# Relationale Datenbanken

Patrick Stahlberg

14. Juli 2004

```
Message-ID: <slrn8ccsdo.qh0.news@teebee.mediaconsult.com>
From: news@browser.org (Rob Partington)
Subject: Re: How would Homer write an SQL query?
Date: 8 Mar 2000 15:29:17 GMT

Patrick Ryan <pat@halcyon.com> wrote:
> 'Tell me, O Muse, of that ingenious hero who travelled far
> and wide after he had sacked the famous town of Troy.'

connect muse
go
select person.*, personality.*, deed.*
from   person, deed, personality
where  person.id in
       (
         select person.id
           from person,
             deed as before,
             deed as after,
             personality
           where personality.trait='ingenious'
             and personality.trait='hero'
             and person.id=personality.id
             and before.action='sack'
             and before.location='Troy'
             and before.who=person.id
             and after.action='travel'
             and after.date > before.date
             and after.who=person.id
       )
go
```

gefunden in *comp.databases.theory*

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Der Entwurf einer neuen Datenbank gliedert sich in fünf Phasen, die in Abbildung 1.1 dargestellt sind (vgl. [5, S. 3]).

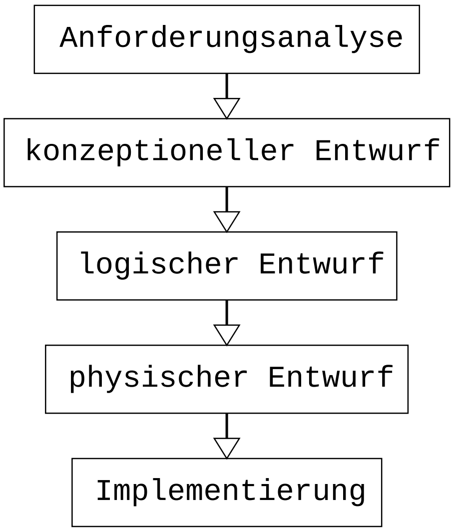


Abbildung 1.1: Die fünf Phasen des Datenbank – Entwurfs

Mit den ersten drei Phasen werde ich mich in Kapitel 2 beschäftigen. Im Zuge der Anforderungsanalyse wird eine *Miniwelt* erstellt (Kapitel 2.1). Der konzeptionelle Entwurf führt zu einem konzeptuellen DB – Schema. Dieses Schema dient dazu, die Struktur der Datenbank vollständig zu beschreiben und dabei unabhängig vom später verwendeten Datenmodell zu sein. Es gibt verschiedene konzeptionelle DB – Schemata. Ich werde aber nur auf das sog. *ER – Schema* eingehen (ab Kapitel 2.2). Dem konzeptionellen Entwurf folgt der logische Entwurf. Dieser umfasst die Transformation des konzeptionellen DB – Schemas in ein *logisches DB – Schema*. Das logische DB – Schema ist bereits von dem verwendeten Datenmodell abhängig. Der logische Entwurf umfasst deshalb auch für das Datenmodell spezifische Optimierungen des DB – Schemas, wie zum Beispiel die *Normalisierung* (Kapitel 2.7). Getreu dem Thema meiner Arbeit werde ich als Datenmodell das *relationale Datenmodell* wählen (ab Kapitel 2.5).

Der physische Datenbankentwurf wird nicht Thema dieser Arbeit sein.

Das logische DB – Schema ist nur bedingt (durch das verwendete Datenmodell) von dem letztendlich verwendenen DBMS (Datenbankmanagementsystem) abhängig. Erst die *Implementierung* beinhaltet die genaue Schemadefinition in der Sprache des DBMS. Unter relationalen DBMS ist die Sprache *SQL* zur Schemadefinition quasi – Standard. Darauf wird Kapitel 3 eingehen.

Ziel jeder Datenbankentwicklung ist die Benutzung der Datenbank. Als Schnittstelle zwischen dem laufenden DBMS und dem Anwender oder Anwendungsprogrammen gibt es sog. *Abfragesprachen* (Kapitel 4).

Das relationale Datenmodell ist nicht das einzige existierende Datenmodell. Es gibt durchaus Alternativen. Eine davon werde ich kurz in Kapitel 5 vorstellen.

Nicht jeder soll Zugriff auf alle Daten haben. Um diese Forderung zu erfüllen, gibt es einige *Datenschutzmechanismen*. Diese werden in Kapitel 6 erklärt.

Datenbanken sammeln naturgemäß meist riesige Mengen an Daten an. Mit zunehmender Komplexität müssen spezielle Vorkehrungen getroffen werden, dass die Daten immer in einem widerspruchsfreien und gültigen Zustand sind. Diesem Thema widmet sich das Kapitel 7.

# Kapitel 2

# Das relationale Datenmodell

Ich werde im Folgenden das relationale Datenmodell anhand der Vorgehensweise erklären, nach der man vorgeht, wenn man eine neue Datenbank erstellen möchte. Der Weg führt von den ersten Überlegungen darüber, welche Daten man überhaupt mit der Datenbank verwalten möchte bis hin zu einem fertigen, stabilen Datenmodell, das bereit für die Implementierung in einem relationalen Datenbankmanagement – System (RDBMS) ist.

## 2.1 Die Miniwelt

Wenn man eine Datenbank plant, muss man sich als erstes überlegen, welche Daten denn überhaupt gespeichert werden sollen. Es ist klar, dass man nicht sämtliche zur Verfügung stehenden Daten mit in das Datenbanksystem übernehmen kann und will. Das liegt einerseits daran, dass auch heutige Speichermedien noch eine begrenzte Speicherkapazität haben (Aller Wahrscheinlichkeit nach wird sich das auch nicht so schnell ändern). Ein anderer Grund liegt darin, dass Datenbanken mit zunehmender Menge an Informationen an Komplexität und Unübersichtlichkeit und damit auch an Instabilität zunehmen.

