3 Datenspeicherung, Datenbanken

Dieses Kapitel befaßt sich mit der Speicherung und der Verwaltung von Daten auf externen Speichern (Festplatten, Disketten, Magnetbändern, ...). Da riesige Datenmengen auf diese Weise gespeichert werden können, kommt der Organisation der Speicherung und damit verbunden dem gezielten (möglichst schnellen) Zugriff auf bestimmte Informationen die entscheidende Rolle bei allen nachfolgend beschriebenen Problemen zu.

Natürlich darf sich der Anwender in der Regel nicht selbst um die Adressierung bei der Datenspeicherung kümmern müssen. Wie schon bei den höheren Programmiersprachen werden Daten nicht über ihre physikalischen Adressen, sondern über **Namen** angesprochen. Hinter einem Namen kann z. B. ein einzelnes Datenelement, eine Datei, eine Gruppe von Dateien oder eine Datenbank stehen. Damit werden dem Anwender (wie dem Programmierer) die Probleme der physischen Organisation der Datenspeicherung weitgehend abgenommen (von Kapazitätsproblemen, den Problemen mit den Grenzen von Übertragungsgeschwindigkeiten usw. kann der Anwender allerdings nicht befreit werden).

Im Abschnitt 3.1 werden (ergänzend zu den Themen, die bereits Gegenstand der Programmierausbildung im Fach Informatik 1 waren) einige Grundkonzepte der Datenorganisation in Dateien (und ihre Realisierung mit Hilfe einer höheren Programmiersprache) behandelt, in den nachfolgenden Abschnitten werden die Prinzipien der Arbeit mit Datenbanken dargestellt.

3.1 Sequentieller und direkter Zugriff

Die historische Entwicklung der Hardware-Komponenten hat entscheidenden Einfluß auf die Dateiorganisation gehabt:

- Auf **Magnetbändern** werden Daten **sequentiell** gespeichert und können nur in der gepeicherten Reihenfolge wieder abgerufen werden. Die Speicherung wird logisch untergliedert in Datensätze (**Records**). Diese können unterschiedliche Längen haben, ihnen ist jedoch keine physikalische Adresse zugeordnet, so daß die Suche nach einem bestimmten Record stets mit dem systematischen Durcharbeiten der Datei ("sequentielles Suchen") verbunden ist.
- Auf **Magnetplatten** können Daten **direkt adressierbar** gespeichert werden. Diese **Direct Access Files** werden ebenfalls in Records unterteilt, denen jeweils eine Adresse zugeordnet wird. Unter dieser Adresse kann ein Record direkt angesprochen, gelesen und gegebenenfalls geändert werden.

Hinweis:

Die Modellvorstellungen "Magnetband" und "Magnetplatte" sind für das Verständnis sehr nützlich, obwohl gegenwärtig fast alle sequentiellen Dateien auf Speichern abgelegt werden, die auch für die Aufnahme direkt adressierbarer Dateien geeignet sind. Am "logischen Erscheinungsbild" und damit an den Zugriffsstrategien ändert sich damit allerdings nichts.

3.1.1 Direct Access Files in FORTRAN

Die für das Arbeiten mit sequentiellen Files (Informatik 1) in der Programmiersprache FORTRAN verwendeten Befehle **OPEN**, **CLOSE**, **INQUIRE**, **READ** und **WRITE** sind auch für das Arbeiten mit Direct Access Files vorgesehen (die für sequentielle Files wichtigen Befehle BACKSPACE und REWIND werden nicht benötigt, es macht keinen Sinn, um einen Record oder an den Anfang des Files "zurückzuspulen", wenn jeder Record ohnehin direkt unter einer Adresse angesprochen wird).

Die Standardannahme für die genannten Befehle ist immer "sequentiell" (typische "Ansprechpartner" für READ und WRITE sind Tastatur, Bildschirm und Drucker, die sequentiell anzusprechen sind). Zum Öffnen eines Direct Access Files müssen (neben der **UNIT**-Nummer und dem File-Namen) mindestens zwei weitere Parameter in der OPEN-Anweisung angegeben werden: **ACCESS='DIRECT'** definiert die Art des Zugriffs (Standardeinstellung ist ACCESS='SEQUENTIAL') und **RECL=recl** legt die (für alle Records einheitliche) Länge der Records **recl** (Maßeinheit: Byte) fest, Beispiel:

OPEN (10 , FILE = 'wertetab.dat' , ACCESS = 'DIRECT' , RECL = 200)

öffnet 'wertetab.dat' als Direct Access File mit einer Record-Länge von 200 Byte.

Hinweis: Während für den **STATUS** wie bei sequentiellen Files **'UNKNOWN'** angenommen wird, wenn der STATUS-Parameter nicht explizit spezifiziert wird, gelten für den **FORM**-Parameter unterschiedliche Standard-Einstellungen: Ohne explizite Angabe von FORM=... werden sequentielle Files als 'FORMATTED' (formatierte EIN- und Ausgabe), **Direct Access Files** als **'UNFORMATTED'** behandelt.

Wie beim Arbeiten mit sequentiellen Files dient die (vom Programmierer vergebene) UNIT-Nummer (im Beispiel: 10) zur Identifizierung des Files, zusätzlich muß noch die Nummer des zu lesenden bzw. zu schreibenden Records (INTEGER-Ausdruck, der eine Zahl größer als Null bestimmt) angegeben werden, z. B.:

WRITE (10 , REC = 7 , ERR = 500) (A (I) , I = 1 , 12)

schreibt 12 Werte des Arrays A in den "Record 7" der "Unit 10" (Direct Access File "wertetab.dat"). Der ERR-Parameter hat die gleiche Funktion wie bei sequentiellen Files (Sprungmarke als Ziel bei einem aufgetretenen Fehler). Man beachte, daß ein Record immer nur vom Record-Beginn an beschrieben wird, nicht beschriebene Teile eines Records bleiben undefiniert (wenn A zum Beispiel ein DOUBLE-RECISION-Array ist, werden mit der Beispiel-Anweisung die ersten 96 Byte des Records beschrieben). Mit der Anweisung

READ (10 , REC = 7 , ERR = 500) C , D , E

werden die ersten drei Werte aus diesem Record gelesen. Da das Schreiben und Lesen unformatiert ausgeführt wird, ist es im allgemeinen nur sinnvoll, wenn die geschriebenen und die gelesenen Größen den gleichen Datentyp haben. Auch beim Lesen wird mit einer READ-Anweisung immer am Record-Anfang begonnen: Wenn nur der dritte Wert interessiert, müssen die beiden ersten Werte (mit der gleichen READ-Anweisung) vorher "weggelesen" werden (innerhalb des Records gelten "sequentielle Regeln").

Aufgabe 3.1 Man schreibe ein Programm, das für eine Datei **STUDENT.DAT** in einer Schleife jeweils den Namen (maximal 15 Zeichen), den Vornamen (maximal 15 Zeichen), Geburtsjahr (INTEGER) und Matrikel-Nummer (maximal 10 Zeichen) einliest und in einem Record (Record-Länge: 120 Byte) eines Direct Access Files wie folgt ablegt:

Nummer **NEXTRC** des nächsten beschriebenen Records	(INTEGER: 4 Byte),
Matrikel-Nummer	(String: 10 Byte)
Name des Studenten	(String: 15 Byte),
Vorname des Studenten	(String: 15 Byte),
Geburtsjahr	(INTEGER: 4 Byte).

Diese Eintragung benötigt insgesamt nur 48 Byte, der restliche Platz im Record bleibt vorläufig ungenutzt. Im letzten beschriebenen Record wird für NEXTRC eine Null eingetragen. Dies dient zunächst zur Kennzeichnung des "Listen-Endes", weitere Möglichkeiten, die sich mit einer solchen "Verkettung" der einzelnen Eintragungen ergeben, werden im Abschnitt 3.1.5 besprochen.

3.1.2 Zugriff auf Direct Access Files in Netzwerken

Ein wichtiges Problem des **Zugriffs auf eine Datei in einem Netzwerk** kann mit Direct Access Files besonders elegant gelöst werden. Wenn mehrere Anwender gleichzeitig auf die gleiche Datei zugreifen, so ist dies bei nur "lesendem Zugriff" im allgemeinen unproblematisch. Gleichzeitiger "schreibender Zugriff" auf eine Datei von mehreren Anwendern führt zu Konflikten und wird im allgemeinen vom Netzbetriebssystem nicht zugelassen (und führt zu einer Fehlermeldung).

Bei Direct Access Files können **einzelne Records** vor dem Zugriff anderer Benutzer geschützt werden (**LOCKING**). So kann ein Programm immer gerade einen Record (z. B. während eines schreibenden Zugriffs) vor dem Zugriff anderer Benutzer schützen. Die Zugriffsmöglichkeit auf alle übrigen Records des Files bleibt erhalten.

