL BEGIN DECLARE SECTION;
   char name[20];
   int anzahl;
EXEC SQL END DECLARE SECTION;
cout << "Bitte den gewünschten Familiennamen eingeben: " ;
cin >> name+1 ;  name[0]='%';          // wegen Like-Ausdruck in Where-Klausel
EXEC SQL SELECT COUNT (*) INTO :anzahl FROM personal
   WHERE Name LIKE :name;
if (anzahl == 0)
   cout << "Personaltabelle enthält Familiennamen " << name+1 << " nicht\n" ;
else
   cout << "Personaltabelle enthält " << name+1 << " " << anzahl << "-mal\n" ;
```

3) auszugsweise, ohne Einloggen in die Datenbank:

```
EXEC SQL BEGIN DECLARE SECTION;
   char name[25];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT Name INTO :name FROM Personal WHERE Nr=10;
if (strcmp(SQLSTATE, "00000") == 0)
   cout << "Unter Personalnummer 10 steht die Person: " << name << endl ;
else if (strcmp(SQLSTATE, "02000") == 0)
   cout << "Die Personalnummer 10 ist nicht vergeben!\n"
else
   cout << "Fehler beim Ausfuehren des Select-Befehls.\n" ;
```

4) EXEC SQL DECLARE Curs CURSOR FOR ... ;

```
        ...
EXEC SQL OPEN Curs ;
do
   EXEC SQL FETCH FROM Curs INTO ... ;
   if ((i =strcmp (SQLSTATE, "02000")) != 0)        /* oder: SQLCODE != 100 */
   {
   ...
   }
while ( i != 0 )
EXEC SQL CLOSE Curs ;
```

5) Umfangreiches Programm, siehe Hinweise in Anhang D.
6) Umfangreiches Programm, siehe Hinweise in Anhang D.

zu Kapitel 10:

1) Es muss ein komplettes Datenbankverwaltungssystem (DBMS) geschaffen werden, wobei die Anforderungen aus Kapitel 2 zu erfüllen sind. Unter anderem bereitet die Unterstützung der Integrität große Probleme, da mathematisch fundierte Theorien, auf die aufgesetzt werden könnte, fehlen.
2) Gemeinsamkeiten: beschreibt ein Tupel (Satz) eindeutig, darf nicht NULL sein.

   Unterschiede: In relationalen Datenbanken sind die Tupel nicht nach dem Primärschlüssel sortiert.
3) Indexe.
4) Gemeinsamkeiten: weiterer schneller Suchzugriff.

   Unterschiede: In IMS existiert nur maximal ein Sekundärschlüssel mit einer hierarchischen Struktur.
5) Netzwerkartige Datenbanken: Sets; hierarchische Datenbanken: Hierarchie.
6) In relationalen Datenbanken werden Relationen verwaltet, in CODASYL-Datenbanken Records und Sets. Dies verdoppelt mindestens die Anzahl der DDL-Befehle in CODASYL-Datenbanken. Analoges gilt für die Zugriffsbefehle.
7) Gemeinsamkeiten: tabellenartiger Aufbau.

   Unterschiede: In relationalen Datenbanken sind die Tupel nicht sortiert und enthalten zusätzlich Fremdschlüssel.
8) Im Prinzip ist eine Überführung in beiden Richtungen immer möglich (siehe auch Aufgabe 10).
9) Stärken: minimaler Speicherbedarf, hoher Durchsatz, sehr gute Antwortzeiten.

   Schwächen: nur geeignet, wenn die Daten leicht als Hierarchie angeordnet werden können; die Verwaltung und Zugriffe sind sehr komplex; genaue Kenntnisse der physischen Struktur auch für den Anwender erforderlich; Änderungen der Datenbankstruktur sind kaum möglich.
10) Relationale Datenbanken besitzen eine sehr einfache und leicht erweiterbare Struktur. Zugriffe mittels SQL sind wesentlich einfacher zu programmieren als mittels prozeduraler Zugriffssprachen. SQL ist sehr weit verbreitet, so dass die

Einarbeitungszeit für neue Mitarbeiter reduziert wird. Die Umstellung scheitert entweder am nicht allzu geringen Aufwand, meist jedoch am höheren Speicherplatzbedarf (Platten- und Arbeitsspeicher) und am schlechteren Antwortzeit- und Durchsatzverhalten.

zu Kapitel 11:

1) Fragmentierung: Daten einer Relation können fragmentiert sein, d.h. auf mehrere Rechner verteilt sein.
   Replikation: Daten können auf mehreren Rechnern gleichzeitig gehalten werden.
