HANSER

Leseprobe

Lothar Piepmeyer

Grundkurs Datenbanksysteme

Von den Konzepten bis zur Anwendungsentwicklung

ISBN: 978-3-446-42354-1

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42354-1

sowie im Buchhandel.

1

Was sind eigentlich Datenbanken?

Menschen sammeln. Die einen sammeln Briefmarken, die anderen Musik und andere wiederum Geld. Wir alle haben aber die Gemeinsamkeit, mit unseren Sinnesorganen Daten zu sammeln, die wir dann zu Informationen verdichten; aus den Informationen wird Wissen. Wissen, das uns hilft, unsere Welt zu verstehen und zu kontrollieren. Die Grenzen zwischen Daten, Informationen und Wissen sind unscharf und sollen hier nicht weiter abgesteckt werden. Daten, die wir so aufnehmen, lagern wir in unserem Gehirn. Unwichtige Daten vergessen wir, wichtige Daten notieren wir vielleicht, eben um sie nicht zu vergessen. Die Vielzahl von Daten wächst zu einem Datenchaos, das seinerseits kaum beherrschbar ist. Darum haben Menschen auch immer wieder praktische Hilfsmittel erfunden, um Daten zu konservieren, zu strukturieren oder auszuwerten. Bereits vor der Erfindung der Schrift fing das an, ging über Kartei- und Zettelkästen, Registraturen, Tabelliermaschinen hin zu Computern. Im Kontext moderner IT wurde dabei im Laufe der Zeit der Begriff „Datenbank“ (dt. für *database*) geprägt.

Definition: Datenbanken

Eine Datenbank ist eine Sammlung von Daten, die von einem Datenbankmanagementsystem (DBMS) verwaltet wird.

1.1 Konsistenz ist Grundvoraussetzung

Die Daten der Datenbank sind die Grundlage vieler Entscheidungen: Bei Rezeptdatenbanken bestimmen sie die Qualität unserer Mahlzeiten, bei unternehmensweiten Datenbanken hängen teilweise riesige Investitionen von Informationen ab,

die wir aus der Datenbank beziehen. Wenn die Daten nicht korrekt sind, werden die Informationen, die wir aus ihnen ableiten, nutzlos oder sogar schädlich. Die Konsequenz aus inkorrekten Daten kann ein schlechtes Essen ebenso wie eine erhebliche Fehlinvestition sein. Wir werden im Laufe des Kapitels erfahren, dass die Eigenschaften eines DBMS keinesfalls einheitlich, sondern sehr produktspezifisch sind. Über allem steht aber die Anforderung, dass der Datenbestand stets konsistent – also logisch korrekt – sein muss. Wie das mit Hilfe eines DBMS erreicht werden kann, sehen wir bald.

Definition: Konsistenz

Jede Änderung des Datenbestands überführt die Datenbank von einem logisch korrekten Zustand in einen anderen logisch korrekten Zustand.

1.2 Keine Datenbank ohne Datenbankmanagementsystem

Das DBMS ist dabei eine Software, die die Rolle übernimmt, die wir Menschen bei der manuellen Kontrolle der Daten gespielt haben. Es gibt heute eine so große Zahl von DBMS mit so verschiedenen Charakteristika, dass eine weitere Präzisierung nicht ganz leicht fällt. Fast immer, wenn wir den Begriff DBMS genauer fassen wollen, entgeht uns wieder ein Spezialfall, der auch mit Fug und Recht als Eigenschaft eines DBMS bezeichnet werden kann.

Wir werden auch beobachten, dass der Begriff der *Datenbank* vage bleibt und unser Interesse vielmehr der Verwaltungssoftware gilt. In diesem Kapitel wollen wir die grundlegenden Eigenschaften eines typischen DBMS beschreiben und uns am Ende des Kapitels mit den gewonnenen Einsichten einen Überblick über den weiteren Verlauf des Buches verschaffen.

Die Hauptaufgabe eines DBMS besteht darin, seinen Anwendern die Möglichkeit zu geben,

- Daten in die Datenbank einzufügen,
- Daten aus der Datenbank zu löschen,
- Daten in der Datenbank zu ändern und
- Daten in der Datenbank zu suchen.

Das alleine ist immer noch eine sehr allgemeine Beschreibung eines DBMS. Diese elementaren Operationen können auch mit Datenstrukturen realisiert werden, wie sie uns der Java-Collection-Framework zur Verfügung stellt. Der folgende Java-Code zeigt eine einfache Implementierung.

```
import java.util.*;

public class SimpleDB {
 public static void main(String[] args) {
  List<String> database = new ArrayList<String>();
  database.add("Elvis Presley");
  database.add("Beatles");
  database.add("Rolling Stones");
  database.remove(database.indexOf("Beatles"));
  int index=database.indexOf("Rolling Stones");
  database.set(index, "The Rolling Stones");
  System.out.println(database.contains("Elvis Presley"));
  System.out.println(database);
 }
}
```

Wir erzeugen ein Objekt `database` vom generischen Typ `List<String>` und fügen nach und nach drei Objekte in diese Datenbank ein. Um ein Objekt zu löschen, ermitteln wir zunächst seinen Index, also die Position in der Liste. Da die Zählung bei 0 beginnt, ist das im Fall der Beatles die 1. Mit der Methode `remove` löschen wir hier das Objekt mit dem Index 1. Indem wir den Index des Textes `"Rolling Stones"` ermitteln und diesen Index zusammen mit dem Text `"The Rolling Stones"` der Methode `set` übergeben, führen wir eine Änderung durch. Die Methode `contains` prüft, ob ein Objekt in unserer Liste vorhanden ist. Wir nutzen sie zur Suche in unserer Datenbank. Das Java-Programm hat also die folgende Ausgabe:

```
true
[Elvis Presley, The Rolling Stones]
```

