Datenbanken

Victor Hahn

Version 0.5 – letztes Update: 3. Juni 2012

Kontakt: info@victor-hahn.de

Inhaltsverzeichnis

1 Einleitung

Herzlich willkommen in der Welt der Informatik! Dieses Script soll dir die Möglichkeit geben, (fast) alles, was wir im Unterricht besprochen haben, noch einmal nachzulesen. Wahrscheinlich findest du hier auch ein paar Informationen mehr, als wir im Unterricht gemacht haben – die Darstellung ist vielleicht etwas „theoretischer“, um fachlich korrekt zu sein. Lass dich davon bitte nicht verwirren.

Dieses Script ist noch nicht fertig! Es wird im Laufe des Halbjahres wachsen und neue Kapitel werden hinzukommen. Das bedeutet auch, dass es wahrscheinlich nicht frei ist von Fehlern und anderen hässlichen Dingen. Wenn du einen Fehler findest oder einfach etwas, was man besser hätte erklären können, sag mir bitte Bescheid. Danke!

1.1 Was sind Datenbanken?

Viele von euch haben in Informatik schon gelernt, wie man programmiert – das Handwerkszeug von Informatikern. Informatik als „die Lehre von der automatisierten Verarbeitung von Informationen“ hat aber noch viele andere Aspekte. Datenbanken sind ein wichtiges Werkzeug: Sie erlauben uns, große Mengen an Daten systematisch zu speichern und zu verarbeiten.

In diesem Kurs beschäftigen wir uns ausschließlich mit *relationalen Datenbanken*. Diese Datenbanken haben also eine mathemathische, mengentheoretische Definition, mit der wir uns in Kapitel 6 beschäftigen werden. Für den Anfang reicht es, wenn wir sagen, Datenbanken speichern Daten in Tabellen.

Daten in Datenbanken werden verwaltet durch ein *Datenbankmanagementsystem* (DBMS). Wir werden in diesem Kurs *MySQL* verwenden. Ein DBMS gibt uns über eine bestimmte Schnittstelle Zugang zu den Daten – lässt sie uns anlegen, lesen und manipulieren. Diese Schnittstelle ist die Abfragesprache *SQL* (Structured Query Language, englisch: „strukturierte Abfragesprache“), gerne gesprochen wie „Sequel“. Diese Sprache werden wir lernen.

Zusammen bilden Daten und Managementsystem (DBMS) das vollständige *Datenbanksystem* (DBS). Als *Datenbank* bezeichnet man dabei die Daten selbst, die in einem geeigneten Format vorliegen müssen – eben in Form bestimmter Tabellen (mathematisch gesagt: Relationen).

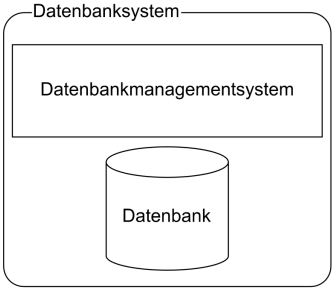

Abbildung 1: Modell eines Datenbanksystems
Bild: Frank Roeing

1.2 Wozu Datenbanken?

Sind Datenbanken also nur eine Art aufgeblasene Tabellen-Kartei? Was ist der wesentliche Unterschied zu beispielsweise Textdateien oder der Datenhaltung mit Tabellenkalkalationsprogrammen wie Excel?

Wir haben am Anfang schon kurz angerissen, dass Datenbanken auf die Verwaltung *großer* Datenbestände optimiert sind. Entsprechend kommen auch nur hier ihre Vorteile wirklich zur Geltung. Kleine Datenbestände sind vielleicht von

Hand viel besser zu Verwalten – ebenso, wie man für für die Aufgabe „1+1“ sicher keinen hochwissenschaftlichen Taschenrechner braucht.

Datenbanksysteme stellen Anwendern die Daten in systematischer und logischer Form zur Verfügung. Diese Anwender können Menschen sein – im Bereich der Datenverarbeitung sind meist aber auch die Anwender *des Datenbanksystems* Programme. Beispiele, in denen Datenbanksysteme eingesetzt werden:

- Internet-Foren
- Soziale Netzwerke
- Banken und Versicherungen
- Lagerhaltungssysteme
- ...

Am Beispiel des Internetforums ist beispielsweise die auf dem Webserver laufende Forensoftware der Anwender der Datenbank. Der Datenbankserver könnte dabei auch auf einer anderen Maschine laufen.

1.3 Vorteile von Datenbanksystemen

Atomarität

Jeder Fakt in der realen Welt hat genau einen bestimmten Platz in der Datenbank. Wir müssen Daten nie von Hand auseinandersortieren.

Beispiel: Ein Versandunternehmen speichert Kundendaten. In der Datenbank gibt es einzelne Felder für Name, Vorname, Straße etc. Wir können so z.B. direkt aus der Datenbank den Vornamen eines bestimmten Kunden erfragen, ohne ihn erst mühsam von anderen Daten, wie beispielsweise dem Nachnamen, „abschneiden“ zu müssen.

Redundanzfreiheit und Konsistenz

Ein bestimmter Fakt aus der realen Welt soll genau einmal in einer einzigen Datenbank abgelegt werden – und nicht etwa mehrfach in verschiedenen Karteien oder Tabellen. So ist sichergestellt, dass Daten nicht *inkonsistent*, also widersprüchlich werden.

Beispiel: Nehmen wir an, ein Versandunternehmen (das keine ordentliche Datenbank führt), hat eine Tabelle mit allen ihren Kundendaten. Zu jedem Kunden sind Name und Adresse abgelegt. Dann führt es eine weitere Tabelle mit den Abonnenten seines Kataloges. Auch in dieser Tabelle sind Name und Adresse von Kunden noch einmal abgelegt – für alle Kunden, die den Katalog beziehen. Nun zieht ein Kunde um. In der Tabelle Kundendaten wird seine Anschrift aktualisiert, in der Tabelle Katalog-Abonnenten wird dies vergessen.

Durch redundante Datenhaltung – also mehrfaches Speichern des gleichen Fakts an mehreren Stellen – ist eine Inkonsistenz entstanden: Es liegen nun sich widersprechende Daten vor (und welche davon sind richtig?).

Unabhängige logische Schicht

Datenbanksysteme unterstützen uns dabei, unsere Daten redundanzfrei abzulegen. DBMS können dabei Daten nach außen hin so umformatieren, dass der Anwender sie im gewohnten Format sieht: In der Datenbank selbst könnten also alle Kundennamen und -adressen an einem zentralen Ort gespeichert werden, dem Anwender könnte die Datenbank aber beispielsweise trotzdem zwei „virtuelle“ Tabelle „Kunden“ und „Katalogabonnenten“ präsentieren, die beide die Adressen enthalten. Dieses Konzept nennt man *Views*.

Datenbanksysteme können also die Nachteile von Redundanzen eliminieren und gleichzeitig ihre Vorteile erhalten, indem sie redundante Sichten auf die Daten simulieren.

„Was statt Wie“

Alle Daten, auch solche in einer Datenbank, müssen physisch irgendwie auf der Festplatte abgespeichert werden. Datenbanksysteme entlasten den Anwender von der Frage der physischen Speicherung (welches Format, unter welchem Pfad, etc.). Diese Aufgabe übernimmt das DBMS – und ist genau darauf optimiert.

Mit einer Sprache wie SQL müssen wir also nicht Pfade oder ähnliches kennen, um zu wissen, wie wir auf die Daten zugreifen. Stattdessen können wir uns darauf konzentrieren, was für Daten wir denn benötigen. Man fasst dieses Prinzip gerne unter dem Schlagwort „Was statt Wie“ zusammen.

Integritätskontrolle

Datenbanksysteme können dabei helfen, Daten vor Korruption zu schützen. So kann ein DMBS beispielsweise eine Anfrage mit einer Fehlermeldung ablehnen, die die Telefonnummer eines Kunden durch Buchstaben ersetzen möchte. Ein DBMS kann unsere Daten also auf *Plausibilität* prüfen.

Auch die Beziehungen zwischen Daten können geschützt werden. Eine Datenbank kann (und sollte!) so eingerichtet sein, dass sie weiß, dass zu einer Bestellung immer ein Kunde gehört – und ein Kunde mehrere Bestellungen haben kann (→ Kapitel 2). Es kann so z.B. verhindern, dass ein Kunde gelöscht wird, so lange Bestellungen dieses Kunden vorliegen. Dies bezeichnet man als *referentielle Integrität*.

Ein DBMS kann auch dafür sorgen, dass eine Datenbank immer in einem konsistenten Zustand vorliegt – also eine Änderung immer ganz oder gar nicht vorgenommen wird, und nicht irgendwie halb. Hierfür bietet sie das Konzept der *Transaktionen*, mit dem wir uns später noch beschäftigen..

Datenschutz und Datensicherheit

In einem großen Datenbanksystem kann es verschiedene Benutzer mit verschiedenen Rollen geben. So kann in einem großen Unternehmen z.B. festgelegt werden, dass nur die Personalabteilung das Gehalt von Mitarbeitern sehen (Datenschutz) und verändern (Datensicherheit) kann.

Zugriffskontrolle

Datenbanken erlauben den gleichzeitigen Zugriff durch verschiedene Anwendungen, ohne dass es dadurch zu Problemen kommt. Sie lösen also das *Leser-Schreiber-Problem* (darauf gehen wir in diesem Kurs nicht weiter ein).

1.4 Und Nachteile?!

Datenbanken, oder sauberer: Datenbankmanagementsysteme, sind Werkzeuge. Es ist immer gut, einen kritischen Blick darauf zu behalten, ob man sie in einer konkreten Situation überhaupt braucht. Wir haben schon angesprochen, dass Datenbanken ein Konzept für Datenverarbeitung in *großen* Mengen ist.

Natürlich haben Datenbanksysteme auch Nachteile. Als Allererstes verursacht der Betrieb des DBMS natürlich einen gewissen *Overhead* – es belegt Ressourcen wie Speicher und Rechenleistung. Was für weitere Nachteile kannst du dir vorstellen?

2 Das Entity-Relationship-Modell

Wie legen wir nun also Daten sinnvoll in einer Datenbank ab? Es hat sich als sinnvoll erwiesen, hierbei nicht etwa von technischen Überlegungen auszugehen, sondern zu untersuchen, wie die anfallenden Daten in der realen Welt logisch zusammenhängen. Hierzu verwenden wir das *Entity-Relationship-Modell* (ER-Modell).

Entity (englisch „Entität“) bezeichnet hierbei ein bestimmtes logisches Etwas, zum Beispiel einen Menschen oder einen Gegenstand.

Eine *Relationship* (englisch „Beziehung“) beschreibt, wie verschiedene Entities miteinander verknüpft sind.

Attribute bezeichnen zusätzliche Daten, die beschreibend zu einer Entity oder zu einer Relationship gehören.

Im ER-Modell werden Entities als Rechtecke und Relationships als Rauten dargestellt. Attribute werden als Elipse dargestellt und mit einem verbindenden Strich an der jeweiligen Entity oder Relationship notiert[1].

Das ganze wollen wir nun an einem Beispiel verdeutlichen.

2.1 Ein Bibliothekssystem

Die Bibliothek von Leserattenstadt hat uns beauftragt, ein Datenbanksystem für sie zu entwerfen. Wir schicken einen Beobachter vor, der das System „Bibliothek“ für uns untersuchen soll. Er kommt zurück mit folgendem Bericht:

[1] Dies gilt in der von uns verwendeten Chen-Notation. Es gibt auch andere Darstellungsmöglichkeiten, von denen sich manche, wie die UML-Notation, auch langsam durchsetzen.

Wie meine Untersuchungen ergeben haben, verleiht die Bibliothek von Leserattenstadt die folgenden Romane:

Douglas Adams: Per Anhalter durch die Galaxis
Aldous Huxley: Schöne neue Welt
George Orwell: Die Farm der Tiere
Johann Wolfgang von Goethe: Faust

Von „Schöne neue Welt“ gibt es zwei Exemplare.