   Unabhängigkeit bedeutet in beiden Fällen, dass sich das System an seiner Schnittstelle nicht anders verhält, ob nun mit oder ohne Fragmentierung oder Replikation gearbeitet wird.
2) Eine Datenbank heißt *verteilt*, wenn die zusammengehörigen Daten dieser Datenbank auf mindestens zwei Rechnern aufgeteilt sind und von einem gemeinsamen Datenbankverwaltungssystem verwaltet werden.
3) In einem verteilten Datenbanksystem können selbstverständlich auch lokale Deadlocks auftreten. Denken wir nur an eine Transaktion, die nur lokale Zugriffe durchführt. Es können also jederzeit sowohl globale als auch lokale Deadlocks vorkommen.
4) Die Regel 2 ist dann erfüllbar, wenn der Koordinator nicht zentral auf einem Rechner gehalten wird. Übernimmt immer derjenige Rechner die Koordination, auf dem eine globale Transaktion gestartet wurde, so ist Regel 2 nicht verletzt.
5) Individuell zu beantworten.
6) Der Primärschlüssel einer Relation beschreibt ein Tupel eindeutig. Er sollte daher nur aus atomaren Attributen bestehen. Eine Determinante ist deshalb ein Attribut, das nur aus atomaren Einzelattributen besteht, von dem ein anderes voll funktional abhängt.
7) Mittels der Einführung von NF^2-Relationen und des *Recursive-Union*-Operators.
8) Zunächst sind Datentypen (insbesondere variable Felder und eingebettete Relationen) mittels des *Create-Type*-Befehls zu definieren. Hier sind als Datentypen auch Unterobjekte zugelassen. Die Erzeugen einer Relation dieses Typs erfolgt dann mit einem syntaktisch erweiterten *Create-Table*-Befehl.
9) Individuell zu beantworten.
10)
```
CREATE OR REPLACE TYPE TPerson AS OBJECT
  ( Name CHARACTER (20),  Adresse TAdresse,  GebDatum DATE ) ;
CREATE TABLE PersonalNeu
  ( Persnr INTEGER,  Person TPerson,  Stand CHARACTER(6),
    Vorgesetzt INTEGER REFERENCES PersonalNeu,  Gehalt NUMERIC(10,2),
    Beurteilung SMALLINT,  Aufgabe CHARACTER(10) );
```
11)
```
SELECT * FROM THE ( SELECT Einzelposten FROM AuftragNeu
                              WHERE AuftrNr = 4 )  T
          WHERE T.Gesamtpreis > 100 ;
```

Anhang D Hinweise zu Begleitprogrammen

„Übung macht den Meister“, so heißt ein bekanntes Sprichwort. Und Übung und nochmals Übung empfiehlt auch der Autor des vorliegenden Buches und bietet Software zum Erstellen der *Radl*-Datenbank kostenlos an. Somit können die zahlreichen Beispiele aus diesem Buch zu Hause, am Arbeitsplatz oder an der Hochschule nachvollzogen und eingeübt werden.

Je ein Softwarepaket steht für die beiden weit verbreiteten Datenbanken Oracle (ab V7.0) und MS-Access97 zur Verfügung. Dieses enthält jeweils:

- das vollständige Installationsprogramm zum Installieren der Beispieldatenbank *Radl*,
- die umfangreichen Beispielprogramme aus Kapitel 9 und Anhang A,
- die Lösungen der Übungsaufgaben 5 und 6 aus Kapitel 5,
- Hinweise zur Anwendung dieser Programme unter MS-Access97 und Oracle.

Steht eine Oracle-Datenbank mit einer eigenen Kennung zur Verfügung, so wird empfohlen, die Beispieldatenbank unter Oracle zu üben. Der Autor hat die gesamte Software aber auch auf die im PC-Bereich weitverbreitete Datenbank MS-Access97 portiert, um möglichst viele Leser zum „Meister“ zu führen. Die Software kann direkt von der Internetadresse des Autors heruntergeladen werden. Die Adresse lautet:

http://homepages.fh-regensburg.de/~sce39014/

Unter dem Menüpunkt „Datenbanken und SQL“ finden sich Anweisungen zum Herunterladen der Software und Informationen zur Installation.

Lohnenswert ist auch die Internetseite des Teubner-Verlags. Dort findet sich neben Informationen zu dem vorliegenden Buch auch ein direkter Link auf die Internetseite des Autors. Die Internetadresse des Teubner-Verlags lautet:

http://www.teubner.de

Literaturhinweis

Zu Datenbanken gibt es eine Fülle von Literatur. Im Folgenden ist eine Liste der von mir empfohlenen bzw. in diesem Buch verwendeten Literatur angegeben. Nicht aufgeführt sind die zahlreichen Handbücher der einzelnen Datenbankhersteller. Diese Bücher sind in der Praxis meist ein Muss, da immer wieder feine Unterschiede der Herstellerimplementierungen zum SQL-Standard existieren. Beim Erwerb der entsprechenden Software werden diese Handbücher in der Regel mitgeliefert, zumindest als Online-Manuale.