1.3 Dauerhafte Speicherung

Wir sehen, dass wir mit wenigen Codezeilen ein einfaches DBMS entwickeln können. Alle Daten werden hier im Hauptspeicher gehalten, so dass die Datenbank nur während der Laufzeit unseres Programms existiert. Das bedeutet, dass diese Form eines DBMS nicht zur *dauerhaften* Datenhaltung genutzt werden kann. *Flüchtige* Daten sind Daten, die nur in dem Kontext existieren, in dem sie erzeugt wurden. Gelegentlich werden sie auch als **volatil** bezeichnet. **Persistente** Daten dagegen werden auf Speichermedien wie Festplatten gehalten und überleben den Kontext ihrer Erzeugung. Sie sind verfügbar, auch wenn die Software, mit der sie angelegt wurden, bereits das Zeitliche gesegnet hat und der Rechner, auf dem sie erzeugt wurden, ausgeschaltet ist. Eine Eigenschaft eines DBMS lautet demnach:

Hinweis

Ein DBMS kann Daten persistieren, also dauerhaft verfügbar machen.

Es sei aber darauf hingewiesen, dass es verbreitete DBMS wie H2[1] gibt, bei denen die volatile Datenhaltung durchaus möglich ist. Die Datenhaltung im Hauptspeicher hat nämlich den großen Vorteil, dass sie wesentlich effizienter ist als die Datenhaltung auf Festplatten. Die Ein- und Ausgabe, also der Datentransport zwischen Hauptspeicher und Festplatte, ist ein signifikanter Engpass für persistierende DBMS. Wenn die Daten nicht für die Ewigkeit geschaffen werden, spricht eigentlich nichts gegen flüchtige Datenhaltung.

1.4 Alle auf einen

Viele Daten sind nur für unseren persönlichen Gebrauch bestimmt. Wenn wir etwa Kochrezepte oder unsere CDs verwalten wollen, benötigen wir diese Informationen in der Regel für uns alleine. Auch wenn noch Freunde, Familien- oder Haushaltsmitglieder davon profitieren sollen, handelt es sich doch um eine sehr private Datensammlung. Da Anwender sich zunächst eher mit privaten Datenbanken beschäftigen, übersehen sie oft, dass dies *nicht* der typische Fall ist. Die Inhalte der meisten Datenbanken werden vielen, zum Teil sogar sehr vielen Personen zur Verfügung gestellt: Der Datenbestand des Internet-Kaufhauses Amazon wird täglich von mehreren Millionen Personen genutzt.

Hinweis

Ein DBMS kann mehreren Anwendern gleichzeitig den Zugriff auf Daten ermöglichen.

Da wir es mit gleichzeitigen Zugriffen auf den gleichen Datenbestand zu tun haben, müssen wir – wenn wir selbst ein DBMS entwickeln wollen – auch die Tücken des gleichzeitigen Zugriffs auf unsere Daten berücksichtigen, um fehlerhafte Datenbestände zu vermeiden. Wenn wir uns beispielsweise für eine CD interessieren, die auf der Webseite unseres Händlers mit dem Hinweis „Auf Lager“ versehen ist, dann wollen wir nicht, dass gerade dann ein anderer Kunde das letzte Exemplar kauft, während wir die Beschreibung der CD lesen. Auch wenn die Mehrbenutzerfähigkeit eine typische Eigenschaft eines DBMS ist, so gibt es doch Datenbanksysteme, die nicht für den Zugriff durch mehrere Benutzer entwickelt wurden.

[1] www.h2database.com

1.5 Auf Nummer sicher

Nur weil mehrere Benutzer auf den gleichen Datenbestand zugreifen dürfen, heißt das noch lange nicht, dass sie alles mit den Daten machen dürfen. Einige Anwender sollen vielleicht Daten nur lesen, aber nicht verändern dürfen. Einige Daten, wie etwa die Mitarbeitergehälter in einer Firma, sind sogar so sensibel, dass sie nur wenigen Nutzern zugänglich sein sollen. Und selbstverständlich wollen wir auch nicht sämtlichen Personen, die Zugang zum Netzwerk unserer Firma haben, den Zugriff auf unsere Daten erlauben.

Hinweis

Ein DBMS kann die Sicherheit der verwalteten Daten sicherstellen, indem es die Definition feingranularer Zugangsbeschränkungen zu den Daten ermöglicht.

Die Granularität reicht also vom vollständigen Ausschluss nicht autorisierter Anwender über spezifische Rechte zum Lesen, Ändern, Einfügen und Löschen von Daten bis hin zur „Generalvollmacht“, also dem Recht zur unbeschränkten Bearbeitung der Daten unserer Datenbank. Datenbanksysteme, wie man sie in Unternehmen findet, haben selbstverständlich diese Eigenschaft. Andere Systeme, wie SQLite[2], haben ihren Fokus auf einfacher Handhabung und schneller Verarbeitung. Sie verzichten dafür auf administrativen Ballast wie eine Benutzerverwaltung. Das Handy-Betriebssystem Android verfügt beispielsweise über dieses DBMS: Hier kommt es auf eine zügige und ressourcenschonende Verarbeitung an. Mehrere Benutzer sind dagegen bei Mobiltelefonen nicht vorgesehen, eine Benutzerverwaltung ist dementsprechend überflüssig.

1.6 Damit alles stimmt

Eine Datenbank hilft uns nur, wenn wir uns auf ihren Inhalt verlassen können. Insbesondere erwarten wir, dass die Daten logisch korrekt sind. So wollen wir beispielsweise bei Daten über Mitarbeiter sicherstellen, dass das Datum ihrer Entlassung zeitlich nach dem Datum der Einstellung liegt. Dabei bestimmen *wir* bereits bei der Definition unserer Datenbank, was „logisch korrekt“ bedeutet. Der Begriff „Konsistenz“ ist in unserem Kontext semantisch zu verstehen, das heißt, er enthält Bedeutung.