Ziel ist es also, ein Abbild der realen Welt zu schaffen, in dem aber nur die wirklich notwendigen Informationen vorkommen [4, S. 13ff]. Informationen sind dann notwendig, wenn sie für die Erledigung der Aufgaben des Datenbanksystems (Bereitstellen und Verarbeiten von Informationen) unabdingbar sind. Es ist nicht nötig, Informationen aufzunehmen, die vielleicht später einmal gebraucht werden könnten, da eine bestehende Datenbank, die dem relationalen Modell entspricht, leicht durch zusätzliche Daten erweitert werden kann [1, Kap. 1].

Das Ergebnis dieses ersten Schrittes der Konzeption einer Datenbank ist die sogenannte *Miniwelt* [10, S. 19]. Die Miniwelt ist also ein Abbild (oder eine Teilmenge) der realen Welt. Sie repräsentiert ein Konzept der realen Welt, wie zum Beispiel eine Firma, eine Schule oder eine CD – Sammlung.

Mir ist keine Notationsform für Miniwelten bekannt. Ein akribisches Festhalten einer Miniwelt ist meiner Ansicht nach auch weder nötig noch möglich, da die Informationen noch ziemlich unstrukturiert sind. Erst der nächste Schritt im Datenbankdesign strukturiert die Informationen und bietet eine Möglichkeit, die strukturierten Informationen in übersichtlicher Form festzuhalten.

## 2.2 Das Entity – Relationship – Modell (ERM)

### 2.2.1 Entities, Relationships

Wenn man eine Miniwelt einmal genauer betrachtet, stellt man fest, dass man dort *Objekte* finden kann. Diese Objekte stehen zueinander in *Beziehungen*.

Zum Beispiel können in der Miniwelt einer Schule "Objekte" wie Lehrer und Schüler vorkommen. Diese Objekte stehen in Beziehung zueinander. Man kann beispielsweise sagen: "Lehrer $A$ ist Lehrer von Schüler $B$."

Objekte werden in der Fachsprache als *Entities* bezeichnet (engl. Entity: Dasein, Wesen, Ding). Beziehungen heißen *Realtionships* (engl. relationship: Beziehung) [4, S. 44 u.a.]. In der Praxis wird der Begriff "Objekt" meist vermieden, da er zweideutig und irreführend sein kann. Deshalb werde ich das im Folgenden auch tun.

Es ist für das Verständnis der folgenden Unterkapitel wichtig, zwischen Entities und den eigentlichen Datensätzen zu unterscheiden. Entities sind nur Abbilder von Konzepten in der realen Welt. Sie werden nie direkt in der Datenbank abgespeichert, sondern bilden vielmehr eine "Schablone" für die eigentlichen Daten. Man könnte Informationen über Entities auch als *Meta – Daten* bezeichnen. Ein Beispiel: Das Entity "Schüler" enthält noch keinen speziellen Schüler. Es beschreibt lediglich, was für Eigenschaften ein Schüler haben kann etc. (Dazu später mehr). Die konkreten Schülerdaten werden dann in Datensätzen gespeichert. Allen Schülerdatensätzen ist gemein, dass sie zu dem Entity "Schüler" gehören.

### 2.2.2 Grafische Darstellung des ERM

Wie ich bereits erwähnt habe, stellt das ERM Methoden bereit, um die Entities und ihre Beziehungen grafisch zu veranschaulichen[1]. Leider gibt es keine einheitliche Notation für diese grafische Darstellung, es hat sich vielmehr eine Vielzahl von Notationsformen für ERM's herausgebildet, die sich aber größtenteils nur in Details unterscheiden [6]. Ich werde im Folgenden das von SAP entwickelte *Strukturierte Entity – Relationship – Modell* (abgekürzt: SAP – SERM) verwenden, da es offensichtlich das am weitesten verbreitete ist[2].

Abbildung 2.1 zeigt, wie Entities im ERM dargestellt werden. Wie man sieht, werden sie dargestellt durch ihren Namen, der von einem rechteckigen Rahmen umrandet wird.


Abbildung 2.1: ERM für ein Entity

Beziehungen werden durch Linien gekennzeichnet, die die betreffenden Entities verbinden. Abbildung 2.2 zeigt eine Beziehung zwischen Lehrer und Schüler.

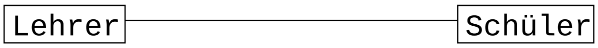


Abbildung 2.2: Beziehung zwischen Lehrer und Schüler im ERM

Zur Qualifizierung der Beziehung kann man noch eine Beschreibung an die Verbindungslinie schreiben [10, S. 232]. Die in Abbildung 2.2 dargestellte Beziehung kann zum Beispiel beschrieben werden mit dem Ausduck: "ist Lehrer von" Das entsprechende ERM zeigt Abbildung 2.3.

[1]Eine solche Grafik wird oft als *Entity – Relationship – Diagramm* oder ERD bezeichnet.

[2]Einen Überblick über einige weitere verbreitete Notationsformen gibt [6] im Kapitel 1.2.3.5.