[Chen76] Chen: The Entity-Relationship Model - Toward a Unified View of Data, ACM TODS 1, Nr. 1, 1976

[ChKn91] Chen / Knöll: Der Entity-Relationship-Ansatz zum logischen Systementwurf, BI, 1991

[Codd70] Codd: A Relational Model of Data for large Shared Data Banks, CACM 13, Nr. 6, 1970

[Codd72] Codd: Relational Completeness of Data Base Sublanguages, Data Base Systems, Series 6, Prentice Hall, 1972

[Date83] Date: An Introduction to Database Systems, Vol. 2, Addison-Wesley, 1983

[Date90] Date: What is a Distributed Database?, Relational Database Writings 1985-1990, Addison-Welsey, 1990

[Date95] Date: An Introduction to Database Systems, Vol. 1, Addison-Wesley, 1995

[DaDa98] Date / Darwen: SQL – Der Standard, Addison-Wesley, 1998

[DaDa98a] Date / Darwen: Foundation for Object/Relational Databases, Addison-Wesley, 1998

[Deux91] Deux: The O_2 System, Communication of the ACM 34(10), S. 34-48, 1991

[Dors98] Dorsey: Oracle8 Object-Oriented Design, Addison-Wesley, 1998

[Gee77] McGee: The IMS/VS System, IBM System Journal 16, No. 2, 1977

[Gopa92] Gopas Software: Ontos - Die objektorientierte Datenbank, 1992

[Heue97] Heuer: Objektorientierte Datenbanken, Addison-Wesley, 1992

[SQL87] ISO 9075:1987 Database Language SQL (SQL-1 !)

[SQL89] ISO 9075:1989 Database Language SQL (SQL-1 mit Integritätserweiterungen)

[SQL92] ISO 9075:1992 Database Language SQL (SQL-2 !)

[SQL3] ISO working draft (May 1992) Database Language SQL3 (Arbeitspapier zu SQL-3)

[KeRi90] Kernighan / Ritchie: Programmieren in C, Hanser, 1990

[Khos93] Khoshafian: Object-Oriented Databases, John-Wiley, 1993

[KoLo97] Koch / Loney: Oracle8. The Complete Reference, Oracle Press, 1997
[Kudl88] Kudlich: Datenbank-Design, Springer, 1988
[Kudl92] Kudlich: Verteilte Datenbanken, Siemens-Nixdorf, 1992
[Lans94] van der Lans: An Introduction to SQL, Addison-Wesley, 1994
[MaFr93] Marsch / Fritze: SQL, Vieweg, 1993
[Meie97] Meier: Relationale Datenbanken, Springer, 1997
[Mehl88] Mehlhorn: Datenstrukturen und effiziente Algorithmen, Band 1, Teubner, 1988
[MeSi93] Melten / Simon: Unterstanding the New SQL, Morgan Kaufmann, 1993
[OtWi93] Ottmann / Widmayer: Algorithmen und Datenstrukturen, BI, 1993
[Scha97] Schader: Objektorientierte Datenbanken, Springer, 1997
[Schl93] Schlageter: Objektorientierte Datenbanksysteme, Elektronische Weiterbildung unter Windows, Fernuniversität Hagen, Computer Based Training, Addison-Wesley, 1993
[Schw92] Schwinn: Relationale Datenbanksysteme, Hanser, 1992
[Sedg90] Sedgewick: Algorithmen, Addison-Wesley, 1990
[StKe91] Stonebraker / Kemnitz: The Postgres next generation database management system. Communications of the ACM 34(10), S. 78-92, 1991
[UDS83] Das universelle Datenbanksystem UDS, Verfahrensbeschreibung, Siemens Nixdorf Informationssysteme AG, 1983
[UDS86] UDS - Universelles Datenbanksystem im BS2000, Siemens AG, 1986
[Vett91] Vetter: Aufbau betrieblicher Informationssysteme, Teubner, 1991
[Voss91] Vossen: Data Models, Database Languages und Database Management Systems, Addison-Wesley, 1991
[Wirt83] Wirth: Algorithmen und Datenstrukturen, Teubner, 1983
[Zehn98] Zehnder: Informationssysteme und Datenbanken, Teubner, 1998

Sachverzeichnis