Bei Gehältern wollen wir gewährleisten, dass sie nicht negativ werden. Wenn wir eine Sammlung von Musiktiteln haben, wollen wir vermeiden, dass es Lieder ohne einen Interpreten gibt. Diese Beispielliste von Anforderungen an die Datenkonsistenz können wir fortsetzen. Allen Regeln ist aber gemeinsam, dass sie nicht

[2] www.sqlite.org

von einem Stück Software, wie etwa dem DBMS, gefunden werden können. Wir formulieren, was in *unserem* Kontext und für *unseren* Datenstand der Begriff „Konsistenz“ bedeutet. So mag es in anderen Szenarien etwa Datenbanken mit Liedern geben, deren Interpret niemanden interessiert. Unsere Anforderungen an einen konsistenten Datenbestand formulieren wir in Form so genannter **Integritätsregeln**.

Hinweis

Ein DBMS überwacht die Einhaltung von Integritätsregeln.

Wenn wir also die Regel formulieren, dass es keine negativen Gehälter geben darf, dann kann niemand – auch wenn er noch so viele Rechte hat – mit Hilfe des DBMS eine Operation ausführen, die den Datenbestand so ändert, dass er negative Gehälter enthält. Die Menge aller Integritätsregeln definiert die Konsistenz unserer Daten. Chris Date, einer der Datenbank-Gurus, hat das einmal sehr treffend (siehe [Dat03]) so formuliert:

> *„Security means protecting the data against unauthorized users. Integrity means protecting the data against authorized users.“*

Die Integritätsregeln sind also unabhängig von der Sicherheit. *Wie* diese Regeln formuliert werden, hängt wieder sehr stark vom Datenbanksystem ab. Da Integritätsregeln die Korrektheit der Daten sicherstellen, gelten sie für *alle* Anwender und sind in keinem Fall nur auf bestimmte Anwender oder Anwendergruppen beschränkt. *Niemand* darf die Konsistenz unserer Daten zerstören.

Die wesentlichen Aufgaben eines DBMS kann man etwa wie folgt in einem Satz zusammenfassen:

Hinweis

Ein DBMS versorgt berechtigte Anwender mit konsistenten Daten.

Wir haben bisher einige typische Eigenschaften eines DBMS gesehen, aber auch gelernt, dass nicht jedes DBMS jede dieser Eigenschaften hat. Hinsichtlich der Konsistenz des Datenbestandes gibt es aber keine Kompromisse. Ein – auch teilweise – inkonsistenter Datenbestand ist wertlos. Inkonsistenzen können nur durch Änderungen des Datenbestandes entstehen. Das DBMS muss also immer dann die Konsistenz sicherstellen, wenn Anwender Daten ändern.

Wenn mehrere Anwender mit den Daten arbeiten, müssen auch Probleme, die sich aus dem gleichzeitigen Zugriff ergeben, berücksichtigt werden.

1.7 Tornadosicher

Da der laufende Betrieb immer wieder durch Störungen unterbrochen werden kann, muss das DBMS fehlertolerant arbeiten. Bei Stromausfällen darf es beispielsweise nicht passieren, dass Datensätze nur teilweise auf die Festplatte geschrieben wurden. Die Daten müssen nicht nur logisch, sondern auch physikalisch konsistent sein. Im Idealfall reagiert unser DBMS auf praktisch jeden denkbaren Fehlerfall so, dass der laufende Betrieb stets gewährleistet ist. Für Unternehmen wie das Internet-Kaufhaus Amazon ist dies von erheblicher Bedeutung (siehe [DeC07]):

> *„... customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados."*

In den meisten Fällen ist es wichtig, dass unsere Anwender störungsfrei, also ohne Unterbrechungen, auf Daten zugreifen können. Wenn Kunden nicht auf Artikel zugreifen können, führt das zu Umsatzeinbußen; wenn Mitarbeiter die Hände in den Schoß legen müssen, weil das DBMS gerade nicht läuft, führt das zu erhöhten Kosten. Welchen Aufwand wir betreiben, um unser DBMS unterbrechungsfrei zu betreiben, hängt im Wesentlichen von den potenziellen Umsatzeinbußen und den entstehenden Kosten eines Ausfalls ab. Wenn wir bereit sind, genug zu investieren, können wir ein praktisch beliebig hohes Maß an so genannter Fehlertoleranz erzielen.

Hinweis

Ein DBMS arbeitet zuverlässig und fehlertolerant.

Diese Eigenschaft ist natürlich auch nicht unbedingt erforderlich: Einfache Anwendungen mit privater Nutzung können einen Ausfall leicht wegstecken, ohne dass die Welt zusammenbricht.

1.8 Der Mensch

Ganz ohne menschliche Unterstützung kommt ein DBMS aber in den meisten Fällen noch nicht aus: Der Datenbankadministrator (**DBA**) legt die Struktur des Datenbestandes fest, er definiert die Integritätsregeln der Datenbank und vergibt und entzieht Zugangsberechtigungen. Der DBA bildet die Schnittstelle zur Supportorganisation des DBMS-Herstellers. Bei Störungen weiß er, was zu tun ist, um das DBMS wieder in den laufenden Betrieb zurückzubringen.

1.9 Warum nicht selber machen?

So schlüssig und vollständig sich diese Anforderungen an ein DBMS auch anhören, es fehlt doch eine ganz entscheidende Zutat. Um das zu verstehen, implementieren wir im folgenden Java-Code ein eigenes, sehr einfaches DBMS zur Verwaltung von Personen, für die wir eine eigene Java-Klasse (`Person`) definieren.