Auf jedem einzelnen Buch klebt ein Schildchen mit einer seltsamen Buchstaben- und Zahlenkombination. Diese ist auf jedem einzelnen Exemplar eines Buches eindeutig und besteht aus den Initialen des Autors, gefolgt von drei Ziffern. Auf Nachfrage erklärte mir die Bibliothekarin dies als die „Signatur“ eines Buches.

Am ersten Tag meiner Beobachtungen betritt eine gewisse Alice Amann, wohnhaft in der Lesestraße 1 in 12345 Leserattenstadt, die Bibliothek. Sie ist begeistert von der großartigen Auswahl und meldet sich an. Auch Bob Bernig, zu Hause in der Lesestraße 2, und Claire Clepto aus der Lesestraße 3 schaffen sich Leseausweise an.

Bob leiht sich am 01.01.2012 „Schöne neue Welt“. Leider vergisst er, es zurückzugeben, so dass er am 14.01.2012 eine Mahnung bekommt und 5 Euro Strafe zahlen muss. Voller Reue gibt er nur einen Tag später das Buch zurück und bezahlt die Strafe. Wenn er schonmal da ist, leiht er sich gleich auch noch „Per Anhalter durch die Galaxis“ aus. Er hat dazugelernt und gibt es nach fünf Tagen zurück.

Claire leiht sich am 03.01.2012 „Schöne neue Welt“ und „Farm der Tiere“ aus. Aus purer Raffgier gibt sie beide nicht zurück, so dass sie am 17.01.2012 je eine Mahnung über 5 Euro bekommt. Trotzdem will sie die Bücher partout nicht zurückgeben, auch die Strafe zahlt sie nicht. Am 03.02.2012 bekommt sie deshalb je eine weitere Mahnung über 10 Euro. Zahlen will sie immer noch nicht - aber um die Bibliothek zu verwirren überweist sie zwei Tage später 42 Cent.

Alice leiht sich am 12.02.2012 „Per Anhalter durch die Galaxis“ aus und hat es bis heute.

Auf Basis dieser Beschreibung wollen wir nun ein ER-Modell „Bibliothek“ erstellen. Wir modellieren also die zugehörigen Daten. Beachte, dass die textuelle Beschreibung eines Systems sicher nicht immer alle Details erfasst, die für die Erstellung einer Datenbank notwendig sind. Modellierung ist immer ein kreativer Prozess: Es gibt mehrere richtige Lösungen – die alle ihre Vor- und Nachteile haben und unterschiedlich geeignet sein können. Bewahre den kritischen Blick!

Wir haben es hier zunächst mit den Entitäten *Leser* und *Buch* zu tun. Diese stehen miteinander in einer Beziehung (Relationship), die wir als *leiht* bezeichnen können: Ein Leser leiht ein oder mehrere Bücher – ein Buch wird geliehen von einem Leser. Diese beiden Entitäten und ihre Beziehung notieren wir.

Dies nennt man eine Beziehung vom Typ 1:n (sprich „eins zu n“): Zu einem Buch gehört immer nur maximal ein Leser (ein Buch kann nicht an mehrere Personen gleichzeitig verliehen werden), aber zu einem Leser können beliebig viele („n“) Bücher gehören. Wir notieren also

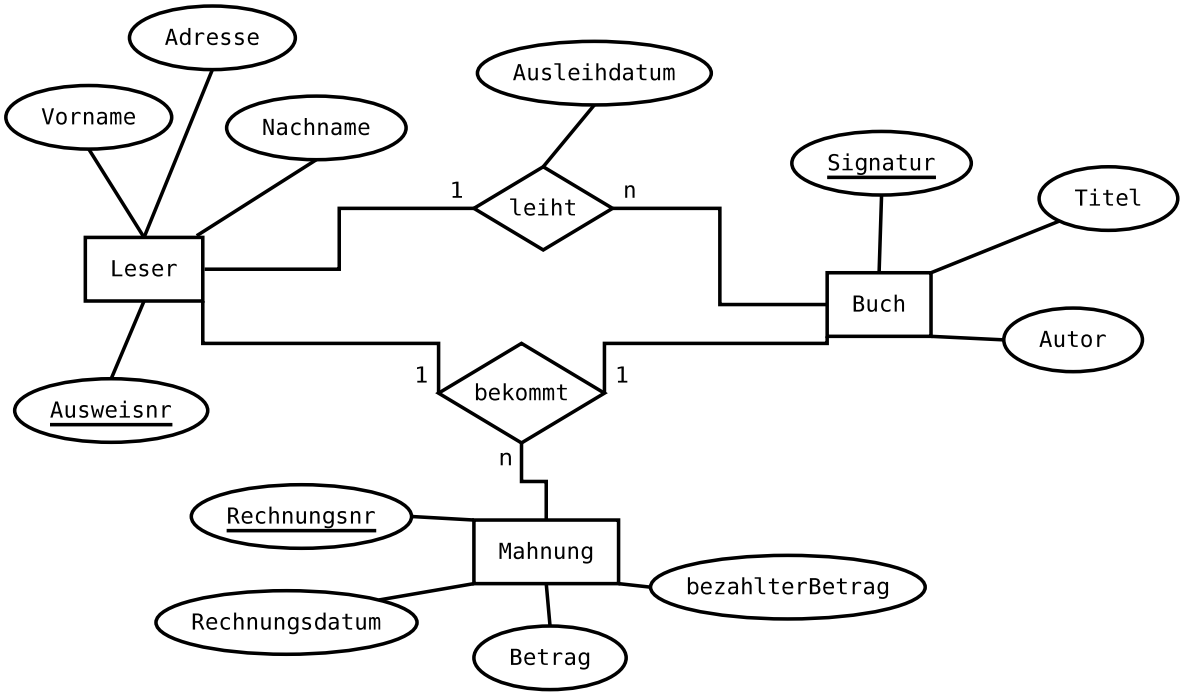

Abbildung 2: ER-Modell „Bibliothek“

an der Beziehung auf der Seite des Lesers eine 1 und auf der Seite des Buches ein n. Dies nennt man die *Kardinalität* der Beziehung.

Weitere mögliche Kardinalitäten sind 1:1 und n:m geben. Darauf gehen wir später ein.[2]

Zu Leser und Buch notieren wir nun noch verschiedene Attribute: Zum Leser gehören mindestens Ausweisnummer und Felder für Namen und Adresse, zum Buch mindestens Signatur, Titel und Autor. Auch die Relationship *leiht* hat ein Attribut, nämlich das Ausleihdatum.

Schließlich gibt es noch eine Entität *Mahnung*. Diese steht in einer Relationship sowohl mit *Leser* (ein Leser bekommt eine Mahnung) als auch mit *Buch* (eine Mahnung bezieht sich auf ein bestimmtes, überzogenes Buch). Nennen wir diese Relationship z.B. *bekommt*. Eine Mahnung hat mindestens die Attribute *Betrag* und *bezahlterBetrag* (beachte die Teilzahlung in der Beschreibung!). Außerdem ist gesetzlich vorgeschrieben, dass Mahnungen eine *Rechnungsnummer* und ein *Datum* benötigen.

Im Unterricht haben wir daraus das ER-Modell in Abb. 2 erarbeitet.

2.2 Primärschlüssel

Jede Entität in einer ER-Modell wird eindeutig durch ein bestimmtes oder durch eine Bestimmte Menge an Attributen repräsentiert. In unserem Beispiel hat jeder Leser eine eindeutige Ausweisnummer, jedes Buch eine eindeutige Signatur und jede Mahnung eine eindeutige Rechnungsnummer. Diese eindeutigen Attribute nennt man *Primärschlüssel* einer Entität.

[2]Genauere Beschreibungen der Kardinalitäten wie z.B. „2:3“ sind in der einfachen Chen-Notation, die wir verwenden, nicht üblich. Dies würde man durch eine n:m-Beziehung darstellen. In anderen Notation, u.A. auch in einer „erweiterten“ Chen-Notation, stellt man solche Nuancen durchaus auch dar.

Es ist meistens ratsam, jeder Entität genau einen Primärschlüssel zuzuweisen. Wie erwähnt, können aber auch mehrere Attribute gemeinsam den Primärschlüssel bilden. Trivial ist, dass zumindest die Kombination aller Attribute einer Entität Primärschlüssel derselben sein *muss*.

Primärschlüssel werden im ER-Diagramm durch Unterstreichen dargestellt.

2.3 Ist das jetzt ein gutes Modell?

Modellierung ist ein kreativer Prozess! Was für Vor- und Nachteile siehst du an unserem Modell einer Bibliothek?

1. Ein wichtiger Nachteil ist sicherlich, dass wir an zwei Stellen das Prinzip der Atomarität verletzen: Ein Attribut speichert mehr als ein „Häppchen“ Daten. Dies ist der Fall bei der *Adresse* des Lesers (diese könnte man sinnvoll aufteilen in Straße, Hausnummer, PLZ und Ort). Auch das Attribut *Autor* des Buches kann man sinnvoll in Vorname und Nachname aufteilen.

2. Ein weiterer Nachteil betrifft die Modellierung des *Autors* als Attribut an sich. Dies ist eine Art vereinfachte 1:1-Beziehung zwischen Buch und Autor. Korrekter wäre jedoch eine n:m-Beziehung: Ein Autor kann beliebig viele Bücher schreiben, und ein Buch kann beliebig viele Autoren haben.

3. Durch die Modellierung der Beziehung *leiht* als 1:n-Beziehung können vergangene Ausleihen nicht ohne weiteres gespeichert werden. Ein Buch kann immer nur von einem Leser geliehen werden – spätestens, wenn es erneut ausgeliehen wird, muss die vergangene Ausleihe aus der Datenbank gelöscht werden, damit die neue abgespeichert werden kann. Mit einer m:n-Modellierung der Beziehung leiht wären auch vergangene Ausleihen ohne Weiteres speicherbar. Je nach Situation kann dies ein Nachteil sein. Es kann aber auch ein Vorteil sein, wenn die Datenbank keine all zu große Datenhaltung insbesondere personenbezogener Daten vorsieht (Stichtwort Datenschutz).

4. Ein Vorteil ist wie oben erwähnt die klare Zuordnung eines einattributigen Primärschlüssels zu jeder Entität.

3 Das relationale Modell

Unser (relationales) Datenbanksystem können wir nicht direkt mit einem Entity-Relationship-Modell füttern. Wir müssen es zuerst in ein relationales Modell – sprich in Tabellenform – umwandeln.

Dabei gehen wir nach folgenden groben Regeln vor:

- Jede Entity im ER-Modell wird zu einer Tabelle im relationalen Modell.
- Alle Attribute einer Entity werden zu Spalten der jeweiligen Tabelle.
- Alle Primärschlüssel im ER-Modell sind auch Primärschlüssel im relationalen Modell.

- Beziehungen der Kardinalität n:m werden durch eine eigene Tabelle umgesetzt.
- Beziehungen der Kardinalität 1:n werden „auf der n-Seite“ umgesetzt: Sie werden in die Tabelle der Entität aufgenommen, zu der nur eine einzelne Entität des anderen Typs gehört.
- Beziehungen der Kardinalität 1:1 führen dazu, dass beide beteiligten Entitäten sowie die Beziehung in einer Tabelle zusammenfallen.

Wie setzen wir eine Relation jetzt aber konkret um? Hier kommen die Primärschlüssel wieder ins Spiel: Eine Zuordnung von einem Buch zu einem Leser können wir sehen als eine Zuordnung ihrer beiden Primärschlüssel – also einer Ausweisnummer zu einer Signatur. Zur Umsetzung der Beziehung *liest* nehmen wir daher den Primärschlüssel der Tabelle Leser – *Ausweisnr* – in die Tabelle Buch mit auf. Einen Primärschlüssel, der in einer fremden Tabelle seine eigene repräsentiert, nennen wir *Fremdschlüssel*. *Ausweisnr* ist also *Fremdschlüssel* in der Tabelle Buch.