Diese Möglichkeit ist in Standard-FORTRAN-77 nicht vorgesehen, MS-FORTRAN bietet ab Version 5.0 den Befehl **LOCKING** dafür an, mit dem eine beliebige Anzahl aufeinanderfolgender Records vor dem Zugriff anderer Benutzer geschützt werden, Beispiel:

LOCKING (10 , LOCKMODE='LOCK' , REC=3 , RECORDS=2)

schützt 2 Records (die Records mit den Nummern 3 und 4) vor lesendem und schreibendem Zugriff anderer Benutzer. Das Programm wartet, wenn einer der zu "lockenden" Records gerade von einem anderen Programm mit "LOCKING" verschlossen wurde. Bei anderen "Lockmodes" wird im Konfliktfall ein Fehlercode erzeugt (und nicht gewartet), so daß der Programmierer einen Hinweis auf den Konflikt ausgeben (und das gewünschte weitere Verhalten erfragen) kann. Der Befehl

LOCKING (10 , LOCKMODE='UNLCK' , REC=3 , RECORDS=2)

gibt die gesperrten Records wieder frei.

Die LOCKING-Anweisung kann nur für vorher geöffnete Files verwendet werden (sonst gäbe es ja keine gültige UNIT-Nummer, im Beispiel: 10). Da im Regelfall ein File nicht von zwei Programmen geöffnet werden kann (die OPEN-Anweisung des zweiten Programms würde einen Fehlercode erzeugen), ist die LOCKING-Anweisung nur sinnvoll in Verbindung mit dem **SHARE-Parameter der OPEN-Anweisung** (nicht Standard-FORTRAN-77):

OPEN (10 , ... , SHARE='DENYNONE' , ...)

gestattet nach dem Öffnen noch Lese- und Schreibzugriff anderer Programme.

Hinweis: Unter MS-DOS setzt die Verwendung der SHARE-Option (in der OPEN-Anweisung) und des LOCKING-Befehls die Verwendung des DOS-SHARE-Programms (SHARE.COM oder SHARE.EXE) voraus, was besonders beim Testen von Programmen auf Einzelplatz-PCs zu beachten ist, wo die Verwendung von SHARE meistens wenig sinnvoll ist. In Netzwerken kümmert sich das Netzwerk-Betriebssystem ohnehin um das "SHARE-Problem".

3.1.3 Suchen in Direct Access Files

Suchstrategien, die das gezielte Auffinden bestimmter Informationen in sehr großen Datenbeständen auf effektive Weise ermöglichen, sind zu einem Spezialgebiet der Informatik geworden. Hier sollen am Beispiel des Suchens in Direct Access Files wenigstens einige Aspekte dieser Problematik verdeutlicht werden.

> Der direkte Zugriff auf eine Information ist stets nur mit Kenntnis der Adresse (bei Direct Access Files: Record-Nummer) möglich.

Gesucht werden kann immer nur mit Hilfe eines **Schlüssels**, dem eindeutig eine numerische Adresse (Record-Nummer) zuzuordnen sein muß. Als Schlüssel können Namen (z. B. von Personen), Inventar-Nummern, Titel (z. B. von Büchern), ganz allgemein beliebige alphanumerische Begriffe dienen. Folgende Suchstrategien sind möglich:

- **Sequentielles Suchen** durch die gesamte Datei bis zu dem Record, der zum Suchschlüssel paßt, ist die primitivste (aber immer mögliche) und besonders aufwendige Suchstrategie (allerdings besonders einfach zu implementieren). Bei **N** gespeicherten Records müssen bei einer Suche im Mittel **N/2** Records gelesen werden.
- Die **Berechnung der Record-Nummer** aus dem Schlüssel ist im allgemeinen eine besonders schnelle Variante, wenn sie mit vertretbarem Aufwand realisierbar ist. Wenn z. B. die Inventarnummern in einer Bestands-Datei mit den Record-Nummern auf sinnvolle Weise korrespondieren (denkbar ist schließlich sogar das Aufnehmen der Record-Nummer in die Inventarnummer), so kann direkt auf den gewünschten Record zugegriffen werden.

  Die Schnelligkeit des Zugriffs bei dieser Variante hat zu zahlreichen Vorschlägen geführt, wie dieses Verfahren auf beliebige alphanumerische Schlüssel auszudehnen

ist, zum Beispiel: Aus den ASCII-Nummern der Zeichen des Schlüssels wird eine Zahl gebildet, der Divisionsrest, der sich bei Division durch eine vorgegebene Zahl **D** ergibt, wird als Record-Nummer verwendet (kann maximal **D − 1** sein). Da natürlich unterschiedliche Schlüssel bei diesem Verfahren auf gleiche Record-Nummern führen können, muß dieser "Kollisions-Fall" gesondert behandelt werden. Diese als "Hash-Verfahren" bezeichnete Strategie hat noch eine Reihe anderer Nachteile (unrationeller Umgang mit Speicherplatz, Sortieren der Informationen nach vorzugebenden Kriterien ist aufwendig, ...) und wird hier nicht weiter betrachtet.

- Die wohl am häufigsten verwendete Unterstützung des Suchens ist das Anlegen einer Zuordnungstabelle (**Index**), die den Schlüsseln die korrespondierenden Record-Nummern zuordnet. Der Index kann in einer separaten Datei ("Indexdatei") oder in einem Record der zu durchsuchenden Datei selbst untergebracht werden. Typisch ist es, den ersten Record eines Direct Access Files als Index zu verwenden, in dem die Schlüssel und die zugehörigen Record-Nummern gespeichert werden. Wenn der Platz in diesem Record nicht ausreicht, können jederzeit weitere Records für den Index genutzt werden, indem z. B. das letzte "Schlüsselfeld" nicht mit einer Index-Information, sondern dem Verweis (Record-Nummer) auf den Fortsetzungs-Record des Indexes belegt wird (und mit einer **0**, wenn es der letzte "Index-Record" ist).

Aufgabe 3.2 Das Programm, das zur Aufgabe 3.1 geschrieben wurde, ist folgendermaßen zu modifizieren:

- Der 1. Record der Datei ist als "Index-Record" einzurichten, der zu jeder Eintragung eines Studenten die Matrikel-Nummer und die Record-Nummer des Records enthält, in dem die Information gespeichert ist.
- Das Programm soll mit einem Menü starten, das mindestens die Angebote "Eintragen eines Studenten", "Suchen nach einer Eintragung" und "Ende des Programms" enthält. Die "Suche nach einer Eintragung" soll mit der Matrikel-Nummer als Suchschlüssel realisiert werden, wobei der "Index-Record" zu nutzen ist.

3.1.4 Physisch sortierter Index, binäres Suchen

Wenn die Eintragungen der Schlüssel in einen Index in der Reihenfolge erfolgt, in der sie erfaßt werden, spricht man von einem **unsortierten Index**. Das Suchen nach einem bestimmten Schlüssel in einem unsortierten Index ist nur sinnvoll auf sequentielle Weise (andere Suchstrategien sind denkbar, aber nicht sinnvoller), bei **N** gespeicherten Schlüsseln muß bei einer Suche im Mittel mit **N/2** Schlüsseln verglichen werden.

Eine Beschleunigung des Suchprozesses ist möglich, wenn die Index-Einträge auf geeignete Weise sortiert werden. Bei dem Beispiel der Aufgabe der 3.2 könnte z. B. bei jeder Eintragung der gesamte Index so umsortiert werden, daß die Matrikel-Nummern in lexikographischer Anordnung im Index stehen (**physisch sortierter Index**). Der Mehraufwand, der beim Eintragen eines Studenten betrieben werden muß, zahlt sich beim Suchen aus: Es kann die (speziell für sehr große Datenbestände effektive) Strategie **"Binäres Suchen"** angewendet werden.

Beim **binären Suchen** wird der gesuchte Schlüssel zunächst in der Mitte des Indexes vermutet (empfehlenswert ist, die Anzahl aller Indexeintragungen gesondert zu speichern). Beim Vergleich mit dem Schlüssel in der Mitte des **sortierten** Indexes stellt sich heraus, ob der gesuchte Schlüssel in der ersten oder zweiten Hälfte liegt (wenn er nicht zufällig auf Anhieb getroffen wurde). Dann wird der gesuchte Schlüssel wieder in der Mittel des halbierten Indexes gesucht usw.

Auf diese Weise kann man mit **N** Vergleichen einen Datenbestand von $\mathbf{2^N - 1}$ Eintragungen durchsuchen (man beachte, daß immer auch der Vergleich mit der letzten Eintragung ausgeführt werden muß, weil das Suchergebnis auch "Nicht gefunden" lauten kann), Beispiel: Mit maximal **20** Vergleichen wird beim binären Suchen ein Index mit **1048575** Eintragungen komplett durchsucht, wofür beim sequentiellen Suchen im Mittel **524288** Vergleichsoperationen erforderlich wären.