Abbildung 2.3: Qualifizierte Beziehung zwischen Lehrer und Schüler im ERM

## 2.3 Der Grad von Beziehungen

Wie bereits gesagt, bestehen zwischen Entities Beziehungen. Diese Beziehungen können von unterschiedlicher Art sein. Man unterscheidet drei verschiedene *Grade von Beziehungen*[3] [1, Kap. 2.2]:

- Die 1:1 – Beziehung (eins – zu – eins)
- Die 1:n – Beziehung (eins – zu – viele)
- Die m:n – Beziehung (viele – zu – viele)

### 2.3.1 Die 1:1 – Beziehung

Eine 1:1 – Beziehung zwischen den Entities $A$ und $B$ tritt dann auf, wenn jedem Datensatz aus dem Entity A genau ein Datensatz aus dem Entity $B$ zugeordnet werden kann und wenn jeder Datensatz aus $B$ genau einem Datensatz aus $A$ zugeordnet ist.

Beispielsweise besteht zwischen einem Spieler und einer Spielsteinfarbe beim Mensch – Ärgere – Dich – Nicht – Spiel eine 1:1 – Beziehung, weil jeder Spieler genau eine Farbe hat und jede Farbe genau vom einem Spieler gespielt wird.

Im SAP – SERM wird eine 1:1 – Beziehung durch einen einfachen Pfeil an beiden Enden der Verbindungslinie zwischen den Entities gekennzeichnet. Die oben beschriebene Beziehung zwischen Spieler und Farbe wird also so gekennzeichnet wie in Abbildung 2.4 dargestellt.

### 2.3.2 Die 1:n – Beziehung

Eine 1:n – Beziehung zwischen den Entities $A$ und $B$ besteht dann, wenn jedem Datensatz aus $A$ mehrere Datensätze aus $B$ zugeordnet werden können

[3] Man spricht auch von *Degrees von Beziehungen*.

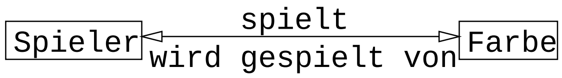


Abbildung 2.4: 1:1 – Beziehung zwischen Spieler und Farbe

und wenn jeder Datensatz aus $B$ genau einem Datensatz aus $A$ zugeordnet ist.

Solch eine 1:n – Beziehung findet man auch beim Mensch – Ärgere – Dich – Nicht – Spiel, wenn man die Beziehung zwischen Spieler und einzelnen Spielsteinen betrachtet. Jeder Spieler besitzt mehrere Spielfiguren, aber jede Spielfigur gehört nur genau einem Spieler. In diesem Fall wäre also der Spieler ein Datensatz aus dem Entity $A$ und der Spielstein wäre ein Datensatz aus dem Entity $B$.

1:n – Beziehungen stellt man im ERM durch einen Doppelpfeil an der "m – Seite" der Verbindungslinie sowie einen einfachen Pfeil an der anderen Seite dar (Aus technischen Gründen benutze ich hier ausgefüllte Pfeile anstelle von Doppelpfeilen). Abbildung 2.5 zeigt ein Beispiel dafür.


Abbildung 2.5: 1:n – Beziehung zwischen Spieler und Spielfiguren

### 2.3.3 Die m:n – Beziehung

Eine m:n – Beziehung zwischen den Entities $A$ und $B$ tritt dann auf, wenn jedem Datensatz aus $A$ mehrere Datensätze aus $B$ zugeordnet werden können und wenn jedem Datensatz aus $B$ mehrere Datensätze aus $A$ zugeordnet werden können.

Ein Beispiel für eine m:n – Beziehung ist die Beziehung zwischen Lehrer – und Schüler – Entities. Während jeder Lehrer mehrere Schüler unterrichtet, wird auch jeder Schüler von mehreren Lehrern unterrichtet[4].

Im ERM wird eine m:n – Beziehung durch Doppelpfeile[5] auf beiden Seiten

[4] Natürlich nicht gleichzeitig. Gemeint sind hier die verschiedenen Unterrichtsfächer, in denen ein Schüler von unterschiedlichen Lehrern unterrichtet wird.

[5] bzw. ausgefüllte Pfeile `;-)`

der Verbindungslinie gekennzeichnet. Abbildung 2.6 zeigt ein Beispiel.


Abbildung 2.6: m:n – Beziehung zwischen Lehrern und Schülern

### 2.3.4 Mitgliedsklassen

Jeder der angesprochenen Beziehungstypen lässt sich noch einmal unterteilen, indem man zwischen verschiedenen Mitgliedsklassen unterscheidet [6].

Mitgliedsklassen unterscheiden zwischen starken und schwachen Beziehungen, bzw. zwischen obligatorischen oder nicht obligatorischen Mitgliedschaften.

Allgemein kann man sagen, dass starke Beziehungen eine unbedingte Mitgliedschaft erfordern, während das bei schwachen Beziehungen nicht der Fall ist. Bei starken Beziehungen ist die Existenz der Beziehung also zwingend erforderlich, während bei schwachen Beziehungen die Möglichkeit besteht, dass für einige Datensätze diese Beziehung nicht existiert[6].

Wenn das Entity "Lehrer" beispielsweise mit dem Entity "Schüler" in einer schwachen Beziehung steht, dann bedeutet das, dass jeder Lehrer Schüler unterrichten *kann*. Dies ist aber nicht zwingend der Fall (Ein arbeitsloser Lehrer unterrichtet z.B. keine Schüler). Anders wäre es, wenn man diese Beziehung als "stark" definieren würde. Dann müsste jeder Lehrer zwangsläufig mindestens einen Schüler unterrichten. Aus dem genannten Grund kann es also sinnvoll sein, die Beziehung als "schwach" zu definieren.