Analog gehen wir bei der Beziehung *bekommt* vor, die eine Mahnung einem Leser und einem Buch zuordnet. Auf der „n-Seite“, also in der Tabelle Mahnung, nehmen wir *Ausweisnr* und *Signatur* als Fremdschlüssel der Tabellen Leser und Mahnung auf.

Wir haben in unserem Modell *Bibliothek* also die Folgenden drei Relationen (Tabellen). In Klammern sind die zugehörigen Attribute (Spalten) aufgeführt. Dabei sind Primär- und Fremdschlüssel unterstrichen, Primärschlüssel zusätzlich fettgedruckt.

- Leser (**Ausweisnr**, Vorname, Nachname, Adresse)
- Buch (**Signatur**, Titel, Autor, Ausweisnr, Ausleihdatum)
- Mahnung (**Rechnungsnr**, Rechnungsdatum, Betrag, bezahlterBetrag, Ausweisnr, Signatur)

4 SQL

Wir haben unsere Daten in ein Modell gebracht, dass ein (relationales) Datenbankmanagementsystem (DBMS) wie zum Beispiel *MySQL* versteht. Die Sprache, mit der wir diese fertig aufbereiteten Daten nun in unser DBMS eingeben, ist *SQL*.

Mit SQL werden wir zunächst Tabellen definieren und Daten in diese Tabellen einfügen. Später werden wir SQL auch dazu nutzen, Daten aus der Datenbank abzufragen. Im Rahmen dieser Abfragen werden wir auch lernen, mit SQL Daten neu zu kombinieren und so Informationen zu erhalten, die nur indirekt in der Datenbank enthalten sind.

4.1 Exkurs: Datentypen

Zunächst wiederholen wir jedoch den Begriff des *Datentyps*, der den meisten aus den Programieren-Kursen bekannt sein müsste.

Neben den Informationen, die wir bisher gesammelt haben (was für Relationen gibt es und welche Attribute haben sie) benötigt ein DBMS auch noch die Information, von welchem Typ diese Attribute sind.

Daten können in Rechnern auf verschiedene Arten dargestellt werden. „42“ zum Beispiel kann der Rechner entweder als Ganzzahl, als Gleitkommazahl oder als Text speichern.

- **Ganzzahl:** Die wohl intuitivste Darstellung für „42“ – schließlich handelt es sich um eine ganze Zahl! Der Rechner speichert die Zahl im Dualsystem, also als Bitmuster ab. Lassen wir *42 + 13* berechnen und sowohl 42 als auch 13 liegen als Ganzzahl vor, sollten wir das erwartete Ergebnis 55 erhalten.
  Ganzzahlen werden in SQL (und in vielen anderen Sprachen) als *integer* bezeichnet – die englische Bezeichnung für Ganzzahl.

- **Text:** Tippt ein Benutzer die „42“ über die Tastatur ein, liegt sie zunächst technisch in einem anderen Format vor – nämlich als Zeichen „4“ und Zeichen „2“. Mehrere Zeichen hintereinander bezeichnet man üblicherweise als Text. Eine „Addition“ auf hintereinanderstehen Zeichen gibt es nicht wirklich. *„42“ + „13“* ergibt in den meisten Programmiersprache nicht etwa 55, sondern „4213“ – die Hintereinanderschaltung (Konkatenation) dieser Zeichen.
  Ein Datentyp für Text variabler Länge bietet SQL unter dem Namen *varchar* (englisch etwa „variable number of character“, variable Anzahl Zeichen) an.

- **Gleitkommazahl:** Bisher können wir noch keine Zahlen mit Komma darstellen. Hierfür benutzt man oft den Datentyp Gleitkommazahl (englisch und auch in SQL: „float“)[3]. *42 + 13* ergibt also etwas wie *55.0* (die Darstellungsweise des Ergebnisses variiert), wenn 42 und 13 als Gleitkommazahl vorliegen. In jedem Fall erhalten wir das mathematisch erwartete Ergebnis einer Addition.

4.2 Das Beispiel *Bibliothek* in SQL

Definition der Tabellen

Definieren wir nun also unsere Tabellen für das Beispiel *Bibliothek*. Hierfür bietet SQL den Befehl *CREATE TABLE*, dem der Name der zu erstellenden Tabelle folgt und dahinter in runden Klammern die Liste der zu erstellenden Spalten mit ihren jeweiligen Datentypen. Beim Datentyp *varchar* ist die maximale Anzahl Zeichen dahinter in Klammern anzugeben.

Handelt es sich bei einem Attribut um den Primärschlüssel, notiert man *PRIMARY KEY* hinter Attribut-Name und -Typ.

Unsere Tabelle *Leser* definieren wir also wie folgt:

```
CREATE TABLE Leser (Ausweisnr integer PRIMARY KEY, Vorname
varchar(200), Nachname varchar(200), Adresse varchar(200));
```

[3]Gleikommazahlen sind eine Form Zahlen zu speichern, hinter der deutlich mehr steckt, als nur, dass die Zahl ein Komma enthalten kann. Gleitkomma bedeutet zum Beispiel auch, dass Zahlen nicht exakt gespeichert werden, sondern um so *ungenauer*, je größer sie sind. In diesem Kurs gehen wir auf diese Details aber nicht weiter ein.

SQL bietet neben den erwähnten, grundlegenden Datentypen auch einige spezielle Typen an, die den Umgang mit bestimmten Arten von Daten erleichtern. Für Kalenderdaten existiert der Typ *date*, den wir in den folgenden Tabelle – Buch – verwenden wollen.

Auch Fremdschlüssel können in der Datenbank deklariert werden. Hierzu deklarieren wir das Attribut zunächst ganz normal (also mit Namen und Datentyp), ohne die Fremdschlüssel-Beziehung zu beachten. Nachdem wir in alle Spalten deklariert haben, deklarieren wir dann ein Attribut nachträglich als Fremdschlüssel. Dazu schreiben wir zunächst FOREIGN KEY, dann den Attributname, dann das Schlüsselwort REFERENCES, den Namen der Tabelle, auf die der Fremdschlüssel verweist, und schließlich in Klammern den Namen des Attributs in der ursprünglichen Tabelle, auf das der Fremdschlüssel verweist.

Unsere Tabellen *Buch* und *Mahnung* definieren wir also so:

```
CREATE TABLE Buch (Signatur varchar(5) PRIMARY KEY, Titel
varchar(200), Autor varchar(200), Leser integer, Ausleihdatum date,
FOREIGN KEY (Leser) REFERENCES Leser (Ausweisnr) );

CREATE TABLE Mahnung (Rechnungsnr integer PRIMARY KEY,
Rechnungsdatum date, Betrag float, bezahlterBetrag float, Leser
integer, Buch varchar(5),
FOREIGN KEY (Leser) REFERENCES Leser (Ausweisnr),
FOREIGN KEY (Buch) REFERENCES Buch (Signatur) );
```

Von der Geschichte zu Datensätzen

Viele der textuell beschriebenen Ereignisse in der „Geschichte“ in Abschnitt 2.1 lassen sich direkt in Operationen auf unserer gerade definierten Datenbank umsetzen. Das ist ja auch der Sinn von Datenbanken: Geschäftsprozesse wie Ausleihe eines Buches oder Erstellen einer Mahnung sollen datenverarbeitungstechnisch erfasst werden.

Gehen wir die Geschichte also in chronologischer Reihenfolge durch.

Zunächst werden alle in der Bibliothek vorhandenen Bücher aufgezählt. Für jedes vorhandene Buch müssen wir also einen neuen Datensatz, eine neue Zeile in der Tabelle *Buch* erstellen.

Hierzu verwenden wir den SQL-Befehl *INSERT INTO*. Darauf folgt der Name der gewünschten Tabelle und dahinter in Klammern eine Liste der *Spalten*, in die wir überhaupt Daten einfügen möchten. Es ist nicht nötig, alle Spalten einer Zeile zu füllen – es dürfen auch Felder leer bleiben. Schließlich folgt das Schlüsselwort *VALUES* und dahinter in Klammern die einzufügenden Werte – in der gleichen Reihenfolge, in der wir vorher die Spaltennamen angegeben haben.

In unserem Fall möchten wir nur in die Spalten *Signatur*, *Autor* und *Titel* schreiben. Wir erinnern uns: Die Spalten *Leser* und *Ausleihdatum* repräsentieren die Beziehung *leiht*. Da das Buch noch nicht ausgeliehen ist, ist hier nichts einzutragen.

Das erste Buch, „Per Anhalter durch die Galaxis“, nehmen wir also wie folgt in die Datenbank auf:

```
INSERT INTO Buch (Signatur, Autor, Titel) VALUES ('DA001', 'Douglas
Adams', 'Per Anhalter durch die Galaxis');
```

Anmerkung: Als *Signatur* haben wir hierbei DA001 gewählt. Dies ergibt sich nur zum Teil aus der Geschichte. Laut dieser soll die Signatur mit den Initialen des Autors beginnen (also DA), woran sich eine dreistellige Zahl anschließt. Diese haben wir willkürlich als 001 gewählt.

Bei den weiteren Büchern, „Schöne neue Welt“, „Die Farm der Tiere“ und „Faust“ verfahren wir genauso:

```
INSERT INTO Buch (Signatur, Autor, Titel) VALUES ('AH001', 'Aldous
Huxlex', 'Schöne neue Welt');

INSERT INTO Buch (Signatur, Autor, Titel) VALUES ('GO001', 'George
Orwell', 'Farm der Tiere');

INSERT INTO Buch (Signatur, Autor, Titel) VALUES ('JG001', 'Johann
Wolfgang von Goethe', 'Faust');
```

Wie wir erfahren, ist das Buch „Schöne neue Welt“ zweimal vorhanden. Also müssen wir einen weiteren Datensatz, eine weitere Zeile für das zweite Exemplar in die Tabelle *Buch* einfügen. Diese unterscheidet sich von der Zeile für das erste Exemplar „Schöne neue Welt“ nur duch eine andere Signatur: Bei den beiden Exemplaren handelt es sich um die *gleichen* Bücher (es steht das selbe drin), aber nicht um ein und das *selbe* Buch (beides sind unterschiedliche Dinge – das eine Exemplar könnte zum Beispiel ausgeliehen sein, während das zweite in der Bibliothek steht.

```
INSERT INTO Buch (Signatur, Autor, Titel) VALUES ('AH002', 'Aldous
Huxlex', 'Schöne neue Welt');
```

Nun haben wir eine Benutzeranmeldung: Alice Amann registriert sich bei der Bibliothek. Wir müssen also wieder einen neuen Datensatz, eine neue Zeile erstellen – diesmal in der Tabelle *Leser*:

```
INSERT INTO Leser (Ausweisnr, Vorname, Nachname, Adresse) VALUES
(1, 'Alice', 'Amann', 'Lesestraße 1, 12345 Leserattenstadt');
```

Bei den folgenden Anmeldungen von Bob Bernig und Claire Clepto gehen wir analog vor:

```
INSERT INTO Leser (Ausweisnr, Vorname, Nachname, Adresse) VALUES
(2, 'Bob', 'Bernig', 'Lesestraße 1, 12345 Leserattenstadt');

INSERT INTO Leser (Ausweisnr, Vorname, Nachname, Adresse) VALUES
(3, 'Claire', 'Clepto', 'Lesestraße 1, 12345 Leserattenstadt');
```

Damit haben wir den Leser- und Bücherbestand in der Datenbank angelegt.