3.1.5 Logisches Sortieren, gekettete Listen

Das im Abschnitt 3.1.4 am Beispiel des Sortierens eines Indexes erläuterte "physische Sortieren" kann sehr aufwendig sein (im Mittel muß die Hälfte aller vorhandenen Eintragungen bei einem neuen Eintrag "bewegt werden"). Dies kann durch sogenanntes **"Logisches Sortieren"** vermieden werden, bei dem ein neues Element physisch auf irgendeinen freien Platz (z. B. an das Ende aller Eintragungen) gestellt wird, um danach "logisch einsortiert" zu werden. Dazu wird jedem Element die Adresse seines logisch nachfolgenden Elementes beigegeben, was beim Einfügen eines neuen Elementes in die logische **Kette** nur zwei Operationen erfordert. Dies soll am Beispiel erläutert werden. Die Eintragungen der Namen in einer Datei seien folgendermaßen "logisch sortiert":

Record-Nummer	Eintragung	Pointer	
3	**Staffenski**	**2**	**ANCHOR = 5**
2	**Stavros**	**0**	
5	**Brabandt**	**9**	
6	**Krause**	**7**	
7	**Scholz**	**3**	
9	**Brandt**	**6**	

Die Eintragungen mögen auf irgendeine Weise völlig unsortiert in die Records gekommen sein, deren Nummer in der linken Spalte angegeben ist. Die Sortier-Information besteht nun aus einem (gesondert zu verwaltenden) **Anker** (**"ANCHOR"**) und den zu jeder Eintragung gehörenden **Zeigern** (**"POINTER"**). Der Anker verweist ("pointert") auf das erste Element der Kette (Record-Nummer der Eintragung "Brabandt"), der Zeiger dieser Eintragung auf den Nachfolger (Record-Nummer der Eintragung "Brandt") usw. Der Zeiger **0** ("Null-Pointer") der Eintragung "Stavros" schließlich zeigt das Ende der Kette an. Man erkennt, daß dies eine "logische" alphabetische Sortierung ist.

Wenn nun eine Eintragung "Flügge" hinzugefügt werden soll, sind folgende Aktionen zu realisieren:

- Die Eintragung wird in irgendeinen freien Record geschrieben (z. B. Record **4**).
- Die Eintragung "Flügge" muß logisch nach der Eintragung "Brandt" und vor der Eintragung "Krause" liegen. Dies wird erreicht, indem der Zeiger der Eintragung "Brandt", der auf die Nachfolge-Eintragung "Krause" zeigt, geändert wird (aus der **6** wird eine **4**, Eintragung "Brandt" zeigt nun auf "Flügge" als Nachfolge-Eintragung), und in der neuen Eintragung "Flügge" wird der Zeiger **6** eingetragen (Eintragung "Flügge" zeigt auf Nachfolge-Eintragung "Krause").

Die Eintragungen mit der geänderten "Verpointerung" haben anschließend folgendes Aussehen:

Record-Nummer	Eintragung	Pointer	
3	**Staffenski**	**2**	**ANCHOR = 5**
2	**Stavros**	**0**	
5	**Brabandt**	**9**	
6	**Krause**	**7**	
7	**Scholz**	**3**	
9	**Brandt**	**4**	
4	**Flügge**	**6**	

Aufgabe 3.3 Welche Änderungen würden sich für das behandelte Beispiel ergeben, wenn die zusätzlichen Eintragungen "Albrecht", "Rühmkorf" und "Wott" eingefügt werden und die Eintragung "Scholz" gelöscht wird?

Der Vorteil der Ergänzung der Datei mit minimalem Aufwand bei gleichzeitiger Erhaltung der Sortierung muß mit einem Nachteil bezahlt werden: **Logisch sortierte Eintragungen können nur sequentiell durchsucht werden.**

Man muß also immer einen geeigneten Kompromiß finden zwischen dem Aufwand, der bei einer Änderung der Eintragungen betrieben werden muß, und der Geschwindigkeit beim Durchsuchen nach einem speziellen Eintrag. Logisches Sortieren ist für das Festlegen einer gewünschten Reihenfolge der Records eines Direct Access Files häufig sehr nützlich. Gleichzeitig kann der Index-Record physisch sortiert sein.

Aufgabe 3.4 Das Programm, das zur Aufgabe 3.2 geschrieben wurde, ist unter Beibehaltung der physischen Sortierung des Index-Records folgendermaßen zu modifizieren:

- Bei jeder neuen Eintragung werden die bereits vorgesehenen Parameter **NEXTRC** so modifiziert, daß damit eine logische Sortierung der Namen in alphabetischer Reihenfolge gegeben ist.
- Es ist ein zusätzliches Angebot "Ausgabe aller Eintragungen" (Bildschirm) zu realisieren, das alle Eintragungen in der Reihenfolge der logischen Sortierung ausgibt.

3.2 Datenbanksysteme

Die Probleme mit der Speicherung von großen Datenbeständen und dem gezielten Zugriff auf spezielle Informationen in Wirtschaft, Verwaltung, Wissenschaft und Technik haben sehr früh zur Entwicklung von Datenbanksystemen geführt. Dabei tauchte sehr schnell das Problem auf, daß unterschiedliche Anforderungen beim Zugriff auf Datenbestände zu speziellen Lösungen führten, ein Problem, das bis heute kaum zufriedenstellend gelöst ist.

Dies soll am Beispiel der Datenhaltung in einem Unternehmen verdeutlicht werden, in dem unter anderem folgende Daten anfallen:

- Die **betriebswirtschaftlichen Daten**, die im gesamten Bereich von der Auftragsannahme bis zum Versand anfallen (Kalkulationsdaten, Daten der Material- und Kapazitätsbewirtschaftung, Daten der Fertigungssteuerung, ...) hatten aus historischen Gründen ("klassische Buchführung") vielfach schon weitgehend einheitliche Strukturen, die einer effektiven Verwaltung im Computer entgegenkamen. Diese Daten, die unter dem Begriff **PPS** ("Produktions-Planungs- und Steuerungssysteme") zusammengefaßt werden können, werden seit vielen Jahren in vielen Firmen in geeigneten Datenbanken verwaltet.
- Die **Daten**, die **im CAD-CAM-Bereich** ("Computer Aided Design", "Computer Aided Manufacturing") anfallen, sind wesentlich heterogener strukturiert. Innerhalb der technologischen Datenkette (z. B. von der Konstruktion und der Berechnung über Arbeitsplanung, NC-Programmierung, Transport- und Montagesteuerung bis zur Qualitätssicherung) werden große Datenmengen immer wieder benötigt. Als hervorstechendes Beispiel mögen die Geometriedaten dienen, die von fast allen Gliedern dieser Kette verarbeitet werden müssen. Andererseits ist zum Teil ein besonders effektiver Zugriff auf die Daten erforderlich, der spezielle Datenstrukturen nahelegt (ausgelegt z. B. auf ein schnelles Anwortverhalten eines CAD-Systems), was natürlich die allgemeine Nutzung außerordentlich erschweren kann.

Das Problem, das schon innerhalb einer typischen Datenkette besteht, ist bei dem naheliegenden Wunsch einer einheitlichen Datenbasis in einem Unternehmen natürlich noch gravierender. Während zum Beispiel die Geometrie-Daten im PPS-Bereich nur eine untergeordnete Rolle spielen, sind die Informationen aus Stücklisten, die im CAD-CAM-Bereich erzeugt werden, grundlegende Daten für die Materialwirtschaft. Auch die technologischen Daten werden im PPS-Bereich (Kapazitätsplanung, Kalkulation, ...) benötigt.

Der Idealzustand einer einheitlichen betrieblichen Datenbasis ist gegenwärtig noch nicht realisierbar und möglicherweise nur mit der Einschränkung erreichbar, daß aus dieser (zentralen) Datenbasis die Datenformate für die speziellen Anforderungen abgeleitet werden. Da aber dezentral ständig neue Daten erzeugt werden, ist ein ständiger Abgleich der zentralen Datenbasis mit den dezentralen Daten erforderlich, um Konsistenz im Datenbestand zu garantieren.

Ein Kompromiß auf dem Weg zum (sicher kaum erreichbaren) Idealzustand ist eine exakte Definition der zentralen Datenbasis, in der sich nur die Daten befinden müssen, auf die von verschiedenen Stellen zugegriffen werden muß. Zu dieser Definition gehören Festlegungen, wer welche Daten verändern und wer nur lesend (und wer aus Sicherheitsgründen gar nicht) zugreifen darf.

Ein **Datenbanksystem** besteht aus einer **Datenbank** und einem **Datenbankverwaltungssystem**.

Eine **Datenbank** ist eine systematische Sammlung von Daten (allgemein wird die Summe der Files, die das Modell der Datenstruktur und die Daten selbst enthalten, als die "Datenbank" bezeichnet).

Ein **Datenbankverwaltungssystem** (**DBMS** - "Data Base Management System") besteht aus einer Vielzahl von Werkzeugen, mit denen das Datenmodell definiert und die Daten eingegeben, geändert, verwaltet, geschützt, gesichert, strukturiert, abgefragt (nach speziellen Daten durchsucht) und ausgegeben werden können.

3.2.1 Grundbegriffe, Datenbank-Strukturen

In einer Datenbank werden Informationen über bestimmte **Objekte** (**"Entities"**, auch die deutsche Übersetzung "Entität" ist gebräuchlich) gespeichert, indem jedem Objekt **Attribute** zugeordnet werden (Beispiel: Das "Objekt" Student wird mit den "Attributen" Matrikel-Nummer, Name, Vorname, Geburtsdatum, usw. gespeichert). Zwischen einzelnen Objekten können **Beziehungen** (**"Relationships"**) bestehen, Beispiel einer "Entity-Relationship"-Aussage: Student (Objekt) besucht (Beziehung) Vorlesung (Objekt, das z. B. durch die Attribute Raum, Name der Lehrveranstaltung, Professor usw. charakterisiert wird).