Listing 1.1: Die Klasse `Person`

```
public class Person {
 private String firstName, lastName;

 public Person(String firstName, String lastName) {
  super();
  this.firstName = firstName;
  this.lastName = lastName;
 }
 static Person toPerson(String text) {
  String[] attributes=text.split("\\|");
  if(attributes.length!=2) throw new RuntimeException();
  else return new Person(attributes[0], attributes[1]);
 }
 public String getFirstName() {
  return firstName;
 }
 public void setFirstName(String firstName) {
  this.firstName = firstName;
 }
 public String getLastName() {
  return lastName;
 }
 public void setLastName(String lastName) {
  this.lastName = lastName;
 }
 public String toString() {
  return firstName + "|" + lastName+"\n";
 }
 public boolean equals(Object o){
  throw new RuntimeException();
 }
}
```

Die Klasse besteht nur aus einem Konstruktor, einer `toString`-Methode, die unsere Personen in Text umwandelt, und einer statischen Methode, die aus

einem Text der Form `"Mickey|Mouse"` eine Person mit Vornamen "Mickey" und Nachnamen "Mouse" macht. Eine solche statische Methode wird auch als Factory bezeichnet. In unserem Fall passt sie sogar genau zur `toString`-Methode: Wenn `p` ein Objekt vom Typ Person ist, so ist dieses Objekt gleich `Person.toPerson(p.toString)`. Der Typ `Person` kann natürlich um weitere Methoden und Attribute ergänzt werden.

Bei einem wirklich einfachen System ist auch die Schnittstelle des DBMS sehr einfach gehalten. Das Interface unseres `DBMS` besteht aus Methoden zum Öffnen und Schließen der Datenbank sowie je einer Methode zum Einfügen und Suchen der Datensätze:

Listing 1.2: Die Klasse `DBMS`

```
public interface DBMS {
 void open() throws IOException;
 void close() throws IOException;
 void insert(Person person) throws IOException;
 List<Person> selectByFirstName(String firstName);
}
```

Die Implementierung ist ebenfalls einfach gehalten: Die Datensätze werden im Format von `toString` in eine Datei geschrieben, aus der sie wieder mit Hilfe der Methode `toPerson` ausgelesen werden. Zum Schreiben verwenden wir die Klasse `BufferedWriter`, zum Lesen die Klasse `Scanner` aus der Java-API, die einfach handhabbar ist.

Listing 1.3: Die Klasse `SimpleDBMS`

```
public class SimpleDBMS implements DBMS {
 private String file;
 private Writer out;
 private Scanner in;

 public SimpleDBMS(String file) {
  this.file = file;
 }
 public void open() throws IOException {
  out = new BufferedWriter(new FileWriter(file, true));
  in = new Scanner(new File(file));
 }
 public void close() throws IOException {
  out.close();
  in.close();
 }
 public void insert(Person person) throws IOException {
```

```
    out.append(person.toString());
    out.flush();
   }
   public List<Person> selectByFirstName(String firstName) {
    List<Person> result = new ArrayList<Person>();
    while (in.hasNext()) {
     Person person = Person.toPerson(in.next());
     if (person.getFirstName().equals(firstName))
      result.add(person);
    }
    return result;
   }
  }
```

Das System arbeitet eigentlich ganz zufriedenstellend: Wenn wir unter 100 000 Datensätzen nach dem zuletzt eingefügten Datensatz suchen, benötigen wir dazu nicht einmal eine Sekunde. Gute DBMS sind da möglicherweise schneller, aber für einen ersten Wurf ist das gar nicht schlecht. Auf der Basis dieses einfachen Programms könnten wir jetzt eine klassische Software mit Anwendungslogik und GUI entwickeln. Bis auf einige Ausnahmen, wie etwa die Mehrbenutzerfähigkeit, sind auch die Charakteristika eines DBMS erfüllt. Integritätsregeln können formuliert werden, wenn wir beispielsweise im Konstruktor der Klasse `Person` die Parameter auf Plausibilität prüfen. Wir sehen aber bereits hier, dass wir immer dann, wenn sich diese Spielregeln ändern, auch den Code ändern müssen.

Irgendwann wollen wir in unserer Datenbank auch weitere Daten verwalten: Möglich sind Adressen oder Aufträge, die wir von den Personen erhalten haben. Wir bemerken bald, dass wir nicht nur GUI und Anwendungslogik unserer Software, sondern auch die Datenhaltung signifikant ändern müssen. Für einen beliebigen Datentyp `T` können wir Teile unseres DBMS – wie etwa die Methode `insert` – in Java generisch implementieren. Die Signatur muss nur in

```
void <T> insert()
```

geändert werden. Objekte wie Adressen ziehen einen ganzen Rattenschwanz an Entscheidungen nach sich: Soll zu jeder Person die Adresse in einer Datei zusammen mit dem Datensatz der Person gespeichert werden, oder kommen alle Adressdaten in eine eigene Datei? Im ersten Fall müssen wir die Möglichkeit berücksichtigen, dass eine Person mehrere Adressen haben kann. Im zweiten müssen wir eine eigene Klasse für Adressen definieren und noch vereinbaren, wie Adressen und Personen miteinander verknüpft werden, wie unser DBMS also zu einer Person die zugehörigen Adressen ermittelt. Die Definition einer eigenen Klasse für die Adressen stellt uns vor keine großen Probleme, da sie der Klasse `Person` sehr ähnelt. Der Typ `DBMS` muss jedoch um einige Methoden, wie die Abfrage `selectByStreet`, erweitert werden.

Wenn wir also Eingriffe in den Code unseres DBMS nicht scheuen, können wir grundsätzlich jede Anforderung an unsere Datenbank umsetzen, müssen aber damit rechnen, dass Änderungen im Einzelfall erhebliche Eingriffe in den Code nach sich ziehen können.

So haben wir in unserem Java-Beispiel bisher die Operation zum Löschen von Daten vermieden. Auch hier müssen Algorithmen entwickelt werden, die eine effiziente Arbeit mit unseren Daten ermöglicht. Beim Entfernen der Daten liegt die Lösung nicht auf der Hand: Soll die Datei unmittelbar nach dem Löschen reorganisiert werden, indem die Lücke, die der gelöschte Datensatz hinterlässt, sofort geschlossen wird? Oder sollen Datensätze nur als gelöscht markiert und die Datei in regelmäßigen Intervallen reorganisiert werden?