Chronologisch geht es weiter mit Bobs Ausleihe von „Schöne neue Welt“ am 01.01.2012. Die Geschichte definiert nicht, welches der beiden Exemplare Bob leiht. Entscheiden wir uns, ihm das erste Exemplar, das mit der Signatur *AH001*, auszuleihen.

Wir erinnern uns, dass wir die Beziehung *leiht* in der Tabelle *Buch* umgesetzt haben. Also müssen wir den entsprechenden Eintrag in der Tabelle *Buch* für das Buch mit der Signatur

AH001 ändern: Das Feld *Leser* muss auf die Ausweisnummer von Bob (2), das Feld *Ausleihdatum* auf den 01.01.2012 gesetzt werden.

Dies tun wir mit dem SQL-Befehl *UPDATE*, gefolgt von dem Namen der Tabelle, in der wir etwas ändern möchten. Anschließend notieren wir *SET* gefolgt von dem Namen der Spalte, in der wir etwas ändern möchten. Nach einem Gleichheitszeichen notieren wir schließlich den gewünschten Wert. Mehrere dieser *SET*-Klauseln können durch Komma getrennt hintereinander geschrieben werden, um so mehrere Spalten gleichzeitig zu ändern.

Wichtig: Der *Update*-Befehl muss nun noch auf eine bestimmte Zeile eingeschränkt werden. Täten wir dies nicht, würde das DBMS in **allen** Zeilen der Tabelle die angegebenen Spalten ändern. Diese Einschränkung erfolgt mit einer *WHERE*-Klausel. Auf das Schlüsselwort *WHERE* folgt eine beliebige Bedingung. Die Formulierung von Bedingungen sollte aus dem Kurs „Grundlagen der Programmierung“ bekannt sein. In diesem Fall lautet sie *Signatur = 'AH001'*.

Der Update-Befehl fürs Bobs Ausleihe lautet also:

```
UPDATE Buch SET Leser=2, Ausleihdatum='2012-01-01' WHERE
Signatur='AH001';
```

Chronologisch folgt nun Claires Ausleihe von „Schöne neue Welt“ (logischerweise nun das zweite Exemplar mit der Signatur *AH002*) sowie „Farm der Tiere“ am 03.01.2012:

```
UPDATE Buch SET Leser=3, Ausleihdatum='2012-01-03' WHERE
Signatur='AH002';
UPDATE Buch SET Leser=3, Ausleihdatum='2012-01-03' WHERE
Signatur='GO001';
```

Am 14.02.2012 bekommt Bob eine Mahnung über fünf Euro, da er sein Buch „Schöne neue Welt“ immer noch nicht zurückgegeben hat. Wir müssen also einen neuen Datensatz in der Tabelle *Mahnung* erstellen:

```
INSERT INTO Mahnung (Rechnungsnr, Rechnungsdatum, Betrag, Leser,
Buch) VALUES (1, '2012-01-14', 5, 2, 'AH001');
```

Einen Tag später, also am 15.02.2012 gibt Bob zunächst sein Buch zurück. Wir löschen die Ausleihe des Buches, indem wir die Felder *Leser* und *Ausleihdatum* auf *NULL* setzen[4]:

```
UPDATE Buch Set Leser = NULL, Ausleihdatum = NULL WHERE
Signatur='AH001';
```

Außerdem bezahlt Bob seine Mahnung über fünf Euro. Dies stellen wir dar, indem wir in dem Datensatz seiner Mahnung das Feld *bezahlterBetrag* auf 5 setzen – aber bitte nur für die Mahnung mit der Rechnungsnummer 1 und nicht etwa für alle im System möglicherweise vorhandenen Mahnungen verschiedener Leser! Auch hier ist die *WHERE*-Klausel also wichtig.

[4] *NULL* steht für „leeres Feld“. Datenbanksysteme unterscheiden zwischen dem Zahleneintrag *0* und dem leeren Feld *NULL*.

```
UPDATE Mahnung SET bezahlterBetrag=5 WHERE Rechnungsnr = 1;
```

Bob leiht sich nun erneut ein Buch – „Per Anhalter durch die Galaxis“. Mit dem Anlegen von Ausleihen sind wir bereits vertraut:

```
UPDATE Buch SET Leser=2, Ausleihdatum='2012-01-15' WHERE
Signatur='DA001');
```

Chronologisch folgend erhält Claire am 17.01.2012 für jedes ihrer beiden ausgeliehenen Bücher jeweils eine Mahnung über je 5 Euro. Auch das Anlegen von Mahnungen ist für uns nicht mehr neu:

```
INSERT INTO Mahnung (Rechnungsnr, Rechnungsdatum, Betrag, Leser,
Buch) VALUES (2, '2012-01-17', 5, 3, 'AH002');
INSERT INTO Mahnung (Rechnungsnr, Rechnungsdatum, Betrag, Leser,
Buch) VALUES (3, '2012-01-17', 5, 3, 'GO001');
```

Fünf Tage nach seiner Ausleihe – am 20.01.2012 – gibt Bob „Per Anhalter durch die Galaxis‘ zurück:

```
UPDATE Buch SET Leser = NULL, Ausleihdatum = NULL WHERE
Signatur='DA001');
```

Am 03.02.2012 erhält Claire einen zweiten Satz Mahnungen, da sie ihre Bücher immer noch nicht wieder zurückgegeben hat – für jedes Buch muss sie nun weitere 10 Euro zusätzlich zahlen:

```
INSERT INTO Mahnung (Rechnungsnr, Rechnungsdatum, Betrag, Leser,
Buch) VALUES (4, '2012-02-03', 10, 3, 'AH002');
INSERT INTO Mahnung (Rechnungsnr, Rechnungsdatum, Betrag, Leser,
Buch) VALUES (5, '2012-02-03', 10, 3, 'GO001');
```

Claire versucht daraufhin, die Bibliothek zu verwirren, indem sie 42 Cent überweist. Wir verbuchen diesen Scherz jedoch einfach als Teilzahlung auf die erste der vier Mahnungen, die wir ihr geschickt haben:

```
UPDATE Mahnung SET bezahlterBetrag=0.42 WHERE Rechnungsnr = 2;
```

Zum Schluss leiht Alice am 12.02.2012 „Per Anhalter durch die Galaxis“ aus:

```
UPDATE Buch SET Leser=1, Ausleihdatum='2012-02-12' WHERE
Signatur='DA001';
```

Damit haben dir die Daten aus der „Geschichte“ vollständig in unserem Datenbankmodell umgesetzt – in der Form und in dem Detailsgrad, wie wir es bei der Modellierung festgelegt haben (siehe Abschnitt 2.3 zur Diskussion über unser Modell).

Abfragen von Datensätzen

Kommen wir nun zum eigentlichen Anwendungszweck einer Datenbank: Aus den gespeicherten Daten sollen bestimmte Informationen extrahiert und ausgegeben werden. Die folgenden Informationen wollen wir in diesem Abschnitt der Datenbank entnehmen:

1. Der komplette Inhalt der Tabelle Buch
2. Titel aller Bücher
3. Autor und Titel aller Bücher
4. Die Signatur beider Exemplare von „Schöne neue Welt“
5. Die Ausweisnummer des aktuellen Lesers von „Per Anhalter durch die Galaxis“
6. Alle Mahnungen – in der Form: Signatur des Buches – Ausweisnummer des Lesers – Betrag
7. Alle Mahnungen von 5 Euro – in der Form: Signatur des Buches – Ausweisnummer des Lesers – Betrag

Datenbankabfragen führen wir mit dem SQL-Befehl *SELECT* durch. Nach diesem Schlüsselwort notieren wir zuerst, aus welcher Spalte wir Informationen entnehmen möchten. Anschließend notieren wir das Schlüsselwort *FROM* und den Namen der Tabelle, aus der die Informationen kommen sollen.

Für Abfrage 1, der komplette Inhalt der Tabelle Buch, notieren wir statt den Namen der gewünschten Spalten einfach ein Sternchen (*). Das steht für „alle Spalten”. Der Abfrage sieht wie folgt aus:

```
SELECT * FROM Buch;
```

Übersetzt: *Zeige* („SELECT”) *alle Spalten* („*”) *aus* („FROM”) *der Tabelle Buch* („Buch”)

In Abfrage 2, die Titel aller Bücher, interessiert uns dagegen nur noch eine Spalte der Tabelle Buch. Wir schreiben sie wie folgt:

```
SELECT Titel FROM Buch;
```

Wir können auch gleichzeitig Informationen aus mehreren Spalten entnehmen. Dazu notieren wir die gewünschten Spalten einfach durch Kommata getrennt. Die dritte Abfrage, Autor und Titel aller Bücher, sieht also so aus:

```
SELECT Autor, Titel FROM Buch;
```

Mit der vierten Abfrage, Signatur beider Exemplare von „Schöne neue Welt“, wollen wir Informationen aus der Spalte *Signatur* der Tabelle *Buch* entnehmen – jedoch nicht aus allen Zeilen. Wir müssen daher, wie wir es bereits gelernt haben, die Abfrage mit einer *WHERE*-Klausel einschränken:

```
SELECT Signatur FROM Buch WHERE Titel='Schöne neue Welt';
```

Auch mit der fünften Abfrage, Ausweisnummer des aktuellen Lesers von „Per Anhalter durch die Galaxis“, entnehmen wir der Tabelle *Buch* Informationen aus einer Spalte und schränken die gewünschten Zeilen mit einer *WHERE*-Klausel ein:

```
SELECT Leser FROM Buch WHERE Titel='Per Anhalter durch die
Galaxis';
```

Mit der sechsten Abfrage, alle Mahnungen, entnehmen wir Informationen aus der Tabelle Mahnung, ohne eine Einschränkung der heranzuziehenden Zeilen vorzunehmen. Sie ähnelt also in gewisser Weise der dritten Abfrage.

```
SELECT Buch, Leser, Betrag FROM Mahnung;
```

Mit der siebten und letzten Abfrage, alle Mahnungen von 5 Euro, schränken wir schließlich auch diese Abfrage mit einer *WHERE*-Klausel ein:

```
SELECT Buch, Leser, Betrag FROM Mahnung WHERE Betrag=5;
```

5 Komplexe Abfragen: Der Join

Bisher haben wir nur gelernt, wie wir Daten aus einer einer einzelnen Tabelle abfragen. Oft benötigt man aber Informationen aus verschiedenen Tabellen gleichzeitig – eine bestimmte Verknüpfung von Daten. Auch das können wir mit Datenbanksystemen erledigen: Das DBMS stellt uns Daten in jeder von uns gewünschten Verknüpfung neu zusammen (es betreibt also Datenverarbeitung!).

Die Notwendigkeit, Daten aus verschiedenen Tabellen zu kombienieren, ergibt sich insbesondere auch deshalb, dass wir durch die systematische Modellierung Daten getrennt (auf mehrere Tabellen aufgeteilt) haben, die man intuitiv gemeinsam ablegen würde.

Beispiel: Würden wir unser Bibliothekssystem nicht als Datenbank sondern (intuitiv) mit einem Tabellenkalkulationsprogramm (als „Excel-Tabelle”) erstellen, hätten wir sicher den Ansatz, nebeneinander den Namen eines Buches und den Namen des aktuellen Entleihers zu notieren. In Kap. 2, als wir das Entity-Relationship-Modell zu unserem Bibliothekssystem entwickelt haben, haben wir uns aber (sinnvollerweise, um Ziele wie z.B. Redundanzfreiheit zu erreichen) darauf festgelegt, die Namen von Büchern und die Namen von Lesern in unterschiedlichen Tabellen zu speichern.

Kombiniere, Datenbank!