Nach der Art, wie die Objekte mit ihren Attributen und den zwischen den Objekten bestehenden Beziehungen in einer Datenbank gespeichert bzw. mit dem DBMS aus der Datenbank herausgezogen werden können, werden die Datenbanksysteme klassifiziert:

- Das **hierarchische Datenbankmodell** kann nur sogenannte "1:n-Beziehungen" abbilden (zu einer Semestergruppe gehören mehrere Studenten, aber jeder Student darf nur einer Semestergruppe angehören). Das File-System in MS-DOS ist ein Modell für eine hierarchische Struktur dieser Art: Zu jedem Verzeichnis gehören beliebig viele Dateien und Unter-Verzeichnisse, aber jede Datei und jedes Unter-Verzeichnis gehören nur genau einem übergeordneten Verzeichnis an. Es gibt ein "Wurzel-Verzeichnis" (**"Root"**).

  Man kann durchaus das File-System auf einer Festplatte als Datenbank ansehen und die für seine Verwaltung verfügbaren Befehle EDIT, TYPE, FIND, ... als (wenn auch primitives) Datenbankverwaltungssystem.

- **Datenbanksysteme mit einer Netzwerkstruktur** (nicht zu verwechseln mit einer Datenbank, auf die über ein Rechnernetz zugegriffen wird) können auch "n:m-Beziehungen" abbilden (jeder Student hört mehrere Vorlesungen, alle Vorlesungen werden von mehreren Studenten besucht). Eine gute Modellvorstellung für eine Netzwerkstruktur sind die Beziehungen zwischen Haupt- und Unterprogrammen in einer höheren Programmiersprache: Jedes Unterprogramm kann auf beliebiger Hierarchieebene beliebig oft aufgerufen werden und selbst beliebige andere Unterprogramme aufrufen. In einigen Programmiersprachen (nicht in FORTRAN 77) sind sogar rekur-

sive Aufrufe möglich (Unterprogramm ruft sich direkt oder über ein oder mehrere Unterprogramme selbst auf).

Die gewaltigen Strukturierungsmöglichkeiten von Netzwerkstrukturen verlangen für ein effektives Arbeiten die Strukturierung durch Spezialisten.

- **Relationale Datenbanken** nehmen seit Mitte der achtziger Jahre eindeutig eine Sonderstellung ein. Sie arbeiten mit sehr einfachen und für den Benutzer damit überschaubaren Datenstrukturen (Tabellen), und erlangen trotzdem eine hohe Flexibilität dadurch, daß die Verknüpfung der Objekte durch Relationen erst vom Datenbankverwaltungssystem (bei der Abfrage) ausgewertet wird. Erkauft werden diese Vorteile durch einen erheblich größeren Aufwand bei der Abfrage (schlechteres "Antwortverhalten"). Immer schneller werdende Computer (vor allem auch immer kürzere Zugriffszeiten auf die Massenspeicher) beheben diesen Mangel mehr und mehr, allerdings immer im Wettlauf mit ständig wachsenden Datenmengen.

Im folgenden wird nur das Arbeiten mit relationalen Datenbanken behandelt.

3.2.2 Relationale Datenbanken, Grundlagen

In **relationalen Datenbanken** werden alle Daten in (unter Umständen sehr vielen) **zweidimensionalen Tabellen** erfaßt. In jeder **Tabellenzeile** wird ein **Objekt** beschrieben, die **Spalten** enthalten die **Attribute**.

Um den Zugriff auf bestimmte Attribute zu ermöglichen, enthält jedes Attribut einen eindeutigen Namen ("Spalten-Überschrift"), siehe folgendes Beispiel:

Tabelle Professoren

Kürzel	Name	Vorname	Raum	Telefon
BNR	Bannier	Ulrich	112	3028
DKT	Dankert	Helga	226f	2571
DNK	Dankert	Jürgen	226f	2571
HDN	Haidan	Rainer	226b	4341
WBE	Wiebe	Erhard	129	3001

Um einen **eindeutigen Zugriff auf ein Objekt** (Tabellenzeile) zu ermöglichen, muß mindestens eine Spalte ein **Schlüsselattribut (Primärschlüssel)** enthalten, mit dem die Zeilen der Tabelle eindeutig voneinander zu unterscheiden sind. Gegebenenfalls muß ein Schlüsselattribut zusätzlich "erfunden" werden, wenn keines der Attribute die Bedingung der Eindeutigkeit erfüllt, im einfachsten Fall können die Zeilen einfach durchnumeriert werden. In der "Tabelle Professoren" kann das eindeutige "Kürzel" als Primärschlüssel verwendet werden (das Attribut "Name" scheidet dafür wegen der "Häufung von Dankerts" aus).

Das einfache Beispiel zeigt aber auch schon ein typisches Problem: DNK ist mehr im Rechenzentrum (Raum 338, Telefon 3075) zu finden als in dem angegebenen Raum, WBE ist besser über das Sekretariat (Raum 128, Telefon 3002) zu erreichen. Man könnte diese Angaben zusätzlich in die Spalten der Tabelle hineinnehmen (nicht so gut) oder zusätzliche Spalten einrichten (ganz schlecht). Attribute, die sich wiederholen (man denke z. B. an Attribute wie "Vorlesungen", "Betreute Diplomanden"), sollten als **Wiederholungsdaten in eine gesonderte Tabelle** ausgelagert werden. In erster Linie wird damit das Problem umgangen, beim Entwurf der Datenbank bereits über die maximal mögliche Anzahl solcher Daten entscheiden zu müssen.

Für das behandelte Beispiel wird nachfolgend die Aufteilung der Information auf zwei Tabellen gezeigt:

Tabelle Professoren

Kürzel	Name	Vorname
BNR	Bannier	Ulrich
DKT	Dankert	Helga
DNK	Dankert	Jürgen
HDN	Haidan	Rainer
WBE	Wiebe	Erhard

Tabelle Räume

Kürzel	Raum	Telefon
BNR	112	3028
DKT	226f	2571
DNK	226f	2571
DNK	338	3075
HDN	226b	4341
WBE	129	3001
WBE	128	3002

In der "Tabelle Räume" stellt das "Kürzel" einen sogenannten **Fremdschlüssel** dar, der den Bezug zu den Objekten in der "Tabelle Professoren" herstellt (Fremdschlüssel werden deshalb auch als **Bezugsschlüssel** bezeichnet). Im Gegensatz zu den Primärschlüsseln **dürfen Fremdschlüssel in der Tabelle mehrfach auftauchen**.

Ein Problem ist damit gelöst: Die "Tabelle Räume" kann gegebenenfalls um eine beliebige Anzahl von Zeilen erweitert werden, wenn weitere Räume (z. B. ein Labor), in denen die "Objekte aus der Tabelle Professoren" gesucht werden können, aufgenommen werden sollen.

Ein neues Problem wird sichtbar: Zweimal taucht in der "Tabelle Räume" der Raum 226f mit der zugehörigen Telefonnummer 2571 auf. Dies ist nicht nur eine Verschwendung von Speicherplatz, sondern vor allen Dingen ein Problem für die Konsistenz des Datenbestandes, wenn Änderungen vorgenommen werden müssen. Natürlich ist ein Datenbestand dann am einfachsten zu ändern, wenn eine Information (hier: "Das Telefon im Raum 226f hat die Nummer 2571") nur an einer Stelle auftaucht. Abhilfe schafft auch hier das Ausgliedern der Telefonnummern in eine spezielle Tabelle, die Räume und Telefonnummern verknüpft.

Das kleine Beispiel macht deutlich, daß der "Entwurf einer Datenbank"

- einigen allgemein zu formulierenden Grundsätzen genügen sollte (siehe dazu den folgenden Abschnitt),

- nur möglich ist, wenn man gute Kenntnisse von der Art der zu speichernden Daten und den Beziehungen dieser Daten untereinander hat.

Der Entwurf (die "Definition") einer Datenbank wird vom Datenbankverwaltungssystem unterstützt, ein guter Entwurf erfordert jedoch immer "Insider-Kenntnisse" über die zu verwaltenden Daten. So ist in dem besprochenen Beispiel eine Tabelle, die Räume und Telefonnummern verknüpft, für die Kleinbüro-Landschaft des Fachbereichs Maschinenbau der FH Hamburg sinnvoll, würde für eine Firma, deren Mitarbeiter in einem Großraumbüro sitzen, allerdings wenig Sinn machen.