Es gibt aber auch Beispiele, für die es durchaus einen Sinn ergibt, starke Beziehungen einzuführen. Die Beziehung zwischen einem Kfz – Kennzeichen und seinem Besitzer ist ein solches Beispiel. Es darf keine Kfz – Kennzeichen ohne Besitzer geben, da andernfalls die Polizei nicht wüsste, wohin sie ihre Strafzettel schicken sollte. Um solche Fälle zu vermeiden, führt man am besten gleich eine starke Beziehung ein.

Zu jeder Beziehung gehören zwei Entities. Man kann die Frage nach starker oder schwacher Beziehung für beide Entities getrennt diskutieren. Dass

[6] Der Unterschied zwischen starken und schwachen Beziehungen lässt sich vielleicht leichter verstehen, wenn man sich eine andere mögliche Bezeichnung vor Augen hält, die von *bedingten* und *unbedingten* Beziehungen spricht.

das sinnvoll ist, kann man an dem Beispiel aus dem letzten Absatz sehen. Ich habe gezeigt, dass eine starke Beziehung auf der Seite des Kfz – Entities zu empfehlen ist (d.h., jede Autonummer muss eine Beziehung zu einem Besitzer haben). Auf der anderen Seite ist jedoch eher eine schwache Beziehung sinnvoll, da ja nicht jede Person ein Auto besitzt (es muss also auch erlaubt sein, das eine Person keine Beziehung zu einer Autonummer hat).

Im ERM werden schwache Beziehungen durch einen kleinen Querstrich durch den Beziehungspfeil gekennzeichnet, der auf der Seite des Pfeils liegt, an der die Beziehung schwach ist. Bei starken Beziehungen fehlt dieser Querstrich (siehe Abbildungen 2.7, 2.8 und 2.9).

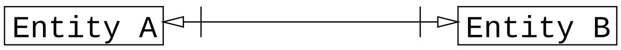


Abbildung 2.7: Schwache Beziehung auf beiden Seiten

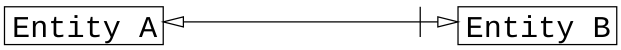


Abbildung 2.8: Diese Beziehung ist stark auf der Seite von Entity $A$ und schwach auf der Seite von Entity $B$.

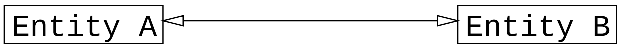


Abbildung 2.9: Starke Beziehung auf beiden Seiten

## 2.4 Eigenschaften von Entities

Jedes Entity hat Eigenschaften, die seine Datensätze identifizieren und näher beschreiben. Der Name ist z.B. eine Eigenschaft von Lehrern und Schülern. Weitere Eigenschaften lassen sich beliebig den Entities zuordnen. Die Eigenschaften werden auch *Attribute* genannt.

Um die Attribute von Entities in ein Entity – Relationship – Diagramm einzutragen, schreibt man die Namen dieser Attribute in das Rechteck der

Entity und umgibt sie mit jeweils einem Oval (vgl. [1, S. 23] [6]). Wie das an einem konkreten Beispiel aussieht, kann man in Abbildung 2.10 sehen.

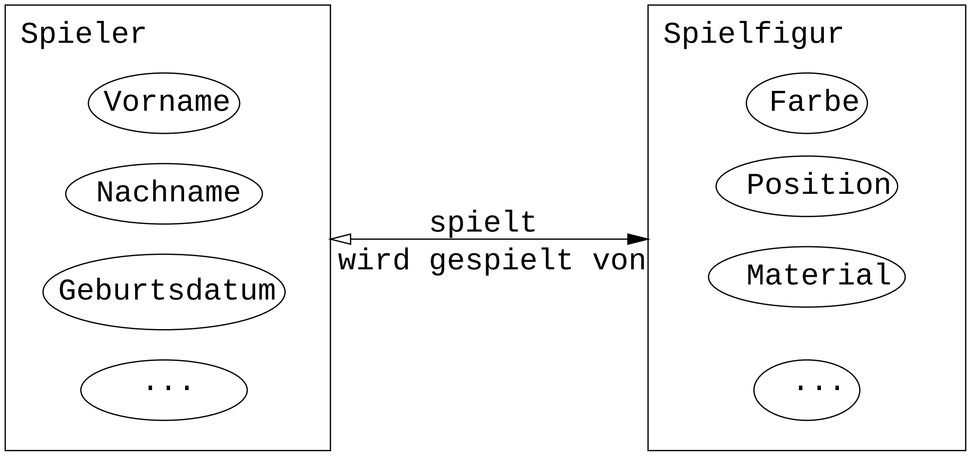


Abbildung 2.10: Entities mit Attributen im ERM

### 2.4.1 Domains

Jedem Attribut einer Entity ist eine sogenannte *Domain* (oder auf deutsch: *Domäne*) zugeordnet. Diese Domain ist der zugelassene Wertebereich eines Attributs. Sie beschreibt also die Menge der für dieses Attribut gültigen Werte [3, S.18f.].

Beispielsweise könnte die Domain für die Farbe eines Spielsteines so aussehen: `dom(Farbe) = {rot; blau; grün; gelb}`. Dann dürfen die Felder des Attributs "Farbe" also keine anderen Werte haben als `rot`, `blau`, `grün` oder `gelb`. Alle anderen Werte sind unzulässig.

Domains sind z.B. nützlich, um Fehleingaben zu verhindern. Gibt man beispielsweise bei einem Datum als Monatszahl 765 an, wird das DBMS diese Eingabe zurückweisen, weil 765 nicht Element der Domain für Monatszahlen ist.