Wir wissen bereits, das wir mit einem *SELECT*-Befehl nicht nur Daten aus einer Spalte auslesen können. Wir können auch mit Kommata getrennt mehrere Spalten auf einmal auslesen. Beide folgenden SQL-Befehle sind also in unserer Datenbank „Bibliothekssystem” gültig – der erste liest Daten aus nur einer Spalte aus, der zweite aus dreien:

```
SELECT Autor FROM Buch;
SELECT Vorname, Nachname, Adresse FROM Leser;
```

Analog ist es auch möglich, hinter dem Schlüsselwort FROM mehrere Tabellen durch Kommata getrennt anzugeben. Das alleine führt allerdings zu einem Ergebnis, das zunächst verwirrend aussieht und in den wenigsten Fällen gewollt ist.

```
SELECT * FROM Buch, Leser;
```

Das Datenbanksystem tut, was wir ihm gesagt haben: Es kombiniert die Einträge der Tabelle Buch mit denen der Tabelle Leser – jeden mit jedem! Jedes in der Datenbank vorhandene Buch wird mit jedem in der Datenbank vorhandenen Leser zu je einer Zeile kombiniert. Betrachtet man die Tabellen als Relationen (mathematische Mengen), sagt man, wir haben das *Kreuzprodukt* der Relationen Buch und Leser erstellt.

Was so sinnlos aussieht, ist aber der Anfang der durchaus richtigen Vorgehensweise: Diese stumpfe Kombination zweier Tabellen filtern wir jetzt so, dass nur wirklich zusammen gehörende Zeilen wirklich kombiniert werden. Einen solchen Filter erstellen wir mit der uns bekannten *WHERE*-Klausel.

Jedem Join sein WHERE

Welche Datensätze in zwei Tabellen wirklich zusammen gehören, verraten uns die *Schlüssel*. Wir erinnern uns: Eine Beziehung zwischen zwei Tabellen (eine logische Ebene höher bedeutet das: zwischen zwei Entitäten) haben wir realisiert, indem wir den Primärschlüssel einer Tabelle als Fremdschlüssel in die zweite Tabelle aufgenommen haben (siehe Kapitel 3).

Zwei Zeilen in zwei verschiedenen Tabellen gehören also dann zusammen, wenn eine bestimmte Spalte der einen Tabelle – nämlich ihr Primärschlüssel – gleich einer bestimmten Spalte der zweiten Tabellen ist – nämlich dem entsprechenden Fremschlüssel.

In unserem Beispiel, den Tabellen Buch und Leser unseres Bibliothekssystems, haben wir den Primärschlüssel *Ausweisnr* der Tabelle Leser unter dem Namen *Leser* als Fremschlüssel in die Tabelle Buch aufgenommen (in der Tabelle Buch steht die Ausweisnummer des aktuellen Lesers). Die WHERE-Klausel zu diesem Join muss also „WHERE Ausweisnr = Leser" lauten.

Der fertige Join sieht in unserem Beispiel also so aus:

```
SELECT * FROM Buch, Leser WHERE Ausweisnr = Leser;
```

Zu jedem Join aus zwei Tabellen brauchen wir eine solche WHERE-Bedingung. Analog zu diesem Beispiel können wir auch einen Join aus drei verbundenen Tabellen erstellen: Hierzu bräuchten wir zwei WHERE-Bedingungen: Je eine für beiden Verbindungen zwischen den Tabellen.

6 Relationenalgebra

Wie wir bereits in Kapitel 1.1 kurz erwähnt haben, ist unsere Sicht auf Datenbanken als Sammlung von Tabellen, wie sie auch auf Papier stehen könnten, nur eine vereinfachte Anschauung. Relationalen Datenbanken liegt ein mathematisches, mengentheoretisches Modell zu Grunde: Die relationale Algebra.

Dieses Thema wirkt vielleicht sehr theoretisch und kompliziert. Tatsächlich ist abstraktes Denken gefragt! Allerdings behandeln wir hier nichts, was wir nicht bisher schon gemacht haben – nur haben wir es eben in einer anderen Darstellung gemacht, nämlich SQL. Vielleicht hilft es beim Verstehen, dass aus all der jetzt folgenden Mathematik am Ende im Großen und Ganzen das dabei heraus kommen muss, was wir schon können.

6.1 Relationen anschaulich

Klären wir zunächst den Grundbegriff des Ganzen: Eine Relation (von lat. *relatio*: „das Zurücktragen“) bezeichnet eine Beziehung zwischen verschiedenen „Dingen“ (man könnte auch wieder den Begriff der *Entität* bemühen). Relationen können hierbei ganz alltägliche Dinge beschreiben, wie zum Beispiel Beziehungen zwischen Personen.

Betrachten wir zum Beispiel Beziehungen, die zwischen den Personen Alice, Bob, Claire und Pika bestehen. Diese fünf Personen bilden eine **Menge**, die wir zum Beispiel als P bezeichnen und wie folgt notieren:

$P = \{Alice, Bob, Claire, Pika\}$

Eine Menge ist in der Mathematik also einfach eine Sammlung verschiedener Objekte, zum Beispiel eben Personen. Dabei hat eine Menge keine innere Ordnung. Es ist also egal, ob wir Alice oder Pika zuerst aufzählen:

$P = \{Alice, Bob, Claire, Pika\} = \{Pika, Bob, Alice, Claire\}$

Eine **Relation** über der Menge P beschreibt nun Beziehungen zwischen den verschiedenen Elementen aus P. Eine solche Relation könnte zum Beispiel die Relation „mag[5]“ sein: „Person Y mag Person Z“. In unserem Beispiel: Alice mag Bob, Claire mag Alice und Pika mag sich selbst. Eine solche Relation können wir problemlos in eine Tabelle fassen, wie wir sie auch in ein Datenbanksystem füttern könnten:

mag:

Y	Z
Alice	Bob
Claire	Alice
Pika	Pika

Alice steht also zu Bob in der Relation „mag“, genauso wie Claire zu Alice, und Pika zu sich selbst. Bob steht dagegen *nicht* zu Alice in der Relation „mag“.

Eine Relation ist mathematisch gesehen eine besondere Menge. Inhalt dieser Menge sind die drei beschriebenen Beziehungs-Paare, wie wir sie auch jeweils durch eine Tabellenzeile dargestellt haben.

[5]Es ist in der Mathematik üblich, die Namen von Variablen auf einzelne Buchstaben zu begrenzen – und da die Namen von Relationen Variablen („Relvars“) sind, auch diese. Wir brechen der Lesbarkeit halber mit dieser Konvention und verwenden Variablennamen, wie sie in der Programmierung üblich sind.

Diese drei der Relation angehörenden Paare – *(Alice, Bob)*, *(Claire, Alice)* und *(Pika, Pika)* – sind auch wieder so etwas ähnliches wie Mengen: Beides sind wieder Sammlungen von irgendwelchen Objekten, in diesem Fall von jeweils zwei Personen. Sie unterscheiden sich von Mengen aber dadurch, das sie geordnet sind. Das Paar *(Alice, Bob)* – es steht in unserem Fall für die Beziehung „Alice mag Bob“ – ist nicht das gleiche, wie das Paar *(Bob, Alice)* – denn vielleicht mag Bob die Alice ja gar nicht zurück. Solche geordneten Paare notiert man genau so, wie wir das gerade getan haben (ihre Elemente in runden Klammern durch Kommata getrennt).

Die *Relation* „mag“ ist also eine Menge, die aus drei Paaren besteht, die wiederum aus jeweils zwei Objekten des Typs Person besteht:

mag = {*(Alice, Bob), (Claire, Alice), (Pika, Pika)*}

Eine solche Relation, deren Elemente Paare sind, nennt man *zweistellige Relation* oder *binäre Relation* (von lat. *binarius*: „zweifach“).

6.2 Mehrstellige Relationen

Es gibt auch Relationen, die nicht aus Paaren bestehen. Bleiben wir bei unserem Beispiel und betrachten eine weitere Relation über der Menge P unserer vier Personen. *denkt* bezeichne die Relation „Person X denkt, dass Person Y Person Z mag“. Sie ist hier in Tabellenform abgebildet:

denkt:

X	**Y**	**Z**
Alice	Bob	Pika
Claire	Alice	Bob
Claire	Claire	Alice
Pika	Pika	Pika

Alice denkt also, dass Bob Pika mag. Claire denkt zum einen, dass Alice Bob mag. Zum anderen denkt – und weiß – sie, dass sie selbst Alice mag. Pika mag sich selbst und denkt das auch!

Die einzelnen Elemente (= Zeilen) der Relation (= Tabelle) sind jetzt also nicht mehr geordnete Paare, sondern geordnete Listen aus drei Dingen. Diese bezeichnet man als **Tupel**, in diesem Fall als **3-Tupel**.

Mathematisch stellt man die Relation „Person X denkt, dass Person Y Person Z mag“, der wir den Namen *denkt* gegeben haben, also so dar:

denkt = {*(Alice, Bob, Pika), (Claire, Alice, Bob), (Claire, Claire, Alice), (Pika, Pika, Pika)*}

6.3 Relationen über verschiedenen Mengen

Beide Beispiele, die wir bisher betrachtet haben, waren Relationen über nur eine Menge: Zueinander in Relation gesetzt waren nur Elemente der selben Menge – nämlich Personen zu anderen Personen. Relationen können aber auch über verschiedenen Mengen gebildet werden.

Führen wir als zweite Beispielmenge folgende Menge von Bahnhöfen ein, die wir mit B bezeichnen:

B = {*Frankfurt HBf, Frankfurt Süd, Hamburg-Altona, Rheinberg*}

Folgendes sei nun die Relation „faehrtNach“ über den Mengen P und B:

faehrtNach:

Reisender	**Fahrziel**
Bob	Frankfurt Süd
Pika	Rheinberg
Claire	Rheinberg

Die Relation *faehrtNach* beschreibt also, dass die Person Bob zum Bahnhof Frankfurt Süd fährt. Außerdem fahren Pika und Claire beide nach Rheinberg und wundern sich, dass dort alles so klein ist[6].

Die Relation *faehrtNach* in mathematischer Notation:
faehrtNach = {*(Bob, Frankfurt Süd), (Pika, Rheinberg), (Claire, Rheinberg)*}

6.4 Überblick: Mathematische Begriffe

Bevor es weiter geht, fassen wir kurz anhand einiger Beispiele zusammen, welche Begriffe der Relationenalgebra wir bisher kennen:

{*Apfel, Birne, Banane*}	**Menge**	Eine *ungeordnete* Ansammlung verschiedener Objekte (verschieden = ohne Doppelungen).
(Apfel, rot, hart)	**Tupel**	Eine *geordnete* Liste von Objekten (die auch gleich sein könnten). Die Anzahl der Elemente wird dem Begriff oft vorangestellt: Das hier ist ein *3-Tupel*.
(Apfel, rot)	**Paar**	Ein spezielles Tupel, nämlich ein 2-Tupel. Funktioniert genau wie andere Tupel auch!
{*(Apfel, rot), (Birne, grün), (Banane, gelb)*}	**Relation**	Eine Menge von Tupeln, deren Elemente „spaltenweise“ (da ist die Tabellensicht wieder!) den gleichen Mengen entstammen.

6.5 Definition über das Kreuzprodukt

In den letzten Abschnitten haben wir uns anschauliche Beispiele für Relationen angesehen. Eine korrekte mathematische Definition der Relation steht aber noch aus.

Zunächst gehen wir zurück zum Begriff der *Menge*. Auf Mengen ist die Rechenoperation *Kreuzprodukt* definiert, eine Art „Multiplikation“ von zwei Mengen miteinander: Das Kreuzprodukt zweier Mengen A und B besteht aus der paarweisen Kombination aller Elemente von A und B miteinander („ jedes mit jedem“).