3.2.3 Empfehlungen zum Datenbank-Entwurf, Normalisierung

Für das Entwerfen von Datenbanken ist ein schrittweises Vorgehen in folgenden Etappen sinnvoll:

- Definition des **Informationsmodells**, das von der Datenbank repräsentiert werden soll (Welche Informationen müssen gespeichert werden? Welche Beziehungen bestehen zwischen den zu speichernden Informationen?). Dazu gehört auch eine genaue Definition aller verwendeten Bezeichnungen ("Data Dictionary").
- Entwurf der Datenstruktur, für relationale Datenbanken wird dies durch die **Definition der Tabellenstruktur** realisiert. Ein wichtiger Schritt für einen "sauberen Entwurf" ist die (noch zu besprechende) **Normalisierung** der Tabellenstruktur. Man beachte: Die Aussagen über ein Objekt stehen in einer Zeile. Zeilen (für weitere Objekte) können (jederzeit) problemlos hinzugefügt werden. Die Spaltenanzahl sollte mit dem Entwurf endgültig festliegen (eine nachträglich hinzugefügte Spalte würde für alle bereits eingetragenen Objekte zunächst keine Information enthalten).
- Definition der **Aufnahmestruktur**, die für jedes Attribut in jeder Tabelle die Datentypen, maximale Größe der Datenfelder, ... festlegt, so daß damit die Aufnahme der Daten mit Hilfe des DBMS möglich ist.

Folgende **Grundsätze** sollten dabei unbedingt beachtet werden:

- Die aufzunehmenden Daten sollten **keine Redundanz** aufweisen. Das Beispiel im vorigen Abschnitt zeigte mit der "Tabelle Räume" ein typisches Beispiel (mehrfach war die einzige Telefonnummer eines Raumes eingetragen). Redundanzfreiheit ist die wichtigste (leider auch am schwierigsten zu realisierende) Forderung an einen Datenbankentwurf. Die meisten Widersprüche in Datenbeständen resultieren aus Nichtbeachtung der Forderung nach Redundanzfreiheit. **Jede Angabe über eine Person oder eine Sache sollte nur einmal in der Datenbank gespeichert sein.**
- **Eindeutigkeit:** Ein Objekt in einer Datenbak kann nur **eindeutig** identifiziert werden, wenn der Zugang über (gegebenenfalls mehrere) **Schlüsselattribute** nur jeweils genau auf ein bestimmtes Objekt führt (vgl. die **zusammengesetzten Schlüssel** im nachfolgenden Beispiel).
- **Unveränderliche Daten** sollten **bevorzugt** aufgenommen werden (z. B. das Geburtsdatum einer Person und nicht das Alter), diese Forderung ist natürlich häufig nicht erfüllbar.

Die Realisierung der formulierten Empfehlungen und die Beschreibung der noch nicht behandelten Begriffe werden an nachfolgendem Beispiel demonstriert:

Beispiel: Professor DNK (der aus der Tabelle des vorigen Abschnitts) möchte sich eine Datenbank **STUDIES** anlegen, in der er für alle Fächer, in denen Studenten bei ihm Prüfungen ablegen oder Leistungsnachweise erbringen, alle Daten, die auf den Leistungsscheinen eingetragen werden, und weitere "archivierungswürdigen" Informationen ablegt (das widerspricht übrigens nicht dem Datenschutz, was Herrn DNK aber ohnehin nicht sonderlich interessiert).

Er definiert also zunächst das **Informationsmodell**. Gespeichert werden sollen **Matrikel-Nummer**, **Name**, **Vorname**, **Geburts-Datum** des Studenten, das **Fach**, das **Semester**, das **Prüfungs-Datum**, die **Zensur**. Da auf den Leistungsscheinen zusätzlich die **Zensur in Worten** angegeben werden muß, soll auch diese Information gespeichert werden. Einige Studenten benötigen ihre Prüfungsergebnisse oft sehr schnell und geben deshalb Herrn DNK ihre Adresse, so daß er ihnen den Leistungsschein sofort nach Korrektur der Klausur zuschicken kann, und er entschließt sich, auch die **Adresse** (soweit bekannt) zu speichern.

Die Begriffe erscheinen eindeutig. Herr DNK glaubt zunächst, sich ein "Data Dictionary" ersparen zu können (er hat es im Kopf, sollte es aber doch dokumentieren). Zwei Begriffe müssen ohnehin genauer definiert werden: **Semester** soll nicht als Zahl verstanden werden ("Student XY im 12. Semester fällt im Fach TM1 durch"), sondern im Sinne der Bezeichnung auf dem Leistungsschein ("... fällt im Sommersemester 1995 durch"). Bei dem Begriff **Fach** ist zu unterscheiden zwischen Fächern, in denen ein Leistungsnachweis erbracht (und damit eine Zensur vergeben) wird, und Fächern, in denen ein Studiennachweis erbracht (und damit die Bewertung "Bestanden" bzw. "Nicht bestanden" vergeben) wird.

Nach Abwägen alternativer Möglichkeiten entschließt sich Herr DNK, den Begriff **Fach** durch die beiden Begriffe **LN-Fach** (Fach mit Leistungsnachweis) und **SN-Fach** (Fach mit Studiennachweis) zu ersetzen, und definiert folgende **Informationsstruktur**:

Attribut	Bedeutung	Datenfeld
Matrikel-Nummer	Eindeutiges Identifikationsmerkmal	MatNum
Name	Nachname des Studenten	Name
Vorname	Vorname des Studenten	VName
Geburts-Datum	Geburts-Datum des Studenten	GebDat
LN-Fach	Fach, in dem eine Zensur vergeben wird	LNFach
SN-Fach	Fach, in dem Studiennachweis zu erbringen ist	SNFach
Semester	Semester, in dem Leistung erbracht wurde	Semester
Prüfungs-Datum	Datum der Prüfung	PrDat
Zensur	Prüfungszensur (1.0, 1.3, 1.7, 2.0, ... , 5.0)	Zensur
Zensur in Worten	"Sehr gut", "Gut", ..., "Bestanden", ...	ZensWort
Adresse	Adresse des Studenten	Adresse

Mit der so dokumentierten Informationsstruktur sind die Begriffe ausreichend beschrieben, die Festlegung der Bezeichnung für die Datenfelder ist ein (im allgemeinen sinnvoller) Vorgriff auf die Definition der Aufnahmestruktur, weil die Kurzbezeichnungen schon bei der nachfolgenden Definition der Tabellenstruktur verwendet werden können.

Wenn die eingangs formulierten Forderungen beachtet werden sollen, dürfen die in der Informationsstruktur festgelegten Datenfelder nicht einfach in eine Tabelle übernommen werden. Eine sehr gute Hilfe beim Festlegen einer guten Tabellenstruktur ist die Beachtung der Regeln der **Normalisierung** der Tabellen. E. J. Codd, der in der Literatur häufig als der "Erfinder" der relationalen Datenbanken genannt wird, hat drei **Normalformen** definiert, denen die Tabellen eines Datenbankentwurfs entsprechen sollten.

Eine Tabelle befindet sich in der **1. Normalform**, wenn jedes Attribut nur einmal in ihr vorkommt.

Die als Schlüsselattribut zu benutzende Matrikel-Nummer (Primärschlüssel), der Name, der Vorname und das Geburtsdatum könnten also in einer "Tabelle Student" zusammengefaßt werden, die der 1. Normalform genügt. Eigentlich dürfte auch die Adresse in diese Tabelle hineingenommen werden, da sie aber nicht für alle Studenten bekannt ist, entscheidet Herr DNK, die Adressen in einer gesonderten Tabelle zu führen. Man beachte, daß man mit der Entscheidung, den Namen und den Vornamen als Attribute mit nur einem Wert zu führen, gezwungen wird, Platz für solche Wortungetüme wie "Leutheusser-Schnarrenberger" vorzuhalten und bei Personen mit mehreren Vornamen nur einen speichern kann oder mehrere Vornamen wie einen behandeln muß.

Alle Informationen, die die Prüfungen betreffen, sind als **Wiederholungsdaten** in eine gesonderte Tabelle auszulagern. Der Bezug zu den Eintragungen in der "Tabelle Student" wird über die Matrikel-Nummer hergestellt, die in der "Tabelle Zensuren" als Fremdschlüssel fungiert. In dem nachfolgend dargestellten 1. Entwurf der Datenbank fehlen noch die Tabellen für die "SN-Fächer" und die Adressen:

Datenbank STUDIES
(1. Entwurf)

Tabelle **Student**

MatNum	Name	VName	GebDat

Tabelle **Zensuren**

MatNum	LNFach	Semester	PrDat	Zensur	ZensWort

Um in der "Tabelle Zensuren" auf ein spezielles Prüfungsergebnis zugreifen zu können, müssen neben der Matrikel-Nummer des Studenten noch weitere Angaben gemacht werden. Unter der Voraussetzung, daß ein Student in einem Fach nur eine Prüfung pro Semester ablegen kann, könnte ein eindeutiger Zugriff auf ein Prüfungsergebnis über den **zusammengesetzten Schlüssel** "MatNum,LNFach,Semester" erfolgen. Diese drei Attribute werden also zu Schlüsselattributen erklärt.

In der "Tabelle Zensuren" ist das "Attribut Zensur" vom (zusammengesetzten) "Gesamtschlüssel MatNum,LNFach,Semester" abhängig. Da Herr DNK die Prüfung in einem "LNFach" grundsätzlich für alle Studenten am gleichen Tag durchführt (Klausur), ist dagegen das Prüfungs-Datum nur vom "Teilschlüssel LNFach,Semester" abhängig.