Lange Zeit waren Lösungen im Sinne unseres DBMS der Standard: Benutzer greifen mit Hilfe einer API (Application Programming Interface) aus einem Programm heraus direkt auf den Datenbestand zu. Zwei Nachteile liegen auf der Hand:

- Mit Änderungen der Datenstrukturen können auch Änderungen der API einhergehen, die dann wiederum Auswirkungen auf die Anwendungslogik und möglicherweise auf die GUI des Gesamtsystems haben.
- Da die API im Allgemeinen nicht vollständig generisch definiert werden kann, muss sich der Anwender des DBMS um interne Details der Datenorganisation kümmern. Er muss ähnlich wie beim Java-Collection-Framework verstehen, wie das DBMS intern arbeitet.

Diese Nachteile scheinen nicht gravierend, führten Ende der 1960er-Jahre aber zum „Application Backlog“, einer Situation, in der die IT-Abteilungen von Unternehmen die Anforderungen der Fachabteilungen nicht mehr umsetzen konnten. Bruce Lindsay, ein weiterer Datenbank-Pionier, erinnert sich (siehe [Win05]):

> *„It was a good deal to be a DBA, because you controlled what got done next. It was necessary for anybody that wanted anything to get done to bring you gifts, you know, bottles of things. And therefore it was a good deal to be a DBA, because of this application backlog.“*

1.10 Das ANSI SPARC-Modell als Lösung

Der Umstand, dass Änderungen bei den für die Daten verantwortlichen Algorithmen und Datenstrukturen zu signifikanten Änderungen an der logischen Struktur der Daten einer Software führen können, wird auch als **physikalische Datenabhängigkeit** bezeichnet.

Die Änderungen verlaufen zwar oft nach gleichen Mustern, müssen aber durchgeführt werden und können jedes Mal zu Fehlern führen. In den 1970er-Jahren erarbeitete das Standards Planning and Requirements Committee (SPARC) des

American National Standards Institute (ANSI) eine Referenzarchitektur für Datenbanksysteme, die Datenunabhängigkeit gewährleisten sollte. Die Architektur sieht drei Schichten in Form einer physikalischen, einer logischen und einer externen Ebene vor.

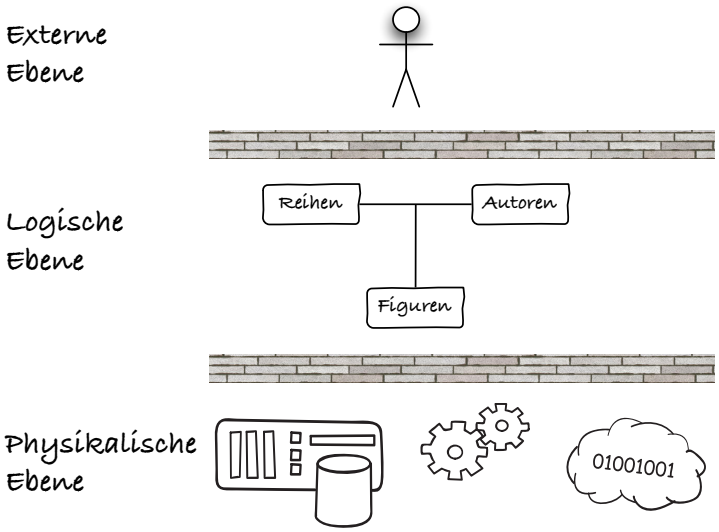

Abbildung 1.1: Das ANSI SPARC-Modell

Die physikalische Ebene: Wenn wir Daten auf Platten schreiben oder über Netzwerke anfordern, greifen wir dazu nicht direkt auf Geräte wie den Festplatten-Controller oder die Netzwerkkarte zu, sondern übertragen diese Aufgaben dem Betriebssystem. Wie das Betriebssystem diese Aufgaben erfüllt, müssen wir dazu nicht wissen. Für uns sind das *Implementierungsdetails*. Die Schnittstellen, die uns das Betriebssystem liefert, ermöglichen so eine höhere Abstraktion. Diese Idee wird auch vom ANSI SPARC-Modell aufgegriffen. Die Implementierungsdetails des DBMS werden auf der so genannten physikalischen Ebene definiert: Dies sind etwa das Speicherformat der Daten, Algorithmen zum Einfügen, Löschen, Ändern und Suchen von Daten oder die Verwaltung von Sperren, die im Mehrbenutzerbetrieb nötig sind. All diese Details werden vom DBMS verborgen. Die physikalische Ebene beinhaltet, anders als unser selbstgebautes System zur Verwaltung von Personen, keinerlei Semantik. Auf der physikalischen Ebene können daher auch keine Integritätsregeln vereinbart werden. Ihre Funktionalität stellt die physikalische Ebene über eine Schnittstelle zur Verfügung. So können auf der physikalischen Ebene gravierende Änderungen durchgeführt werden, ohne dass sich die Schnittstelle ändert und wir als Anwender etwas davon merken.

Das DBMS gewinnt so an Robustheit: Auf der physikalischen Ebene kann beispielsweise das Speicherverfahren für Datensätze komplett ausgetauscht werden, ohne dass der Anwender davon betroffen ist. Die physikalische Ebene ist das Reich der Bits und Bytes, in dem es keine *Semantik* gibt. Der Inhalt der Daten ist bedeutungslos. Die Isolierung der physikalischen Ebene vom Rest des DBMS stellt ein hohes Maß an physikalischer Datenunabhängigkeit sicher.