Bilden wir also das Kreuzprodukt $P \times B$, alle Personen kombiniert mit allen Bahnhöfen:

$P \times B$ = {*(Bob, Frankfurt HBf), (Bob, Frankfurt Süd),*
(Bob, Hamburg-Altona), (Bob, Rheinberg),
(Alice, Frankfurt HBf), (Alice, Frankfurt Süd),
(Alice, Hamburg-Altona), (Alice, Rheinberg),

[6] `http://www.youtube.com/watch?v=JQi2kFCDHro`

(Claire, Frankfurt HBf), (Claire, Frankfurt Süd),
(Claire, Hamburg-Altona), (Claire, Rheinberg),
(Pika, Frankfurt HBf), (Pika, Frankfurt Süd),
(Pika, Hamburg-Altona), (Pika, Rheinberg)}

Es fällt vielleicht auf, dass das Kreuzprodukt $P \times B$ gewisse Ähnlichkeiten mit unserer Relation *faehrtNach* (die ja über P und B definiert ist) hat: Beides sind Mengen, die aus geordneten Paaren bestehen, wobei jeweils das erste Element eines Paares eine Person und das zweite ein Bahnhof ist. Tatsächlich enthält die Menge $P \times B$ alle möglichen Elemente der Relation *faehrtNach* – nur, das manche Elemente von $P \times B$ eben *nicht* auch zur Relation *faehrtNach* gehören.

Man definiert: Eine Relation[7] besteht aus allen Elementen des Kreuzprodukts dieser Mengen, die eine bestimmte Bedingung erfüllen. Die Relation ist also eine **bestimmte Teilmenge des Kreuzprodukts**. Der Vollständigkeit halber stellen wir dies nun noch in der üblichen Notation dar, auch wenn wir sie hier nicht weiter besprechen:

$R = \{(a_1, a_2, ..., a_n) \in A_1 \times A_2 \times ... \times A_n | Relationsbedingung\}$

6.6 Relationen, Tabellen und SQL

Was wir bisher kennen gelernt haben, ist der mathematische Begriff der Relation, wie er sich logisch aus der Mengenlehre ergibt. So weit ist das Mathematik, die zunächst einmal für sich steht und nicht unbedingt etwas mit Datenbanken zu tun haben muss.

Als die Informatik den Begriff adaptiert hat, hat sie ein weiteres Detail hinzugefügt, dass sofort an etwas erinnert, was in unserem Bild der Relation als Tabelle bisher fehlt: Die Spaltenüberschriften.

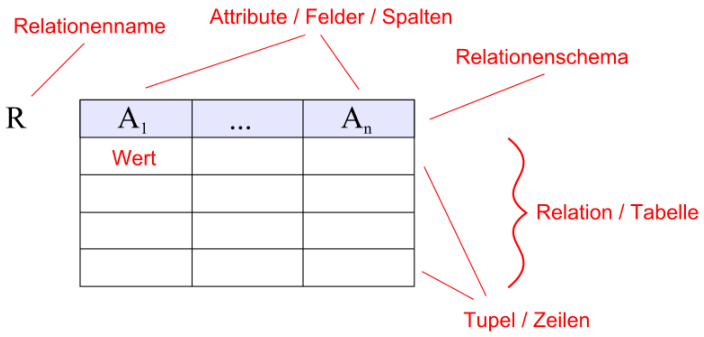

Abbildung 3: Begriffe relationaler Datenbanken
Bild: „Fragment“ / Wikipedia

In der Informatik wird der Begriff der Relation gegenüber der rein mathematischen Variante erweitert: Man definiert, dass zur eigentlichen Relation, wie wir sie bisher kennen gelernt haben (Menge von Tupeln bzw. „Zeilen“) noch ein **Relationenschema** dazukommt.

Ein Relationenschema ist ein Tupel (also eine geordnete Liste) von Namen, die man als **Attribute** bezeichnet. Wie gesagt ist die Verbindung zu unserer Tabellenwelt denkbar einfach: Das Relationenschema ist die Kopfzeile der Tabelle und jedes Attribut darin ist eine Spaltenüberschrift.

[7] In diesem Script werden nur endliche Relationen erwähnt. In der Mathematik werden auch unendliche Relationen betrachtet, die für Datenbanken aber nicht relevant sind.

Abbildung 3 fasst noch einmal alle von uns verwendeten Begriffe der relationalen Algebra und ihre Tabellen-Anschauung zusammen.

Schlagen wir von hier wieder den Bogen zu unserem alten Bekannten SQL: Wir wissen bereits, dass es sich bei jeder „Tabelle“ in unseren Datenbanksystemen um eine Relation handelt. Aber auch das Ergebnis jeder *SELECT*-Abfrage ist eine Relation. Nehmen wir als Beispiel wieder unsere Relation *faehrtNach* aus Abschnitt 6.3: Diese könnte so 1:1 in einem Datenbanksystem abgelegt sein. Trivial ist, dass bei folgender *SELECT*-Abfrage eine Relation heraukommt:
`SELECT * FROM faehrtNach;`

Schließlich gibt dieser Befehl, wie wir wissen, die *gesamte* Tabelle *faehrtNach* aus – also genau die uns bekannte Relation. Hinter einem *FROM* notieren wir also immer den Namen einer Relation, die wir anzeigen lassen möchten.

Der Befehl
`SELECT Reisender FROM faehrtNach;`
ergibt ebenfalls eine Relation aus – nämlich die Relation *faehrtNach* mit nur dem Attribut *Reisender*, ohne das Attribut *Fahrziel*. Hinter *SELECT* können wir also die anzuzeigenden Attribute („Spalten“) bestimmen.

Mit dem Befehl
`SELECT * FROM faehrtNach WHERE Reisender='Pika';`
erhalten wir die Relation *faehrtNach* mit nur den Tupeln („Zeilen“), in denen das Attribut *Reisender* den Wert *Pika* hat. Alle anderen Tupel, bei denen diese Bedingung nicht erfüllt ist, werden ausgeblendet.

Auch diese informationsverarbeitenden Operationen – wie Auswahl bestimmter Attribute oder Tupel aus einer Relation – lassen sich in der relationalen Algebra mathematisch beschreiben. Darum geht es im nächsten Abschnitt.

6.7 Basisoperationen

Genauso, wie auf Zahlen bestimmte Basisoperationen definiert sind, wie zum Beispiel Addition und Multiplikation, kann man auch mit Relationen auf eine spezielle Weise „rechnen“.

Ähnlich, wie wir das z.B. von den natürlichen Zahlen kennen, sind bestimmte Rechenoperationen grundlegend, während sich andere durch diese grundlegenden Operationen konstruieren lassen. Auf den natürlichen Zahlen ist z.B. die Addition eine solche Basisoperation, während sich z.B. die Fakultät aus einfacheren Operationen zusammensetzen lässt.

Wir klären nun zunächst, welche Basisoperationen auf Relationen definiert sind. Anschließend werden wir untersuchen, wie wir aus diesen Basisoperationen eine komplexe Operation, nämlich den Join, zusammensetzen können.

Mengenoperationen

Da Relationen besondere Mengen sind, lassen sich die üblichen Mengenoperationen auf sie übertragen. Als Basisoperationen sind die *Vereinigung* und die *Mengendifferenz* definiert.

Das Ergebnis der **Vereinigung** mehrerer Mengen ist eine Menge, die aus allen Elementen der ursprünglichen Mengen besteht. Auf Grund der Definition des Begriffs Menge (Sammlung

verschiedener Objekte) fallen hierbei jedoch „Doppelungen“ heraus. Die Vereinigung unserer Beispielmengen P und B ergibt also:

$P \cup B = \{$*Pika, Bob, Alice, Claire, Frankfurt HBf, Frankfurt Süd, Hamburg-Altona, Rheinberg*$\}$

Eine Vereinigung mehrerer *Mengen* ist immer möglich. Eine Vereinigung mehrerer *Relationen* dagegen nicht: Sie ist nur möglich, wenn die beteiligten Relationen die gleichen *Attribute* haben. Man spricht davon, dass die Relationen *vereinigungskompatibel* sein müssen. Diese Bedingung ist anschaulich klar, wenn man sich Relationen wieder als Tabellen vorstellt: Ich kann nur dann von zwei Tabellen einfach alle Zeilen untereinanderschreiben, wenn die Spaltenüberschriften übereinstimmen.

Wir haben sie nicht besprochen, aber auch in SQL ist die Vereinigung unter dem Schlüsselwort *UNION* verfügbar: Mit ihm lassen sich zwei geeignete *SELECT*-Abfragen miteinander verbinden.

Beim Bilden der **Mengendifferenz** zweier Mengen werden die abzuziehenden Elemente der Ursprungsmenge entnommen. Dies entspricht anschaulich dem Streichen bestimmter Zeilen aus einer Tabelle. Auch hierfür müssen die beteiligten Mengen *vereinigungskompatibel* sein.

Schließlich ist auch das bereits bei der formalen Definition der Relation angesprochene **Kreuzprodukt** eine Mengenoperation, die auch auf Relationen definiert ist. Sie ist immer möglich. Das Kreuzprodukt kennen wir in SQL bereits – wir haben es in Abschnitt 5, als wir den Join eingeführt haben, als „abschreckendes Beispiel“ erwähnt, um die Notwendigkeit einer *WHERE*-Bedingung klarzumachen. Folgender SQL-Befehl bildet also das Kreuzprodukt aus den beiden Relationen R und S:

```
SELECT * FROM R, S;
```

Diese drei Operationen sind *binäre Operationen*: Sie arbeiten auf (jeweils) zwei Mengen bzw. verbinden diese. Binäre Operationen sind uns bestens bekannt: Auch beispielswiese die Addition oder Multiplikation auf ganzen Zahlen sind binär („eine Zahl + noch eine Zahl“).

Wir werden uns nun zwei Operationen ansehen, die weder aus der Mengenlehre „geerbt“ wurden noch binär sind.

Projektion

Die Projektion reduziert eine Relation auf bestimmte Attribute. Anschaulich kann man sich vorstellen, dass man von einer Tabelle bestimmte Spalten einfach wegklappt: Das Ergebnis sind die Spalten, die übrig bleiben – auf diese wird *projiziert*. Alle Zeilen der Tabelle bleiben dagegen erhalten.

Das Symbol für die Projektion ist das kleine Pi (π). Man notiert sie wie folgt:
$Ergebnis = \pi_{a_1,a_2,\ldots,a_n}(R)$

Dabei ist a_1 bis a_n die Liste der Attribute („Spalten“), die erhalten bleiben sollen – alle anderen fallen weg. R ist die ursprüngliche Relation.

Dies entspricht dem *SELECT* in SQL. Die gleiche Projektion lautet in SQL:
SELECT a_1, a_2, ..., a_n FROM R;

Um unser Beispiel aus Abschnitt 6.6 zurückzukommen, in dem wir die mathematische Sicht auf Relationen mit der Welt der Tabellen und SQL verglichen haben: Dort sagten wir, wir können mit dem Befehl
`SELECT Reisender FROM faehrtNach;`
die Relation faehrtNach auf ihr Attribut *Reisender* einschränken. Dies ist also eine *Projektion* und wird in relationaler Algebra so dargestellt:
$Ergebnis = \pi_{Reisender}(faehrtNach)$

Die Projektion ist eine *unäre* Operation: Sie bearbeitet nur eine einzige Relation. Auch dieses Verhalten kennen wir von den ganzen Zahlen, z.B. durch das Vorzeichen: Ein Minuszeichen vor einer Zahl bearbeitet nur diese eine Zahl – negiert sie.

Selektion

Die Selektion reduziert eine Relation auf bestimmte Tupel, entfernt also bestimmte Elemente der Relation. Dies ist ähnlich der *Mengendifferenz*, die wir bereits kennengelernt haben. Die Selektion entfernt jedoch nicht all die Elemente aus der Ursprungsrelation, die sich in einer zweiten Relation befinden, sondern entfernt Elemente, die eine bestimmte *Bedingung* nicht erfüllen. Sie ist im Gegensatz zur Mengendifferenz also keine binäre Operation, sondern wie die Projektion unär.