> Eine Tabelle befindet sich in der **2. Normalform**, wenn sie sich in der 1. Normalform befindet und außerdem alle Nichtschlüsselattribute ausschließlich vom Gesamtschlüssel (nicht nur von einem Teilschlüssel) abhängen.

Um der Forderung der 2. Normalform gerecht zu werden, wird also das Prüfungs-Datum in eine gesonderte Tabelle ausgelagert, in der auf "PrDat" über den zusammengesetzten Schlüssel "LNFach,Semester" zugegriffen wird.

Das Attribut "ZensWort" in der "Tabelle Zensuren" ist ausschließlich vom Attribut "Zensur" abhängig. Da "Zensur" kein Schlüsselattribut ist, wird damit gegen eine weitere Regel verstoßen:

> Eine Tabelle befindet sich in der **3. Normalform**, wenn sie sich in der 2. Normalform befindet und kein Attribut enthält, das von einem Nichtschlüsselattribut abhängig ist.

Das Attribut "ZensWort" wird also in eine spezielle Tabelle ausgelagert, die das Attribut "Zensur" als Schlüsselattribut enthält.

Man beachte, daß die Forderung der 3. Normalform, die die Erfüllung der beiden anderen Normalformen voraussetzt, auf eine redundanzfreie und damit leicht zu wartende Datenstruktur zielt. Wenn zum Beispiel die Behörde für Wissenschaft plötzlich die Idee hat, daß die Zensur "1" nicht mehr als "Sehr gut", sondern als "Unsozial und elitär" bezeichnet werden soll, so müßte Herr DNK in seiner Datenbank nur eine einzige Eintragung ändern.

Im Gegensatz dazu wird die Frage, **wie** der Zugriff auf die Informationen realisiert werden kann, beim Entwurf einer "sauberen Struktur" nicht gestellt (und damit ausschließlich dem DBMS übetragen). Dies kann sich durchaus mit nicht akzeptablen Antwortzeiten bei der Abfrage der Datenbank rächen.

Aufgabe 3.5 Die Definition der Tabellenstruktur für die "Datenbank STUDIES" ist so zu komplettieren, daß alle Tabellen der Forderung der 3. Normalform entsprechen. Die Tabellen "StNachw" (Studiennachweise für die "SN-Fächer") und "Adresse" (und eventuell weitere Tabellen, wenn dies zur Erfüllung der Forderung der 3. Normalform erforderlich ist) sind zu ergänzen.

3.2.4 Zugriff auf Datenbanken

Der Benutzer korrespondiert mit einer Datenbank entweder interaktiv (z. B. menügeführt in einer Windows-Umgebung oder mit einer Datenbanksprache, deren Befehle sofort abgearbeitet werden) oder über Anwendungsprogramme, bei denen er von der Interaktion mit der Datenbank häufig gar nichts merkt. Auch die Anwendungsprogramme sprechen die Datenbanken in der Regel über Befehle einer speziellen Datenbanksprache an.

Im PC-Bereich war die Firma Borland mit dem Datenbanksystem dBase über eine lange Zeit so eindeutig der Marktführer, daß damit gewisse Standards gesetzt wurden (die Marktführerschaft besteht nicht mehr, Produkte wie Access, FoxPro, Paradox haben wesentlich Anteile gewonnen). Als "xBase-kompatibel" werden Datenbanken (z. B. Clipper, FoxPro, ...) bezeichnet, die das dBase-Datenformat (DBF) verwenden und sich an die Syntax der von dBase verwendeten Sprache anlehnen.

Bei den Datenbanken, die auf Großrechnern oder in "Client-Server-Architekturen" laufen (unterschiedliche "Client"-Programme auf verschiedenen Workstations fordern Dienste von einem Datenbank-Server an), hat sich in den letzten Jahren eindeutig das relationale Modell mit der speziell dafür entwickelten Abfragesprache **SQL** ("Structured Query Language") durchgesetzt (verwendet unter anderem in den sehr weit verbreiteten Systemen ORACLE, INFORMIX, DB/2 von IBM, GUPTA, RDB von DEC, aber auch dBase "versteht" neben der eigenen Sprache SQL). SQL ist seit 1989 genormt (ANSI und OSI) und hat den Charakter einer höheren Programmiersprache.

Der Benutzer kann mit Hilfe der SQL-Befehle mit der Datenbank interaktiv kommunizieren (Befehle werden vom DBMS interpretierend sofort abgearbeitet) oder indirekt über **Kommandodateien** (Zusammenfassung von SQL-Anweisungen ähnlich wie die Zusammenfassung von DOS-Befehlen in Batch-Prozeduren). Schließlich können SQL-Befehle in höhere Programmiersprachen (FORTRAN, Pascal, COBOL, C, ...) eingefügt werden ("Embedded SQL", eingebettet in eine "Wirtssprache"), so daß Anwendungsprogramme entstehen, die direkt mit dem DBMS korrespondieren. Bei der letztgenannten Variante werden die SQL-Anweisungen von sogenannten "Präprozessoren" an die Syntax der Wirtssprache angepaßt, bevor die Programme mit dem entsprechenden Hochsprachen-Compiler übersetzt werden.

Große Datenbanken sind in der Regel so angelegt, daß mehrere Benutzer gleichzeitig auf die Datenbestände zugreifen (und sie verändern) können. Das DBMS muß gewährleisten, daß die dabei auftretenden Konflikte nicht zu Inkonsistenzen in den Dateien führen. Gleichzeitiger schreibender Zugrif auf den gleichen Datensatz (Record) wird im allgemeinen schon vom Netzwerkbetriebssystem nicht zugelassen.

Im praktischen Betrieb können jedoch auch andere kritische Situationen auftreten, für die das DBMS Sicherungen anbieten muß.

Beispiel: Zwei Kunden bestellen gleichzeitig bei verschiedenen Bearbeitern eines Reifenhändlers je 40 Reifen und dazu passende Felgen. Beide Bearbeiter sehen einen Lagerbestand von 60 Reifen und 60 Felgen (lesende Zugriffe auf die Datenbank) und sagen die Lieferungen zu. Bei der Realisierung der Bestellung müssen die Daten verändert werden (schreibende Zugriffe). Der Zugriff auf die Daten wird vom ersten Bearbeiter gesperrt, nach Veränderung der Daten und Aufhebung der Sperre stellt der zweite Bearbeiter

fest, daß die Bestellung mit dem verbliebenen Bestand nicht mehr realisierbar ist. Diese Situation führt jedoch (wegen der Sperre) nicht zu inkonsistenten Datenbeständen.

Es kann jedoch eine noch unglücklichere Situation eintreten: Ein Bearbeiter bucht zuerst die Reifen, der andere (etwa gleichzeitig) zuerst die Felgen aus den Lagerbeständen ab, beides wird wegen der ausreichenden Bestände realisiert, es kommt nicht einmal zu einem Zugriffskonflikt. Bei den Versuchen, jeweils den anderen Teil der Bestellung zu realisieren, wird beiden Bearbeitern (wieder ohne Zugriffskonflikt) ein nicht mehr ausreichender Lagerbestand gemeldet. Da Reifen ohne Felgen (und umgekehrt) für die Kunden uninteressant sind, kann keine der Bestellungen realisiert werden, obwohl für einen Kunden durchaus ausreichnde Lagerbestände vorrätig sind. Eine solche Situation wird in der Informatik als **"Deadlock"** bezeichnet.

Um Situationen wie die in dem Beispiel geschilderte zu vermeiden, muß von einem modernen DBMS erwartet werden, daß mehrere Aktionen zu **Transaktionen** zusammengefaßt werden können und eine "Transaktionssicherung" in dem Sinne möglich ist, daß alle beteiligten Datensätze vorab gesperrt werden und der Zustand vor der Veränderung in einem "Transaktions-Protokoll" gesichert wird. Dann wird entweder die gesamte Transaktion erfolgreich ausgeführt, oder der Zustand, der vor der Transaktion bestand, wird wieder hergestellt.

3.2.5 SQL (Structured Query Language), Einführung

In diesem Abschnitt sollen die wichtigsten Befehle der Datenbanksprache SQL exemplarisch vorgestellt werden:

- Anlegen, Öffnen und Schließen einer Datenbank, Anlegen der Tabellen,
- Daten in die Tabellen eingeben und Daten aus den Tabellen abrufen.

Die Befehle werden nur mit den wichtigsten Optionen beschrieben. Nicht betrachtet werden die Befehle, mit denen Tabellen geändert werden können (Hinzufügen und Löschen von Tabellenspalten), Befehle zur Vergabe und Rücknahme von Zugriffsrechten auf die Daten und spezielle Befehle zur Verwaltung und Wartung (BACKUP, Zugriffsprotokollierung usw.).