Die logische Ebene: Auch wenn uns die physikalische Ebene bereits einige Abstraktion liefert, reicht uns das noch nicht. Es fehlt die Semantik, also die Bedeutung der Daten. Welche Arten von Daten (Personen, Gegenstände oder Termine) gibt es eigentlich in unserer Datenbank, und in welcher Beziehung stehen sie zueinander? Wenn wir diese konzeptionellen Begriffe im Code der Anwendungslogik definieren, werden wir nie einen Gesamtüberblick darüber haben, wie unsere Datenbank konzipiert ist. Überall in unserer Software könnten neue Daten definiert werden, so dass am Ende kein Mensch mehr einen Gesamtüberblick hat. Die Semantik ist in diesem Chaos auch irgendwo vorhanden, jedoch möglicherweise nicht mehr überschaubar. Da aber in einem Softwareprojekt neben den Anwendungsentwicklern mehrere Parteien wissen müssen, um was es in der Datenbank eigentlich geht, ist es wichtig, über ein zentrales Verzeichnis der Daten und ihres Beziehungsgeflechts – also ein Modell der Datenbank – zu verfügen.

Auf der logischen Ebene wird definiert, welche Daten in der Datenbank gespeichert werden. Hier spielt die Semantik der Daten die zentrale Rolle. Die logische Struktur der Daten wird beschrieben und die Beziehung der Daten untereinander vereinbart. So sollen beispielsweise, Daten über CDs und die Lieder in der Datenbank abgelegt werden. Es wird festgelegt, dass

- CDs durch ihren Titel und ihren Interpreten und
- Lieder durch ihren Titel und ihre Position innerhalb der CD

charakterisiert werden. Die Beziehung zwischen CDs und Liedern besteht darin, dass jedes Lied zu einer CD gehört und jede CD mindestens ein Lied enthält. Das alles kann beschrieben werden, ohne das kleinste Detail über die physikalische Ebene zu kennen; Änderungen auf physikalischer Ebene haben damit auch keine Konsequenzen für die logische Ebene.

Die externe Ebene: Datenbanksysteme werden häufig von Sachbearbeitern innerhalb der Firma aber natürlich auch von Kunden genutzt, die nicht zur Firma gehören. Eine weitere Nutzergruppe stellen die Anwendungsentwickler dar, diejenigen also, die mit Hilfe von Entwicklungsumgebungen Software schreiben, die auch mit unserem DBMS kommuniziert und so auf unsere Datenbank zugreift. Diese Software wird dann dem Endanwender zur Verfügung gestellt.

Der Anwendungsentwickler muss sich daher gut mit der Datenbank auskennen. Das Leben ist leider nicht immer so überschaubar wie im Fall von Liedern und CDs. Eine Enterprise-Software wie SAP R/3 nutzt beispielsweise eine Datenbank mit weit über 10 000 Tabellen, die für den einzelnen Anwender unüberschaubar

ist. Da jeder Anwender – auch der Anwendungsentwickler – nur einen bestimmten Teilbereich der Datenbank kennen muss, ist eine vollständige Sicht auf die logische Ebene weder notwendig noch wünschenswert. Die externe Ebene des ANSI SPARC-Modells stellt **Sichten** (dt. für *views*) zur Verfügung, die einzelnen Anwendern oder ganzen Anwendergruppen diejenigen Ausschnitte aus der *logischen* Ebene geben, die sie benötigen. Diese Sichten können dabei das logische Modell weiter abstrahieren. Neben dieser Anpassung an individuelle Erfordernisse kann so auch die logische Ebene gekapselt werden. Die Sichten sind damit die Schnittstellen der externen Ebene zur logischen Ebene. Bei Änderungen auf logischer Ebene müssen nur die Sichten angepasst werden. Diejenigen Teile der Anwendungslogik, die die Sichten nutzen, bleiben unverändert. Diese Art der Datenunabhängigkeit wird als **logische Datenunabhängigkeit** bezeichnet.

Das ANSI SPARC-Modell gilt noch heute weitgehend als die ideale Software-Architektur für ein DBMS. Die klare Trennung der Schichten, die nur sehr kontrolliert miteinander kommunizieren können, ist Grundlage für die logische und physikalische Datenunabhängigkeit. Sie wird aber in dieser Form selten erreicht, insbesondere gibt es mit Sichten einige sehr fundamentale Probleme, mit denen wir uns noch in Kapitel 15 beschäftigen werden. Dennoch findet man bei modernen Datenbanksystemen die Trennung der drei Ebenen, auch wenn sie nicht vollständig gegeneinander gekapselt sind. Wer sich zum ersten Mal mit Datenbanken beschäftigt, kann die mehrschichtige Architektur des ANSI SPARC-Modells vielleicht nicht ganz einfach nachvollziehen. Vieles wird sich aber in den folgenden Kapiteln ergeben, wenn wir die einzelnen Ebenen eingehender diskutieren und anhand von Beispielen illustrieren.

Da es heutzutage eine Vielzahl von leistungsstarken DBMS gibt, die teilweise auch kostenfrei genutzt werden können, gibt es auch keinen Grund mehr, eigene Softwarekomponenten für die Verwaltung von Daten zu entwickeln. Jedes Mal, wenn wir in unserer Software Daten in Dateien eintragen oder aus Dateien lesen, sollten wir den Einsatz einer Standardsoftware in Form eines DBMS erwägen. Tatsächlich können wir sogar den Einsatz von In-Memory-Datenbanken erwägen, wenn wir speicherresidente Daten intensiv nutzen. Dabei wird der Zugriff auf langsame Speichermedien, wie Festplatten, vermieden. Die Geschwindigkeit steht möglicherweise nur wenig hinter einer selbstentwickelten Lösung zurück; in jedem Fall ist die Standardsoftware aber ausgereifter als unsere selbstgestrickte Datenverwaltung.

1.11 Wie alles anfing

Die Idee, Programme zu entwickeln, mit deren Hilfe die Datenhaltung einer Software nicht jedes Mal aufs Neue erfunden werden muss, begann mit der zunehmenden Verbreitung der entsprechenden Hardware in Form von Festplatten in den 1960er-Jahren.