Das Symbol für die Selektion ist das kleine Sigma (σ). Man notiert sie wie folgt:
$Ergebnis = \sigma_{\varphi}(R)$
Das kleine Phi (φ) steht dabei für die Selektionsbedingung.

Dies entspricht dem *WHERE* in SQL. Die gleiche Projektion lautet in SQL:
`SELECT * FROM R WHERE φ;`

Kommen wir wieder zurück auf unser Beispiel aus Abschnitt 6.6: Dort sagten wir, wir können mit dem Befehl
`SELECT * FROM faehrtNach WHERE Reisender='Pika';`
die Relation faehrtNach auf alle Tupel einschränken, bei denen das Attribut *Reisender* den Wert *Pika* hat. Dies ist also eine *Selektion* und wird in relationaler Algebra so dargestellt:
$Ergebnis = \sigma_{Reisender='Pika'}(faehrtNach)$

6.8 Der Join als zusammengesetzte Operation

Wie wir einen Join bilden, haben wir bereits in Kapitel 5 anhand der SQL-Syntax ausgiebig geübt. Diese Operation setzen wir jetzt aus den relationalen Basisoperationen zusammen, die wir gerade kennen gelernt haben.

Ein Join kombiniert Informationen aus zwei (meistens) verschiedenen Relationen. Basis dafür ist das Kreuzprodukt dieser Relationen – es kombiniert alle Tupel („Zeilen“) der einen Relation mit allen Tupeln der anderen. Aus all diesen möglichen Kombinationen wählen wir mit einer Selektion („*WHERE*“) diejenigen aus, die auch tatsächlich logisch zusammengehören. Im einfachsten Fall sind das diejenigen, die in allen gemeinsamen Attributen gleich sind. Das nennt man den einfachen oder **natürlichen Join**.

Anschließend entfernt man noch die doppelten, da gemeinsamen Attribute – also genau diejenigen, über die man den Join erstellt hat: Von je zwei identischen Attributen wird nur eines

übrig gelassen. Wie erwähnt betrifft das im einfachsten Fall (*natürlicher* Join) alle gleichnamigen Attribute.

Ein Join ist also eine kombinierte Operation aus Kreuzprodukt, Selektion und Projektion.

Man stellt den Join mit dem Symbol $\bowtie$ dar und notiert ihn wie folgt:
$R \bowtie S = \pi_{a_1,a_2,\ldots,a_n}(\sigma_\varphi(R \times S))$

Ganz innen, also zuerst auszuführen, sehen wir das Kreuzprodukt ($R \times S$). Eine Klammerschicht weiter außen sehen wir die Selektion (σ), wobei φ für den natürlichen Join genau die Selektionsbedingung darstellt, dass gemeinsame (gleichnamige) Attribute gleich sind. Außen sehen wir die Projektion (π), wobei a_1 bis a_n für die Liste aller nicht-gemeinsamen Attribute der beiden Ausgangsrelationen R und S sowie je *eine Version* der gemeinsamen Attribute steht.

Nicht-natürliche Joins

In unserer praktischen Arbeit mit Datenbanken haben wir uns nicht auf die Bedingung „gleichnamige Attribute sind gleich“ festgelegt, um festzustellen, welche Tupel verschiedener Relationen tatsächlich zusammengehören (Selektionsbedingung). Vielmehr haben wir uns auf die von uns zuvor definierten *Schlüsselbeziehungen* berufen. Oft kann es sinnvoll sein, dem Fremdschlüssel in einer Tabelle einen anderen Namen zu geben, als er in seiner eigenen Tabelle als Primärschlüssel heißt.

Auch solche Operationen sind Joins, jedoch keine *natürlichen* Joins. In diesem Fall ist die Selektionsbedingung φ nicht die, dass gleiche Attribute gleich sind, sondern eine beliebige, die wir korrekt wählen müssen. Diese spezielle Selektionsbedingung notiert man in diesem Fall unter dem $\bowtie$-Symbol.

Mit nicht-natürlichen Joins werden wir uns jedoch im Rahmen dieses Kurses nicht beschäftigen (d.h. nicht in relationaler Algebra – in SQL haben wir sie ja schon gemacht).

Der Join am Beispiel

Kommen wir noch einmal zurück zu unseren beiden Relationen, mit den wir das Thema Relationenalgebra angeschnitten haben: *mag* („Person Y mag Person Z“, siehe Abschnitt 6.1) und *denkt* („Person X denkt, dass Person Y Person Z mag“, siehe Abschnitt 6.2).

Der natürliche Join dieser beiden Relationen ist eine Relation mit der Aussage „Person X denkt, dass Person Y Person Z mag, und das stimmt auch“.

Diesen Join bilden wir jetzt sowohl in der Notation der relationalen Algebra als auch in SQL. Wie erwähnt bildet sich der natürliche Join über die (implizite) Selektionsbedingung, dass gleiche Attribute den gleichen Inhalt haben müssen.

Zunächst — und in einem mathematischen Ausdruck bedeutet das: in der innersten Klammer – bilden wir das Kreuzprodukt der beiden Relationen: $mag \times denkt$.

Anschließend (eine Klammer weiter außen) selektieren wir aus diesem Kreuzprodukt die tatsächlich zusammengehörenden Zeilen. Unsere Selektionsbedingung ist dabei wieder, dass gleiche Attribute gleich sind. Das Attribut Y der Relation *mag* muss gleich dem Attribut Y der

Relation *denkt* sein und das Attribut Z der Relation *mag* muss gleich dem Attribut Z der Relation *denkt* sein[8]:

$\sigma_{mag.Y=denkt.Y \wedge mag.Z=denkt.Z}$
($\wedge$ ist das Zeichen für das logische UND)

Zuletzt (also ganz außen in der Klammer-Hierarchie) zaubern wir die doppelten (weil gemeinsamen) Attribute weg, so dass nur noch je eines davon übrig bleibt: Wie *projizieren* also auf jeweils eine Version des doppelten Attributs. Welche wir nehmen, ist egal – wie wir mit der Selektion bereits sichergestellt haben, sind sie ja schließlich gleich. Nehmen wir zum Beispiel beide aus der Relation *mag*:

$\pi_{X,mag.Y,mag.Z}$

So sieht der natürliche Join aus *mag* und *denkt* in relationaler Algebra aus:
$mag \bowtie denkt = \pi_{X,mag.Y,mag.Z}(\sigma_{mag.Y=denkt.Y \wedge mag.Z=denkt.Z}(mag \times denkt))$

Der gleiche Join in SQL:

```
SELECT X, mag.Y, mag.Z FROM mag, denkt
WHERE mag.Y = denkt.Y AND mag.z = denkt.Z;
```

Das Ergebnis ist folgende Relation:

mag ⋈ denkt:

X	Y	Z
Claire	Alice	Bob
Claire	Claire	Alice
Pika	Pika	Pika

6.9 Grenzen des Modells

Die relationale Algebra ist das theoretische Modell, der „mathematische Unterbau“ relationaler Datenbanken. Modelle sind gut, um einen Überblick über das zu haben, was man tut. Formale, mathematische Modelle haben zudem den Vorteil, dass man sie – mathematisch korrekt – auf ihre Eigenschaften untersuchen und mathematisch sichere Beweise führen kann.

Wie jedes Modell hat jedoch auch die relationale Algebra ihre Grenzen, die sich in real existierender Datenbanksoftware wiederspiegeln. Manche Features und sonstigen Dinge sind in Datenbanksoftware einfach „gewachsen“ oder die Anforderungen der Benutzer gingen einfach nicht 100% mit dem mathematischen Modell konform.

Wenn man so will und es ganz streng sieht, ist real existierende Datenbanksoftware gar nicht wirklich relational. Einige Beispiele:

- Aus der Definition der Relation als spezielle Menge ergibt sich, dass Elemente einer Relation keine bestimmte Ordnung haben. Zeilen in Datenbanktabellen werden jedoch durchaus in einer bestimmten Reihenfolge gespeichert – und, sofern sie nicht umsortiert werden, auch so ausgegeben. In den meisten Datenbanksystemen ist das die Reihenfolge des Einfügens in die Tabelle.

[8] Die folgende eindeutig-machende Bezeichnung gleichnamiger Attribute in der Form *Relationsname.Attributname* ist in der Darstellung der relationalen Algebra nicht üblich. Stattdessen führt man die *Umbenennung* als weitere relationale Basisoperation ein. Wir verwenden diese Schreibweise als Vereinfachung und analog zur SQL-Schreibweise trotzdem.

- Ebenfalls direkt aus der Tatsache, dass Relationen Mengen sind, ergibt sich, dass sie keine gleichen Elemente (Doppelungen) enthalten können. In den meistens Datenbanksystemen ist das dagegen möglich.
- Weil Doppelungen in realen Datenbanksystemen möglich sind und insbesondere auch in *SELECT*-Ergebnissen möglich sind, gibt es in SQL die Klausel *GROUP BY*, mit dem sich Doppelungen wieder entfernen lassen. Außerdem bietet GROUP BY auch die Möglichkeit, „Doppelungen“ zu eliminieren, selbst wenn diese nur in bestimmten Attributen übereinstimmen. Die Funktionalität von *GROUP BY* ist in der traditionellen relationalen Algebra nicht darstellbar.
- Die einzelnen Elemente einer Relation müssen „spaltenweise“ den gleichen Mengen entstammen. Manche Datenbanksysteme – insbesondere das auch von uns verwendete *SQLite*, weichen von dieser Regel so stark ab, dass beliebige Daten in jedem Tabellenfeld stehen können. *SQLite* betrachtet selbst die mit *CREATE TABLE* Befehlen deklarierten Datentypen nur als eine Art Richtwert.

Es ist sicher so, dass relationale Algebra heute für viele Nutzer von Datenbanksystemen irrelevant ist. Dieses mathematische Konstrukt mag wie eine unglaublich aufgeblasene Theorie um schnöde Tabellen wirken.

Sie ist jedoch auch im Grundkurs Pflichtinhalt des Lehrplans.

6.10 Übungen

Die folgenden Übungen beziehen sich auf die Mengen P, O und G:

$P = \{Alice,\ Bob,\ Claire,\ Pika\}$
$O = \{Apfel,\ Banane,\ Orange,\ Birne\}$
$G = \{knackig,\ weich,\ kräftiges\ Gelb,\ besonders\ weich\}$

Über diesen Mengen seien die folgenden Relationen *isstGerne* und *magNicht* definiert:

isstGerne:

Person	Obst	Sonderwunsch
Pika	Banane	kräftiges Gelb
Pika	Orange	-
Bob	Orange	besonders reif
Bob	Birne	knackig
Alice	Orange	besonders reif
Alice	Banane	-
Alice	Birne	weich

magNicht:

Person	Obst
Alice	Apfel
Pika	Birne
Claire	Banane
Claire	Apfel
Claire	Orange
Claire	Birne

Außerdem betrachten wir die Menge W und die über P und W definierte Relation *wohnt* sowie die Menge J und die über P und J definierte Relation *arbeitet*.

W={*Ammergau, Bonn, Clauen, Berlin, Frankfurt am Main*}

wohnt:

Person	**Wohnort**
Alice	Ammergau
Bob	Bonn
Claire	Clauen

J={*Programmierer, Programmiererin, Banker, Bankerin, Wirtschaftskrimineller, Wirtschaftskriminelle*}

arbeitet:

Person	**Job**
Alice	Programmiererin
Bob	Banker
Claire	Wirtschaftskriminelle

Bitte bestimmen Sie die Ergebnisse folgender Operationen. Schreiben Sie die Ergebnisse sowohl in Tabellenform als auch in algebraischer Notation auf (lassen Sie in der algebraischen Notation das Relationenschema jeweils weg).

Bilden Sie außerdem die diesen Operationen entsprechenden SQL-Befehle.