SQL-Befehle für das Erzeugen, Öffnen, Schließen und Löschen einer Datenbank:

```
CREATE   DATABASE  dbname  ;
START    DATABASE  dbname  ;
STOP     DATABASE  ;
DROP     DATABASE  dbname  ;
```

Der "Name der Datenbank" **dbname** wird im CREATE-DATABASE-Befehl vergeben und dient später dazu, die Datenbank unter diesem Namen anzusprechen. Die nachfolgend angegebenen Beispiele beziehen sich immer auf das Beispiel aus dem Abschnitt 3.2.3.

Beispiel für das Erzeugen einer Datenbank:

CREATE DATABASE Studies ;

Nach dem Erzeugen einer Datenbank können sofort Tabellen erzeugt werden (START DATABASE ist nur erforderlich, wenn eine existierende Datenbank mit STOP DATABASE geschlossen wurde und erneut geöffnet werden soll).

SQL-Befehl für die Definition einer Tabelle

```
CREATE   TABLE   tabname
      (Attribut  Datentyp  [ , Attribut  Datentyp  ... ] ) ;
```

Hinweis: Die eckigen Klammern in den Syntax-Definitionen deuten an, daß der Inhalt optional ist, die eckigen Klammern selbst gehören nicht zum Befehl. Drei Punkte deuten an, daß ein Teil der Syntax-Definition beliebig oft wiederholt auftreten kann.

Im CREATE-TABLE-Befehl ist **tabname** der Name einer Tabelle, unter der diese von späteren Anweisungen angesprochen werden kann, **Attribut** ist der Name eines Attributs (Spalten-Überschrift). Jedes Attribut hat einen Datentyp, erlaubt sind unter anderem folgende

Datentypen (Auswahl):

INTEGER	-	Ganze Zahlen
DECIMAL(i,n)	-	Dezimalzahl mit insgesamt i Stellen (einschließlich Dezimalpunkt), davon n Nachpunktstellen
DATE	-	Datum, Format entsprechend Landeseinstellung, z. B.: tt.mm.jjjj
CHAR(n)	-	String mit n Zeichen

Die Festlegung der Namen für die Attribute und ihrer Datentypen gehört zur Definition der Aufnahmestruktur (vgl. Abschnitt 3.2.3), Beispiel:

```
CREATE   TABLE   Student
      (MatNum    CHAR(7) ,  Name      CHAR(20) ,
       VName     CHAR(15) ,  GebDat   DATE )  ;
```

(diese Definition nimmt auf bestimmte Werte der Attribute keine Rücksicht, es könnte zum Beipiel für **Name** maximal "Leutheusser-Schnarre" eingetragen werden, geschieht ihr recht).

SQL-Befehl für das Eingeben einer Zeile in eine Tabelle:

```
INSERT  INTO    tabname
        VALUES  ( Wert  [ , Wert  ... ]  ) ;
```

Mit **tabname** wird die Tabelle bestimmt, in die eine Zeile mit den **Werten** für die Attribute eingefügt werden soll, die in der Klammer nach dem Schlüsselwort VALUES stehen.

Beispiel:
```
INSERT INTO    Student
       VALUES  ( '1230815' , 'Korn' , 'Klara' , {14.04.1970} ) ;
```

SQL-Befehle für das Ändern bzw. Löschen von Tabellenzeilen:

```
UPDATE    tabname
      SET   Attribut=Ausdruck [ , Attribut=Ausdruck  ... ]
      WHERE     Bedingung  ;

DELETE    FROM      tabname
      WHERE     Bedingung  ;
```

Für **Ausdruck** und **Bedingung** sind recht komplizierte Konstruktionen möglich. Im einfachsten Fall kann für **Ausdruck** einfach ein Attribut (Spaltenname) oder der Wert eines Attributs und für **Bedingung** der Vergleich zweier Ausdrücke stehen.

Die folgenden Befehle würden in der mit dem oben angegebenen INSERT-Befehl erzeugten Tabellenzeile die Eintragung in der Spalte GebDat ändern bzw. die gesamte Zeile löschen:

```
UPDATE    Student
      SET   GebDat={12.04.70}  WHERE     Name='Korn' ;

DELETE    FROM      Student
      WHERE     MatNum='1230815' ;
```

Man beachte, daß diese beiden Befehle durchaus mehrere Tabellenzeilen betreffen können (alle Zeilen, für die die angegebene Bedingung zutrifft). Mit dem UPDATE-Beispiel-Befehl würde also für alle Studenten in der Tabelle mit dem Namen Korn das Geburtsdatum 12.04.70 eingetragen werden (man sollte also vorzugsweise das Schlüsselattribut verwenden, um eine Zeile zu identifizieren).

Aufgabe 3.6 Man erzeuge eine Datenbank **Studies** und darin eine Tabelle **Student** mit den Attributen, die im Tabellenkopf der nebenstehenden Tabelle angegeben sind.

In die Tabelle **Student** sind die 6 Objekte (mit INSERT und gegebenenfalls UPDATE und DELETE) einzugeben.

MatNum	Name	VName	GebDat
1230815	Korn	Klara	12.04.70
1230816	Cron	Maria	04.05.69
4560815	Urbock	Einbecka	31.07.66
4560816	Christ	William	11.06.68
7890815	Walker	Jonny	12.12.74
7890816	Daniels	Jack	08.11.72

Für das Auswählen und Anzeigen einzelner (oder aller) Attribute aus den Zeilen einer Tabelle steht der außerordentlich mächtige SELECT-Befehl zur Verfügung, der zunächst nur mit stark vereinfachter Syntax angegeben wird:

SQL-Befehl SELECT (stark vereinfacht):

```
SELECT    Ausdruck [ , Ausdruck  ... ]   |   *
      FROM      tabname
      [ WHERE   Bedingung ]  [ ORDER BY   Attribut ]  ;
```

Hinweis: Der vertikale Strich | in der Syntaxdefinition kennzeichnet die Alternative: Nach **SELECT** kann entweder eine Auflistung von Ausdrücken (z. B.: Attribute der Tabelle, die angezeigt werden sollen) oder der Stern ***** stehen (der Stern bedeutet, daß alle Attribute angezeigt werden sollen).

Beispiele: **SELECT * FROM Student ;**

... zeigt die gesamte Tabelle Student mit allen Attributen an (die Reihenfolge der Zeilen entspricht der Eingabe-Reihenfolge).

```
SELECT Name , VName , MatNum   FROM  Student
      WHERE   GebDat < {01.01.70}   ORDER BY  Name  ;
```

... zeigt in einer alphabetisch (nach dem Namen) geordneten Liste Name, Vorname und Matrikel-Nummer aller Studenten an, die vor dem 1.1.1970 geboren wurden.

Aufgabe 3.7 Man probiere die beiden angegebenen Beispiel-Befehle mit der mit Aufgabe 3.6 erzeugten Datenbank aus.

In **Ausdrücken** können

- **Attribute** (Spalten-Überschriften),
- **Konstanten** (**Werte**, die von den Attributen angenommen werden können),
- **Operatoren** (z. B.: + − * /),
- **Klammern**,
- **Funktionen** und
- komplette (in Klammern zu setzende) **SELECT-Anweisungen** vorkommen.

Die Verwendung von **Funktionen** in Ausdrücken ermöglicht außerordentlich flexibel zu formulierende SELECT-Anweisungen, nachfolgend eine kleine Auswahl der verfügbaren Funktionen:

Spaltenfunktionen beziehen sich auf alle Attribute einer Spalte:

MIN (Attribut)	-	Minimalwert aller Attribute der Spalte
MAX (Attribut)	-	Maximalwert aller Attribute der Spalte
AVG (Attribut)	-	Mittelwert aller Attribute der Spalte
SUM (Attribut)	-	Summe aller Attribute der Spalte
COUNT (*)	-	Anzahl der Zeilen einer Tabelle

Die meisten SQL-Implementationen enthalten eine sehr große Anzahl weiterer Funktionen, deren Syntax jedoch systemabhängig ist. Wichtig für die Anwendung sind **mathematische Funktionen** (im allgemeinen in Anlehnung an die höheren Programmiersprachen definiert ,z. B. **INT()** für den Ganzzahlanteil des Arguments, **SQRT()** für die Quadratwurzel), **Zeichenkettenfunktionen** (z. B. **LEN()** für die Länge eines Strings). Sehr nützlich sind die **Datumsfunktionen** (z. B. **DATE()** für das Rechnen mit DATE-Attributen, bei fehlendem Argument wird das aktuelle Datum verwendet).

Aufgabe 3.8 Die mit Aufgabe 3.6 erzeugte Datenbank **Studies** soll durch die nebenstehend gezeigte Tabelle **Zensuren** erweitert werden.

Man erzeuge die Tabelle mit den angegebenen Eintragungen und probiere für beide Tabellen mit unterschiedlichen SELECT-Anweisungen die Benutzung von Ausdrücken unter Einbeziehung von Funktionen.