Das erste marktreife Produkt, das wir aus heutiger Sicht als DBMS bezeichnen können, ist wohl der Integrated Data Store (IDS), den Charles Bachman bei General Electric entwickelte. Die Daten werden in einer netzwerkähnlichen Struktur abgespeichert, die aus Datenknoten und Kanten zwischen diesen Knoten besteht. Entwickler integrieren das DBMS mit Hilfe einer API in ihre eigene Software. Die API ermöglicht es, durch das Netzwerk zu navigieren und so die Daten zu verwalten. Diese Form des DBMS galt seinerzeit als ausgesprochen solide: Die Conference on Data Systems Languages (CODASYL), der wir auch die Entwicklung der Programmiersprache COBOL zu verdanken haben, setzte es sich etwa zum Ziel, Sprachen zu entwickeln, um Daten für Netzwerk-DBMS zu definieren und diese Daten zu bearbeiten. So wundert es nicht, dass noch weitere DBMS für Netzwerkdatenbanken entwickelt wurden. Im Laufe der Zeit wurden diese Produkte immer perfekter, so dass sie auch heute noch ausgesprochen stabil betrieben werden, auch wenn der Support bereits seit vielen Jahren ausgelaufen ist. Dass es bereits frühzeitig diese Art von DBMS gab, heißt noch lange nicht, dass sie auch in vielen Firmen genutzt wurde. In vielen Firmen fand der Umschwung zu dedizierten DBMS erst in den 1990er-Jahren statt. Gelegentlich werden auch heute noch selbstgebaute Systeme mit direktem Zugriff auf das Dateisystem verwendet.

1.12 Mit IMS zum Mond

> *„Ich glaube, dass dieses Land sich dem Ziel widmen sollte, noch vor Ende dieses Jahrzehnts einen Menschen auf dem Mond landen zu lassen und ihn wieder sicher zur Erde zurück zu bringen." (J. F. Kennedy)*

Das amerikanische Apollo-Programm hatte in den 1960er-Jahren einen wesentlichen Einfluss auf die Entwicklung der Technologie. Dazu gehört die Erfindung von Alltagsgegenständen wie der Quarzuhr oder dem Akkubohrer, aber auch die Entwicklung eines DBMS durch IBM in den Jahren 1966–1968, das für die Verwaltung der für die Apollo-Mission eingesetzten Bauteile entwickelt wurde. IMS (Information Management System) ist ein so genanntes hierarchisches DBMS, das hierarchisch organisierte Daten verwalten kann. Komplexe Geräte, wie etwa die Saturn V-Rakete, bestehen aus vielen Einzelteilen, die wiederum aus weiteren Teilen zusammengesetzt sind. Wie bei einer russischen Matrjoschka-Puppe kann diese Zerlegung bis hin zum kleinsten Bauteil weitergeführt werden. Genau für diese Art von hierarchischen Daten wurde IMS ursprünglich gemacht. Hinzu kommen eine ausgesprochen hohe Verarbeitungsgeschwindigkeit und eine sehr hohe Ausfallsicherheit. Diese angenehmen Eigenschaften haben dafür gesorgt, das IMS das Apollo-Programm überlebt hat und bis heute von Banken, Versicherungen und in der Automobilindustrie eingesetzt wird. Auch wenn hierarchische Datenbanksysteme unternehmenskritische Daten verwalten, haben sie heute für die meisten Entwickler keine Bedeutung mehr: Die historisch gewachsenen Systeme (Legacy-

Systeme), in die IMS-Systeme integriert sind, arbeiten zuverlässig und benötigen kaum Wartung.

1.13 Und heute?

Software, die in den letzten zwanzig Jahren entwickelt wurde, nutzt überwiegend relationale DBMS, die eigentlich auf einer ganz einfachen Idee beruhen. Einen ersten Eindruck davon werden wir im nächsten Kapitel bekommen.

Außer dem Netzwerk-, dem hierarchischen und dem relationalen Modell gibt es eine Menge weiterer Konzepte, von denen wir einige im letzten Kapitel des Buches kennenlernen.

Wir haben gesehen, dass es eine Vielzahl von DBMS mit zahlreichen Eigenschaften gibt. Einfache DBMS benötigen nur wenige Ressourcen und können auch für kleine Anwendungen eingesetzt werden; große DBMS sind Tausendsassas mit einem ständig wachsenden Leistungsspektrum. Eigentlich ist für jeden Anwendungsbereich das geeignete DBMS dabei, so dass nur selten Grund besteht, Daten im Dateisystem zu lagern oder sie von dort auszulesen. Systeme wie SQLite schlagen bei der Installation mit weniger als einem Megabyte zu Buche und sind wesentlich zuverlässiger als selbstgeschriebener Code, der Daten in Dateien verwaltet.

Wie wird ein DBMS eigentlich benutzt? Es gibt natürlich einfache Szenarien, in denen die Endanwender mit einem Werkzeug direkt auf die Datenbank zugreifen. Wir werden das selbst im folgenden Kapitel ausprobieren. In der Praxis ist das DBMS mit seinen Datenbanken aber Baustein einer Software-Architektur (siehe Abb. 1.2).

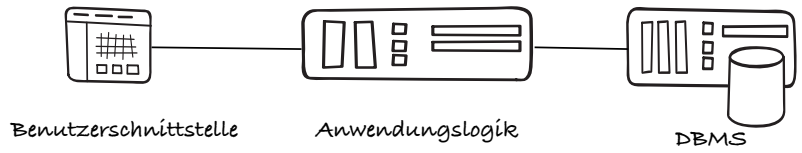

Abbildung 1.2: Einfache Software-Architektur mit DBMS

Weitere Bausteine sind die Anwendungslogik und die Benutzerschnittstelle der Software. Typischerweise finden wir die zentralen Aufgaben der Software in der Anwendungslogik. Hier werden Preise kalkuliert oder Schachkombinationen ermittelt. Um Daten zu persistieren oder auf persistente Daten zuzugreifen, kommuniziert die Anwendungslogik mit dem DBMS. Die Benutzerschnittstelle verbindet die Anwendungslogik mit den Endanwendern. Sie greift nicht auf den Datenbestand zu, sondern stellt die Resultate der Anwendungslogik dar oder fordert

Eingaben vom Benutzer ein, die dann von der Anwendungslogik weiterverarbeitet werden.