Die Domain spielt auch beim Vergleich zweier Attributwerte eine Rolle. Datenbankmanagementsysteme lassen den Vergleich von Attributwerten bei bestimmten Kombinationen der Domains nicht zu. Das ist zum Beispiel der Fall, wenn man eine Zahl mit einem Buchstaben vergleichen will.

### Null – Werte

Ein Sonderfall von Attributwerten in einem relationalen Datenbanksystem sind *Null – Werte* [10, S. 32]. Null – Werte zeigen an, dass ein Wert nicht bekannt oder nicht gesetzt ist. Gutes Datenbank – Design schliesst Null – Werte fast aus, es gibt jedoch einige Situationen, in denen die Anwendung von Null – Werten nötig werden könnte. Solche Fälle treten zum Beispiel bei der Erweiterung eines bestehenden Datenbanksystems manchmal auf. Null – Werte müssen explizit in die Domain eines Attributs aufgenommen werden, damit sie zugelassen werden[7].

## 2.4.2 Schlüssel

In einer relationalen Datenbank muss jedes Entity bestimmte Attribute haben mit denen man seine Datensätze identifizieren kann [1, S. 33]. Solche identifizierenden Attribute nennt man *Schlüssel*[8]. In der relationalen Datenbanktheorie gibt es verschiedene Arten von Schlüsseln. Ich werde auf folgende Schlüsselarten näher eingehen (vgl. [1, Kap. 2.3.1]):

- Super Key
- Candidate Key
- Primary Key
- Foreign Key

### Super Key

Ein Super Key ist ein Attribut oder eine Menge von Attributen eines Entities, die die Datensätze des Entities eindeutig identifizieren. Es darf also in einem Entity keine zwei Datensätze mit dem gleichen Super Key geben. Die Gesamtheit aller Attribute eines Entities ist immer ein Super Key. Es lassen sich natürlich für die meisten Entities auch noch andere Super Keys finden[9].

[7] Bei der Datenbankabfragesprache SQL (Kapitel 3) müssen Null – Werte explizit *ausgeschlossen* werden, wenn man sie verhindern will [1, S.180].

[8] Die englische Bezeichnung für Schlüssel ist "Key". Dieser Begriff wird auch in der Fachsprache häufig verwendet.

[9] Obwohl Super Keys in keinem mir vorliegenden Buch erwähnt sind, scheinen sie doch eine bekannte Schlüsselart zu sein (siehe [8]).

### Candidate Key

Ein Candidate Key ist ein Super Key, der keine “überflüssigen” Schlüsselattribute enthält. Das bedeutet, dass man kein Schlüsselattribut von einem Candidate Key entfernen kann, ohne dass der Candidate Key seine Schlüsseleigenschaft verliert. Man kann also Candidate Keys als minimale Super Keys auffassen.

### Primary Key

Ein Primary Key ist ein Candidate Key, der idealerweise aus nur einem Attribut bestehen sollte[10]. Das kann man in jedem Fall erreichen, indem man ein neues Attribut einführt, das z.B. eine Seriennummer enthält[11]. Primary Keys werden hauptsächlich benutzt, um Beziehungen zu realisieren. Da Primary Keys in relationalen Datenbanken sehr wichtig sind, kennzeichnet man sie auch in ERMs, indem man das entsprechende Attribut unterstreicht[12][13] (siehe Abbildung 2.11[14]).

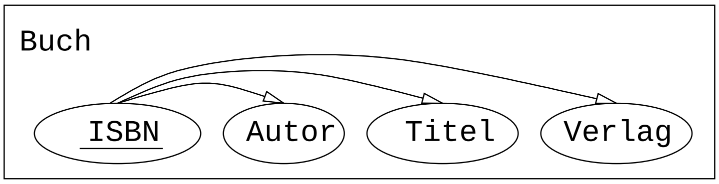


Abbildung 2.11: Primary Key im ERM

[10]Primary Keys die aus mehreren Attributen bestehen, heißen *Primary Concatinated Keys*.

[11]In einigen Fällen ist das aber nicht nötig, zum Beispiel bei Relationen, die Beziehungen repräsentieren (siehe z.B. Abbildung 2.21 auf Seite 27)

[12]Bei Concatinated Primary Keys schreibt man alle zum Primary Key gehörenden Attribute in ein Oval und trennt sie durch ein “+” – Zeichen. Dann unterstreicht man den gesamten Text.

[13]Das Unterstreichen von Primary Keys gehört soweit ich weiß nicht zur SAP – SERM – Spezifikation. Es wird aber z.B. in [1] angewandt und ich finde es sehr sinnvoll.

[14]In Abbildung 2.11 sind die Attribute mit Pfeilen verbunden. Der Grund dafür ist, dass das Entity sich in 2NF befindet (siehe Kapitel 2.7: Normalisierung).

**Foreign Key**

Ein Foreign Key ist in den meisten Fällen kein Schlüssel der Entity, die ihn enthält. Der Foreign Key ist vielmehr ein Primary Key einer anderen Entity. Er zeigt an, dass zwischen den beiden Entities eine Beziehung besteht. Ein Beispiel für den Einsatz von Foreign Keys zur Herstellung von Beziehungen zwischen Entities zeigt Abbildung 2.12 (Verlag_ID ist in der Entity *Buch* ein Foreign Key, da er auf den Primary Key der Entity *Verlag* zeigt).