MatNum	LNFach	Semester	Zensur
1230815	Mathe1	SS92	2,3
1230816	TM2	SS92	4,0
1230815	TM2	SS92	5,0
4560816	TM2	WS92	5,0
1230815	TM2	WS92	3,3
1230816	Mathe2	WS92	1,3
1230815	Mathe2	WS92	5,0
4560816	Mathe2	WS92	3,7

Beispiele für mögliche SELECT-Anweisungen:

```
SELECT   COUNT (*)
       FROM   Student ;
```

... liefert die Anzahl der Eintragungen in der Tabelle **Student**.

```
SELECT   INT ((DATE () − GebDat) / 365)  FROM   Student
       WHERE   MatNum='1230816' ;
```

... liefert das Alter der Studentin Maria Cron.

```
SELECT   AVG (Zensur)   FROM   Zensuren   WHERE   Semester='WS92' ;
```

... liefert den Durchschnitt aller Zensuren der Prüfungen des Wintersemesters 92.

In **Bedingungen** dürfen folgende Vergleichsoperatoren verwendet werden:

< > = <> ("ungleich") <= >=

Die Vergleichsoperatoren dürfen zwischen Ausdrücken stehen, die numerische Ergebnisse oder Zeichenketten repräsentieren ("kleiner als" bedeutet bei Zeichenketten, daß der erste Operand in der alphabetischen Reihenfolge vor dem zweiten Operanden steht) bzw. vom Datentyp DATE sind ("kleiner als" bedeutet "früher als").

Bedingungen können (ähnlich wie in höheren Programmiersprachen) durch **NOT** negiert und durch **OR** bzw. **AND** miteinander verknüpft werden.

Beispiel:
```
SELECT AVG (Zensur) FROM Zensuren
    WHERE (LNFach = 'Mathe1' OR LNFach = 'Mathe2') AND
          Zensur <= 4 ;
```

... liefert den Zensurendurchschnitt aller bestandenen Prüfungen in den Fächern Mathe1 und Mathe2.

3.2.6 Spezielle SQL-SELECT-Optionen und VIEWS

In diesem Abschnitt sollen zunächst einige zusätzliche Optionen des SELECT-Befehls besprochen und an Beispielen demonstriert werden, die die Vielfalt der Möglichkeiten bei der Verwendung dieses Befehls zeigen.

Im **SELECT**-Befehl kann die Auswahl sich auch über mehrere Tabellen erstrecken. In diesem Fall müssen

- in der **FROM**-Klausel die beteiligten Tabellen (durch Komma getrennt) angegeben werden und
- in der **WHERE**-Klausel die Attribute mit den Tabellen, auf die sich beziehen, in der Form **Tabelle.Attribut** spezifiziert werden.

Beispiel:
```
SELECT   Name , VName , LNFach , Zensur
    FROM      Student , Zensuren
    WHERE     Student.MatNum = Zensuren.MatNum AND
              LNFach = 'TM2'
    ORDER BY  Name ;
```

... liefert eine (nach den Namen alphabetisch geordnete) Liste aller TM2-Prüfungsergebnisse.

Der **SELECT**-Befehl (in Klammern gesetzt) kann überall dort stehen, wo ein **Ausdruck** stehen darf (z. B. in Bedingungen), wenn das Ergebnis des SELECT-Befehls einen Datentyp hat, der für den Ausdruck zulässig ist.

Der **SELECT**-Befehl kann im **INSERT-INTO**-Befehl die komplette VALUES-Klausel ersetzen (in diesem Fall ohne umschließende Klammern).

Ein Beispiel für die erste der beiden Aussagen findet sich nachfolgend im Zusammenhang mit der HAVING-Klausel. Für die Demonstration der zweiten Aussage wird zunächst eine neue Tabelle **Erfolg** definiert, die aus den beiden Tabellen **Student** und **Zensuren** erzeugt wird und alle Prüfungsergebnisse mit einer Zensur besser oder gleich **4** enthalten soll, wobei nur **Name**, **LNFach** und **GZens** (auf ganzzahligen Wert gerundete Zensur) gespeichert werden sollen:

```
CREATE   TABLE   Erfolg
        (Name  CHAR (20) , LNFach  CHAR (6) , GZens  INTEGER)  ;
```

... definiert die neue Tabelle.

```
INSERT INTO   Erfolg
        SELECT      Name , LNFach , INT (Zensur + 0.4)
        FROM        Student , Zensuren
        WHERE       Student.MatNum = Zensuren.MatNum  AND
                    Zensur <= 4  ;
```

... erzeugt 5 Zeilen für die Tabelle **Erfolg**. Man beachte, daß man mit dieser Tabelle redundante Daten in die Datenbank einfügt. Da das nicht so gut ist, wird schon hier auf die alternative Möglichkeit der VIEW-Definition hingewiesen (wird später in diesem Abschnitt behandelt).

GROUP-BY- und HAVING-Klauseln im SELECT-Befehl

Die **GROUP-BY**-Klausel (einzufügen **nach FROM**-Klausel und **WHERE**-Klausel und **vor** der **ORDER-BY**-Klausel) mit der Syntax

GROUP BY Attribut [, Attribut ...]

ist erlaubt für Attribute (Spaltennamen), die auch unmittelbar nach der **SELECT**-Klausel genannt wurden, und sinnvoll, wenn Aussagen gewünscht werden, die jeweils alle Zeilen mit dem gleichen Wert dieses Attributs betreffen (z. B. alle Mathe1-Klausuren, alle Mathe2-Klausuren, ...).

Die GROUP-BY-Klausel kann ergänzt werden um die **HAVING**-Klausel, die mit der Syntax

HAVING Bedingung

angefügt werden muß und das GROUP-BY-Ergebnis gegebenenfalls einschränkt.

Beispiele:
```
SELECT    LNFach , AVG (Zensur)
          FROM      Zensuren
          GROUP BY  LNFach  ;
```

... faßt jeweils alle Attribute in der Spalte LNFach mit dem gleichen Wert (Mathe1, Mathe2 bzw. TM2) zu einer Gruppe zusammen, bevor die Durchschnitte der Zensuren für die drei Gruppen berechnet und in drei Zeilen ausgegeben werden.

```
SELECT    LNFach , AVG (Zensur)
     FROM      Zensuren
     GROUP BY  LNFach
     HAVING    AVG (Zensur) <
               (SELECT  AVG (ZENSUR)  FROM  Zensuren)  ;
```

... liefert im Gegensatz zum ersten Beispiel nur zwei Zeilen als Ergebnis ab, weil der Druchschnit aller TM2-Ergebnisse nicht kleiner als der Gesamtdurchschnitt aller Zensuren ist und von der HAVING-Klausel herausgefiltert wird. Der Durchschnitt aller Zensuren wird in einer eigenen SELECT-Anweisung ermittelt: Überall, wo ein Ausdruck stehen darf (hier ist es die rechte Seite einer Bedingung) darf in Klammern eine SELECT-Anweisung stehen.

Mit dem SELECT-Befehl sind unterschiedliche "Sichten" auf den Datenbestand möglich. Da es ganz typische (immer wiederkehrende) Anforderungen gibt, ist es sinnvoll, daß man eine solche "Sicht" definieren und in der Datenbank ablegen kann, um sie immer wieder zu benutzen. Dafür dienen

VIEWS: Mit der Syntax

```
CREATE   VIEW  viewname   [ ( Attribut [ , Attribut ... ] ) ]
                 AS   Select-Anweisung  ;
```

wird eine "Sicht" (VIEW) definiert (nicht ausgeführt), die in der Datenbank gespeichert wird. Der **viewname** kann dann z. B. in der FROM-Klausel der SELECT-Anweisung an Stelle eines Tabellennamens stehen.

Beispiel:
```
CREATE   VIEW    KList    (Name , LNFach , GZens)
       AS  SELECT      Name , LNFach , INT (Zensur + 0.4)
           FROM        Student , Zensuren
           WHERE       Student.MatNum = Zensuren.MatNum  AND
                       Zensur <= 4  ;
```

... definiert eine "Sicht" auf die Daten (mit dem Namen KList), die dem Erzeugen der Tabelle **Erfolg** in einem bereits behandelten Beispiel dieses Abschnitts entspricht. Im Unterschied zum Erzeugen der Tabelle werden aber keine redundanten Daten in die Datenbank eingefügt. Es werden durch CREATE VIEW überhaupt keine Daten erzeugt (abgesehen natürlich vom Speichern der VIEW-Definition).

- Die Attributsliste nach dem Viewnamen kann weggelassen werden, wenn nach AS SELECT auch eine reine Attributsliste folgt, die dann übernommen wird. In dem Beispiel mußte sie angegeben werden, weil nach AS SELECT (nach zwei Attributen) auch ein Ausdruck steht.

Im einfachsten Fall kann nach der VIEW-Definition einfach durch

SELECT * FROM KList ;

eine Sicht auf die Daten erzeugt werden, die der in der VIEW-Definition enthaltenen SELECT-Anweisung entspricht. Es sind jedoch auch alle übrigen Klauseln der SELECT-Anweisungen erlaubt, zum Beispiel:

```
SELECT  Name , GZens  FROM    KList
                      WHERE   LNFach = 'TM2' ;
```

Aufgabe 3.9 Die mit Aufgabe 3.6 erzeugte und mit der Aufgabe 3.8 erweiterte Datenbank soll genutzt werden, um alle Beispiele des Abschnitts 3.2.6 nachzuempfinden (und weitere Varianten der Befehle auszuprobieren).