1.14 Wie geht es weiter?

Wir haben in diesem Kapitel in lockerer Reihenfolge einige Ideen, Konzepte und Hintergründe zum Thema Datenbanken und Datenbankmanagementsysteme kennengelernt. In diesem Buch konzentrieren wir uns auf relationale Datenbanken. Warum diese Fokussierung sinnvoll ist, zeigen die nächsten Seiten. Dort tasten wir uns Schritt für Schritt in die Welt der Relationen vor. In den Kapiteln 3 und 4 unterfüttern wir unsere Erfahrungen mit einer Theorie, die uns über weite Strecken des Buches immer wieder begegnen wird.

Der zweite Teil des Buches umfasst die Aspekte der logische Ebene des ANSI SPARC-Modells. In Kapitel 6 lernen wir eine Methode kennen, die uns nicht nur hilft, den Überblick über unseren Datenbestand zu behalten, sondern Projektteams eine Kommunikation über die Daten ermöglicht. Diese Form der Modellierung kann nicht unmittelbar mit Hilfe von relationalen DBMS umgesetzt werden. Wir lernen deshalb in Kapitel 5, wie wir in der Praxis Daten und Integritätsregeln definieren, und eignen uns anschließend in Kapitel 7 eine Methode an, um unser Modell praktisch umzusetzen. Trotz dieser bewährten Techniken kann unsere Datenbank noch schwere Designfehler enthalten. In Kapitel 8 begegnen wir so genannten Anomalien. Schon das Wort hört sich nicht gut an. Wir sehen Beispiele für Anomalien und bekommen mit der Normalisierung ein Werkzeug an die Hand, mit dem wir sie vermeiden können.

Der folgende Teil widmet sich vollständig der Abfragesprache SQL: Wir verschaffen uns zunächst in Kapitel 9 einen allgemeinen Überblick und arbeiten uns dann in den Kapiteln 10 bis 14 in die Tiefen von `select`, der wichtigsten und komplexesten SQL-Anweisung, ein. Wie Sichten (siehe Abschnitt 1.10) in relationalen Datenbanken realisiert und mit Hilfe von SQL definiert werden, erfahren wir in Kapitel 15.

Im vorherigen Abschnitt 1.13 haben wir gesehen, dass das DBMS nur Teil einer Software-Lösung ist. Dabei ist der Begriff der **Transaktion** von zentraler Bedeutung. Was darunter zu verstehen ist und warum Transaktionen so wichtig sind, lernen wir in Kapitel 17. Wie andere Komponenten auf relationale DBMS zugreifen können, erfahren wir anhand der Programmiersprache Java in den Kapiteln 18 und 19. Im letztgenannten Kapitel stoßen wir auf ein Werkzeug, das den objektorientierten Ansatz von Java und die mengenorientierte Perspektive von SQL harmonisiert. Ein abschließendes Kapitel vermittelt dann noch einige Hintergründe zu der physikalischen Ebene. Wir erfahren, wie ein relationales DBMS arbeitet, wie die Daten organisiert sind und wie das System Konsistenz und einen zuverlässigen Betrieb erreicht. Mit diesem Hintergrundwissen ausgestattet, verstehen

wir dann auch den Index, ein wichtiges Instrument zur Leistungssteigerung relationaler DBMS.

Mit Netzwerk- und hierarchischen Datenbanken haben wir uns bereits in diesem Kapitel (siehe Abschnitt 1.11 und 1.12) beschäftigt. Objekt- und XML-Datenbanken fristen zusammen mit einer Vielzahl von NoSQL-Datenbanken[3] nur ein Nischendasein. Da es aber in der Praxis durchaus Fälle geben kann, in denen diese Arten von Datenbanken besser sind als traditionelle SQL-basierte, relationale Datenbanken, gibt der letzte Teil des Buches einen kurzen praxisorientierten Überblick zu jedem dieser drei Typen.

Alles klar?

- Datenbanken sind Datensammlungen.
- Datenbanken werden mit Hilfe einer Software, dem Datenbankmanagementsystem (DBMS), verwaltet.
- Es gibt keinen einheitlichen Anforderungskatalog an ein DBMS. Einige typische Eigenschaften sind
  - Dauerhafte Speicherung der Daten
  - Überwachung von Integritätsregeln
  - Authentifizierung und Rechtevergabe
  - Fehlertoleranz
- Im ANSI SPARC-Modell werden die physikalische, die logische und die externe Ebene unterschieden.
- Die physikalische Ebene enthält im Wesentlichen die Algorithmen und Datenstrukturen, die das DBMS nutzt.
- Auf der logischen Ebene werden die Daten unabhängig von der physikalischen Ebene modelliert.
- Die externe Ebene stellt Sichten auf das Datenmodell zur Verfügung.
- Physikalische Datenunabhängigkeit bezeichnet die Eigenschaft, dass Änderungen innerhalb der physikalischen Ebene für die logische und die externe Ebene transparent sind.
- Logische Datenunabhängigkeit bezeichnet die Eigenschaft, dass Änderungen innerhalb der logischen Ebene für die externe Ebene transparent sind.
- Die historische Entwicklung der Datenbanken führt von Netzwerk- über hierarchische zu relationalen DBMS. Neben diesen Paradigmen existiert eine Vielzahl weitere Typen von DBMS.

[3] NoSQL ist dabei ein Akronym für „not only SQL“.