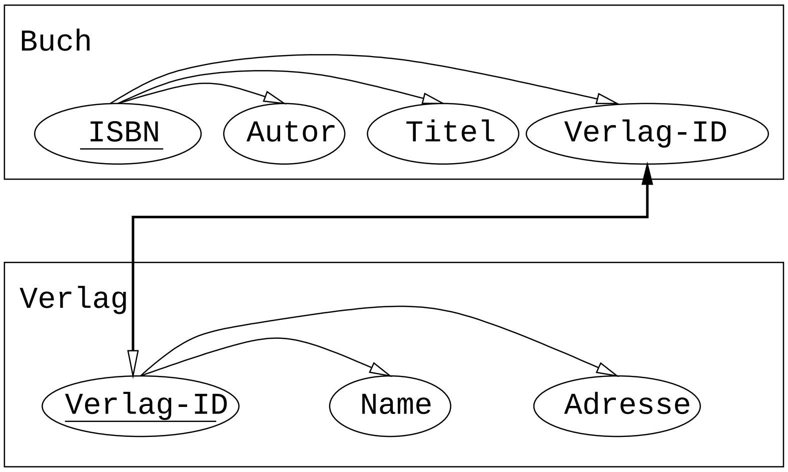


Abbildung 2.12: Herstellen einer Beziehung über einen Foreign Key

## 2.5 Relationen

In relationalen Datenbanken werden die Daten in *Relationen* gespeichert. Dieser Begriff kommt aus der Mathematik und bezeichnet eine eindeutige Zuordnung. Für das relationale Datenmodell reicht es aber aus, sich unter Relationen Tabellen vorzustellen [10, Kap. 2.2.2]. Abbildung 2.13 zeigt eine Beispielrelation, die Namen und Vornamen von Personen beinhaltet.

### 2.5.1 Elemente von Relationen

Die Abbildung 2.13 zeigt den prinzipiellen Aufbau von Relationen. Es gibt Spalten und Zeilen, an deren Kreuzungspunkten jeweils ein Wert steht. Die

| Vorname | Nachname |
|---|---|
| Franz | Müller |
| Christine | Schulze |
| Karl | Meier |
| Susanne | Müller |
| Herbert | Bäcker |

Abbildung 2.13: Eine Beispieltabelle

Gesamtheit aller Werte in einer Zeile wird als *Tupel* bezeichnet. Der Begriff "Tupel" kann also äquivalent zum Begriff "Zeile" verwendet werden. Jeder Spaltenwert eines Tupels wird *Attribut* genannt. Gleiche Attribute verschiedener Tupel stehen in der Relation in der gleichen Spalte. Die Spaltenüberschriften in der ersten Zeile[15] sind deshalb Bezeichner für das jeweilige Attribut.

Die Menge der Attribute eines Tupels wird als *Degree* bezeichnet. Aufgrund des Aufbaus einer Relation haben alle Tupel den gleichen Degree. Man spricht deshalb vom *Degree einer Relation* und meint den Degree jedes einzelnen Tupels der Relation.

Die Anzahl der in einer Relation enthaltenen Tupel nennt man *Kardinalität*.

Das relationale Datenmodell schreibt vor, dass die Reihenfolge der Tupel und der Attribute in einer Relation nicht festgelegt ist. Man kann also beliebig Spalten und Zeilen in der Relation vertauschen, ohne dass sich die Relation im Sinne des Datenmodells ändert. Eine weitere Folge daraus ist, dass Attribute nur anhand ihres Namens[16] und Tupel nur anhand ihres Primary Keys[17] identifiziert werden können.

Die bereits im Zusammenhang mit dem ERM angesprochenen Domänen von Attributen gelten auch in Relationen. Jeder Wert in einer Relation muss also Element der entsprechenden Domäne sein. Jeder Spalte wird genau eine

[15] Die erste Zeile ist natürlich kein Tupel.

[16] Der Name eines Attributs ist der bereits angesprochene Bezeichner in der Kopfzeile der Relation.

[17] Richtiger wäre hier: Super Keys, da diese ja auch das Tupel innerhalb der Relation (bzw. den Datensatz innerhalb der Entity) eindeutig identifizieren. Im allgemeinen spielen aber Super Keys und Candidate Keys kaum eine Rolle bei der Datenmodellierung. Deshalb spricht man im Zusammenhang mit Relationen eigentlich nur noch von Primary Keys.

Domäne zugeordnet.

Abbildung 2.14 zeigt die wichtigsten Elemente einer Relation.

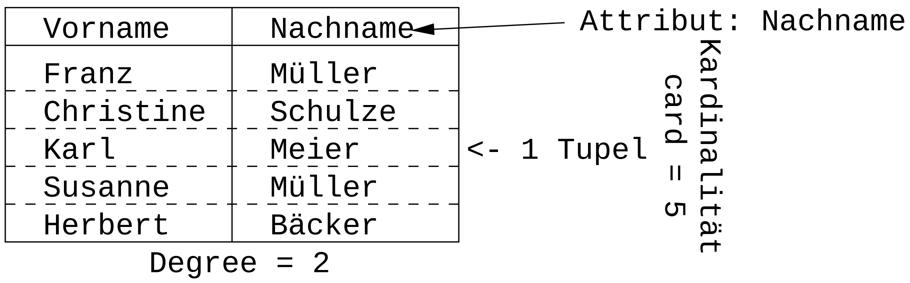


Abbildung 2.14: Die wichtigsten Elemente einer Relation (vgl. [10, S. 30])