1. $\pi_{\text{Person, Obst}}(isstGerne)$
2. $\pi_{\text{Person}}(magNicht)$
3. $\sigma_{\text{Person=”Bob”}}(isstGerne)$
4. $\sigma_{\text{Obst=”Orange”}}(isstGerne)$
5. $\sigma_{\text{Obst=”Orange”} \wedge \text{ Sonderwunsch=”besonders reif”}}(isstGerne)$
6. $\sigma_{\text{Obst=”Birne”} \wedge \text{ Person = ”Pika”}}(isstGerne)$
7. $\sigma_{\text{Obst=Person}}(isstGerne)$
8. $\pi_{\text{Person}}(\sigma_{\text{Obst=”Apfel”}}(magNicht))$
9. $\sigma_{\text{Obst=”Apfel”}}(\pi_{\text{Person}}(magNicht))$
10. $\pi_{\text{Obst}}(\sigma_{\text{Person=”Alice”}}(isstGerne))$
11. $isstGerne \bowtie magNicht$
12. $isstGerne \bowtie wohnt$
13. $\pi_{\text{Obst, Wohnort}}(isstGerne \bowtie wohnt)$
14. $\pi_{\text{Obst, Wohnort}}(wohnt \bowtie isstGerne)$
15. $\sigma_{\text{Person=”Pika”}}(wohnt \bowtie magNicht)$
16. $\pi_{\text{Obst}}(\sigma_{\text{Wohnort=”Bonn”}}(wohnt \bowtie isstGerne))$

17. $wohnt \times magNicht$
18. $\pi_{\text{wohnt.Person, Wohnort, Obst}}(\sigma_{\text{wohnt.Person = magNicht.Person}}(wohnt \times magNicht))$
19. $isstGerne \bowtie wohnt \bowtie arbeitet$
20. $\sigma_{\text{Person="Claire"}}(isstGerne \bowtie wohnt \bowtie arbeitet)$
21. $\pi_{\text{Obst, Wohnort, Job}}(\sigma_{\text{Obst="Orange"}}(isstGerne \bowtie wohnt \bowtie arbeitet))$

Stellen Sie außerdem die natürlichen Joins $wohnt \bowtie isstGerne$ sowie $magNicht \bowtie wohnt \bowtie arbeitet$ als Zusammensetzung von relationalen Basisoperationen dar.

Lösungen

Im Folgenden sind nur die Lösungen in algebraischer Notation sowie die SQL-Befehle wiedergegeben.

1. $\pi_{\text{Person, Obst}}(isstGerne) =$
   {(Pika, Banane), (Pika, Orange), (Bob, Orange), (Bob, Birne), (Alice, Orange), (Alice, Banane), (Alice, Birne)}
   `SELECT Person, Obst FROM isstGerne;`
2. $\pi_{\text{Person}}(magNicht) =$
   {Alice, Pika, Claire}
   `SELECT Person FROM magNicht;`
3. $\sigma_{\text{Person="Bob"}}(isstGerne) =$
   {(Bob, Orange, besonders reif), (Bob, Birne, knackig)}
   `SELECT * FROM isstGerne WHERE Person="Bob";`
4. $\sigma_{\text{Obst="Orange"}}(isstGerne) =$
   {(Pika, Orange, $\emptyset$), (Bob, Orange, besonders reif), (Alice, Orange, besonders reif)}
   Anmerkung: $\emptyset$ ist die leere Menge und entspricht einer leeren Tabellenspalte. Die leere Menge ist formal Teilmenge jeder beliebigen Menge und damit auch von G.
   `SELECT * FROM isstGerne WHERE Obst="Orange";`
5. $\sigma_{\text{Obst="Orange"} \wedge \text{ Sonderwunsch="besonders reif"}}(isstGerne) =$
   {(Bob, Orange, besonders reif), (Alice, Orange, besonders reif)}
   ```
   SELECT * FROM isstGerne WHERE Obst="Orange" AND
   Sonderwunsch="besonders reif";
   ```
6. $\sigma_{\text{Obst="Birne"} \wedge \text{ Person = "Pika"}}(isstGerne) = \emptyset$
   Anmerkung: Das Ergebnis ist leer, da kein Tupel existiert, das diese Bedingung erfüllt. $\emptyset$ steht wieder für die leere Menge, also ein leeres Ergebnis.
   `SELECT * FROM isstGerne WHERE Obst="Birne" AND Person="Pika";`
7. $\sigma_{\text{Obst=Person}}(isstGerne) = \emptyset$
   Anmerkung: Die Bedingung, dass die Attribute *Obst* und *Person* gleich sein sollen, ist offensichtlich Unsinn.
   `SELECT * FROM isstGerne WHERE Obst=Person;`

8. $\pi_{\text{Person}}(\sigma_{\text{Obst=”Apfel”}}(magNicht)) =$
{Alice, Claire}
```
SELECT Person FROM magNicht WHERE Obst="Apfel";
```

9. $\sigma_{\text{Obst=”Apfel”}}(\pi_{\text{Person}}(magNicht)) = \emptyset$
Anmerkung: So herum funktioniert es nicht! Wenn wir das Attribut *Person* erst (in der Klammer) durch eine Projektion entfernen, können wir es später (außerhalb der Klammer) nicht mehr überprüfen. Ein korrekter SQL-Befehl zu dieser Operation existiert nicht.

10. $\pi_{\text{Obst}}(\sigma_{\text{Person=”Alice”}}(isstGerne)) =$
{Orange, Banane, Birne}
```
SELECT Obst FROM isstGerne WHERE Person="Alice";
```

11. $isstGerne \bowtie magNicht = \emptyset$
Anmerkung: Obwohl formal korrekt, macht dieser Join keinen Sinn. Das Ergebnis hätte die Aussage „Personen, die ein Obst gerne essen, obwohl sie es nicht mögen”.
```
SELECT isstGerne.Person, isstGerne.Obst, Sonderwunsch FROM
isstGerne, magNicht WHERE isstGerne.Person = magNicht.Person
AND isstGerne.Obst = magNicht.Obst;
```

12. $isstGerne \bowtie wohnt =$
{(Alice, Orange, besonders reif, Ammergau), (Alice, Banane, ∅, Ammergau), (Alice, Birne, weich, Ammergau), (Bob, Orange, besonders reif, Bonn), (Bob, Birne, knackig, Bonn)}
```
SELECT isstGerne.Person, Obst, Sonderwunsch, Wohnort FROM
isstGerne, wohnt WHERE isstGerne.Person = wohnt.Person;
```

13. $\pi_{Obst,Wohnort}(isstGerne \bowtie wohnt) =$
{(Orange, Ammergau), (Banane, Ammergau), (Birne, Ammergau), (Orange, Bonn), (Birne, Bonn)}
```
SELECT Obst, Wohnort FROM isstGerne, wohnt WHERE
isstGerne.Person = wohnt.Person;
```

14. $\pi_{\text{Obst, Wohnort}}(wohnt \bowtie isstGerne) =$
{(Orange, Ammergau), (Banane, Ammergau), (Birne, Ammergau), (Orange, Bonn), (Birne, Bonn)}
Anmerkung: Die Reihenfolge der Operanden ist beim Join mit anschließender Projektion egal - so, wie z.B. die Reihenfolge der Operanden auf ganzen Zahlen bei einer Addition egal ist. Vorsicht: Ohne anschließende Projektion hängt die Reihenfolge der Attribute („Spalten“) im Ergebnis von der Operandenreihenfolge des Joins ab.
```
SELECT Obst, Wohnort FROM wohnt, isstGerne WHERE
isstGerne.Person = wohnt.Person;
```

15. $\sigma_{\text{Person=”Pika”}}(wohnt \bowtie magNicht) = \emptyset$
Anmerkung: Pika kommt in der Relation *wohnt* nicht vor.
```
SELECT wohnt.Person, Wohnort, Obst FROM wohnt, magNicht WHERE
wohnt.Person = magNicht.Person AND Person="Pika";
```

16. $\pi_{Obst}(\sigma_{\text{Wohnort=”Bonn”}}(wohnt \bowtie isstGerne)) =$
{Orange, Birne}
```
SELECT Obst FROM wohnt, isstGerne WHERE wohnt.Person =
isstGerne.Person AND Wohnort="Bonn";
```

17. $wohnt \times arbeitet =$
{(Alice, Ammergau, Alice, Programmiererin), (Alice, Ammergau, Bob, Banker), (Alice, Ammergau, Claire, Wirtschaftskriminelle), (Bob, Bonn, Alice, Programmiererin), (Bob, Bonn, Bob, Banker), (Bob, Bonn, Claire, Wirtschaftskriminelle), (Claire, Clauen, Alice, Programmiererin), (Claire, Clauen, Bob, Banker), (Claire, Clauen, Claire, Wirtschaftskriminelle)}
```
SELECT * FROM wohnt, arbeitet;
```
Anmerkung: Wie wir sehen, lässt sich das unglaublich umfangreiche und selten wirklich gewollte Kreuzprodukt sehr einfach (und auch sehr leicht versehentlich) mit SQL erstellen – ein bekannter Kritikpunkt an dieser Sprache.

18. $\pi_{\text{wohnt.Person, Wohnort, Obst}}(\sigma_{\text{wohnt.Person = magNicht.Person}}(wohnt \times magNicht)) =$
{(Alice, Ammergau, Apfel), (Claire, Clauen, Banane), (Claire, Clauen, Apfel), (Claire, Clauen, Orange), (Claire, Clauen, Birne)}
Anmerkung: Das ist genau der natürliche Join $wohnt \bowtie magNicht$.
```
SELECT wohnt.Person, Wohnort, Obst FROM wohnt, magNicht WHERE
wohnt.Person = magNicht.Person;
```

19. $isstGerne \bowtie wohnt \bowtie arbeitet =$
{(Bob, Orange, besonders reif, Bonn, Banker), (Bob, Birne, knackig, Bonn, Banker), (Alice, Orange, besonders reif, Ammergau, Programmiererin), (Alice, Banane, $\emptyset$, Ammergau, Programmiererin), (Alice, Birne, weich, Ammergau, Programmiererin)}
```
SELECT isstGerne.Person, Obst, Sonderwunsch, Wohnort, Job FROM
isstGerne, wohnt, arbeitet WHERE isstGerne.Person =
wohnt.Person AND wohnt.Person = arbeitet.Person;
```

20. $\sigma_{\text{Person=”Claire”}}(isstGerne \bowtie wohnt \bowtie arbeitet) = \emptyset$
Anmerkung: Claire kommt in der Relation *isstGerne* nicht vor.
```
SELECT isstGerne.Person, Obst, Sonderwunsch, Wohnort, Job FROM
isstGerne, wohnt, arbeitet WHERE isstGerne.Person =
wohnt.Person AND wohnt.Person = arbeitet.Person AND
isstGerne.Person = "Claire";
```

21. $\pi_{\text{Obst, Wohnort, Job}}(\sigma_{\text{Obst=”Orange”}}(isstGerne \bowtie wohnt \bowtie arbeitet)) =$
{(Orange, Bonn, Banker), (Orange, Ammergau, Programmiererin)}
```
SELECT Obst, Wohnort, Job FROM isstGerne, wohnt, arbeitet WHERE
isstGerne.Person = wohnt.Person AND wohnt.Person =
arbeitet.Person AND Obst = "Orange";
```

Die beiden gefragten Joins lassen sie wie folgt in relationalen Basisoperationen ausdrücken:

- $wohnt \bowtie isstGerne =$
$\pi_{wohnt.Person,Wohnort,Obst,Sonderwunsch}(\sigma_{wohnt.Person=isstGerne.Person}(wohnt \times isstGerne))$

- $magNicht \bowtie wohnt \bowtie arbeitet =$
  $\pi_{magNicht.Person,Obst,Wohnort,Job}(\sigma_{magNicht.Person=wohnt.Person \wedge wohnt.Person=arbeitet.Person}(magNicht \times wohnt \times arbeitet))$

7 Datenschutz

Kommt noch! :-)