### 2.5.2 Beziehungen zwischen Relationen

Wie bereits angedeutet, werden Beziehungen zwischen Relationen durch Primary Keys und Foreign Keys erzeugt [1, S. 22]. Die Abbildung 2.15 zeigt ein Beispiel dafür. Die Leute aus den beiden letzten Tabellen wollten ins Internet und sind kurzerhand Kunden eines Internet – Providers geworden. Der Provider hat zwei verschiedene Tarifmodelle die sich in dem Preis unterscheiden, den die Kunden pro Minute Internet – Nutzung bezahlen müssen und darin, ob die Kunden eine E – Mail – Adresse bekommen. Der Provider speichert seine Daten in zwei Relationen ab. Eine davon beinhaltet Kundendaten und die andere die Informationen über die Tarifmodelle. Jedes Tarifmodell hat eine eindeutige Identifikationsnummer, die gleichzeitig Primary Key der Tarifmodell – Relation ist. Man kann nun eine Beziehung zwischen Kunden und Tarifmodellen herstellen, indem man die Tarifmodell – Identifikationsnummer als Foreign Key in die Relation mit den Kundendaten aufnimmt.

### 2.5.3 Leere Relationen

Natürlich gibt es auch Relationen, die eine Kardinalität von 0 haben. Solche Relationen ohne Tupel werde ich künftig darstellen wie in Abbildung 2.16[18]

[18] Das ist eine von mir frei erfundene Darstellungsweise. Sie soll keinen Anspruch auf Allgemeingültigkeit erheben.

| Vorname | Nachname | Tarif_ID |
|---|---|---|
| Franz | Müller | 1 |
| Christine | Schulze | 2 |
| Karl | Meier | 1 |
| Susanne | Müller | 1 |
| Herbert | Bäcker | 2 |

| Tarif_ID | Preis/min | E-Mail |
|---|---|---|
| 1 | 3 Pf | nein |
| 2 | 4 Pf | ja |

Abbildung 2.15: Beziehung zwischen Relationen über einen Foreign Key

| Vorname | Nachname | Tarif-ID |
|---|---|---|

Abbildung 2.16: Eine leere Relation

## 2.6 Abbildung von ERMs auf Relationen

Die Erstellung des ERM ist nur ein Hilfsmittel, sich über die zu speichernden Datenstrukturen ein klares und strukturiertes Bild zu machen. Da eine relationale Datenbank aus Relationen besteht, muss man aus dem ERM Relationen machen. Das ist ein sehr komplizierter Teil der Datenbankerstellung. Er ist so zeitaufwendig, dass diese Aufgabe heutzutage praktisch niemand mehr von Hand erledigt. Stattdessen gibt es sogenannte *CASE – Tools*, die ausgehend von einem bestehenden ERM automatisch einen an das benutzte DBMS angepassten Satz von Relationen erstellen. Beispielhaft sei hier nur der VisioModeler genannt, der als Teil des Programmpakets "Visio Enterprise 2000" von er Firma Visio vertrieben wurde. Diese Firma wurde aber vor kurzem von Microsoft aufgekauft. Deshalb weiß ich nicht, ob diese Produktreihe weitergeführt wird[19].

Ich werde im Folgenden einige Regeln vorstellen, wie man aus einem ERM Relationen erstellt. Die vorgestellten Regeln decken alle Möglichkeiten des

[19]vgl. Message `<MptD4.208$RH5.10498@typhoon.mn.mediaone.net>` vom 26. März 2000 aus comp.databases.theory

ERM ab und zeigen somit, dass prinzipiell jedes ERM in Relationen umgewandelt werden kann. (Die Regeln sind hauptsächlich aus [10] übernommen und zum Teil verbessert, insbesondere um Null – Werte zu vermeiden.)

### 2.6.1 Abbildung von Entities auf Relationen

Die Umwandlung von Entities in Relationen ist trivial. Der Umstand, dass in beiden Modellen "Attribute" vorkommen lässt darauf schließen, dass sie auch 1:1 übernommen werden können. So kann man denn auch aus jedem Entity mit bestimmten Attributen eine Relation mit genau denselben Attributen machen. Das bedeutet auch, dass die Attribute der Relationen genau die gleichen Domänen haben wie die entsprechenden Attribute der Entities. Abbildung 2.17 verdeutlicht diesen Vorgang an einem Beispiel.

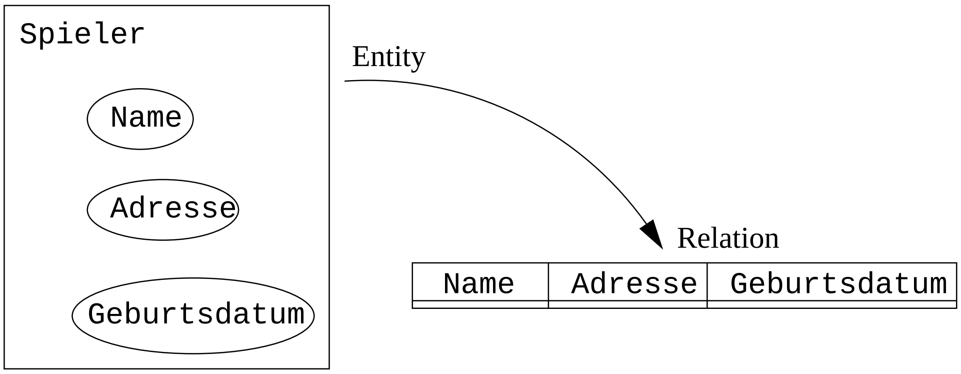


Abbildung 2.17: Umwandlung eines Entities in eine Relation

Da es unmittelbar nach der Erstellung einer Relation noch keine Daten gibt, die darin gespeichert sein könnten, ist sie naturgemäß leer.

Das Umwandeln von Entities in Relationen ist also problemlos möglich. Da ERMs aber nicht nur aus Entities, sondern auch aus Beziehungen (Relationships) bestehen, muss man auch Wege finden, die verschiedenen Grade von Beziehungen auf Relationen abzubilden.

### 2.6.2 Abbildung von 1:1 – Beziehungen

Bei der Auflösung von 1:1 – Beziehungen in Relationen ist es sinnvoll, die Mitgliedsklasse der Beziehung zu beachten, da für unterschiedliche Mitgliedsklassen unterschiedliche Auflösungsmethoden empfehlenswert sind.

### Auf beiden Seiten starke Beziehungen

Die einfachste und offensichtlichste Möglichkeit, 1:1 – Beziehungen auf Relationen abzubilden, besteht darin, die beiden in Beziehung stehenden Entities mitsamt ihren Attributen zu vereinigen. Ein Beispiel dafür gibt die Abbildung 2.18.

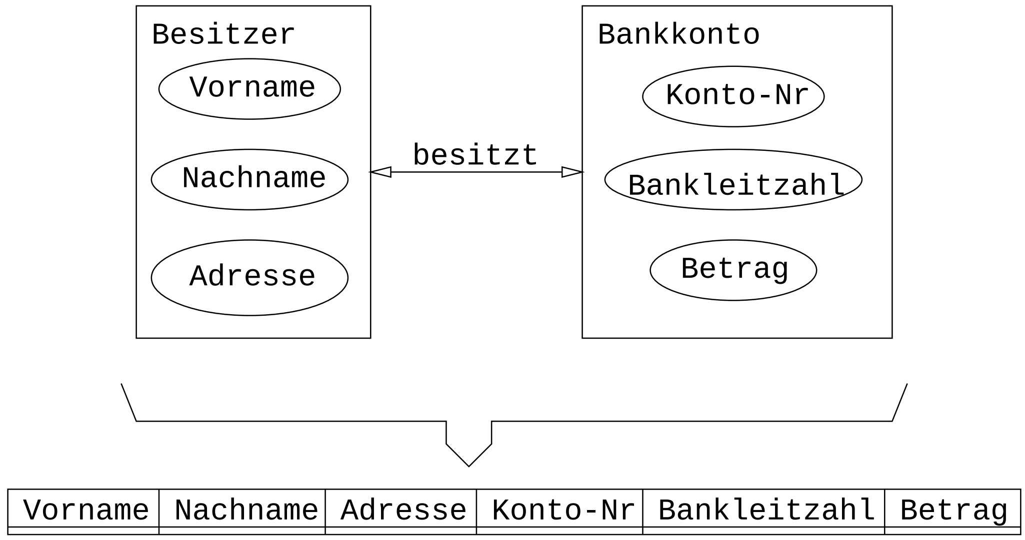


Abbildung 2.18: Auflösung einer 1:1 – Beziehung in *eine* Relation

Prinzipiell ist gegen eine solche Vorgehensweise nichts einzuwenden. Es ist aber schon aus Gründen der Übersichtlichkeit nicht günstig, die Attribute *zweier* verschiedener Entities in *einer* Relation zu speichern[20]. Deshalb empfehle ich, für jedes Entity eine eigene Relation zu erstellen und diese dann in der bereits beschriebenen Art und Weise mittels Foreign Keys zu verknüpfen. Abbildung 2.19 zeigt, wie das am obigen Beispiel ausgesehen hätte[21].

Wie man sieht, hat diese zweite Lösung den Nachteil, dass sie mehr Speicherplatz verbraucht, da das Attribut `Konto-Nr` zweimal gespeichert wird.

[20]Ein weiterer Grund ist, dass solche Relationen meist nicht 2NF sind (Nähere Informationen dazu: Kapitel 2.7: Normalisierung).

[21]Ich gehe hier stillschweigend davon aus, dass "Konto-Nr" der Primary Key von "Bankkonto" ist.

Besitzer

| Vorname | Nachname | Adresse | Konto-Nr |
| --- | --- | --- | --- |

Bankkonto

| Konto-Nr | Bankleitzahl | Betrag |
| --- | --- | --- |

Abbildung 2.19: Auflösung einer 1:1 – Beziehung in *zwei* Relationen

Das dürfte aber bei der Größe heutiger Speichermedien kaum noch ein Entscheidungskriterium sein.

Ein anderer Nachteil, der einen durchaus zur Anwendung der ersten Methode verleiten könnte, ist, dass bei dieser Methode die Sicherstellung der Datenintegrität zusätzlichen Aufwand erfordert. Zum Beispiel müsste man bei dem Beispiel in Abbildung 2.19 bei jeder Änderung des Attributs “Konto-Nr” eines Tupels aus der Relation “Besitzer” sicherstellen, dass dieser Wert kein zweites Mal in der Relation vorkommt, das sonst das betreffende Konto eine Beziehung zu zwei Besitzern hätte. Da es sich aber definitionsgemäß um eine 1:1 – Beziehung handelt, wäre dieser Zustand eine Inkonsistenz der Datenbank.

### Beziehungen, in denen mindestens eine obligatorische Mitgliedschft vorkommt

Bei allen 1:1 – Beziehungen, die nicht zwei obligatorische Mitgliedschaften enthalten, ergibt sich bei Anwendung der oben vorgestellten Lösung das Problem, dass Null – Werte nicht vermeidbar sind.

Nehmen wir wieder die Besitzer – Konto – Beziehung aus dem letzten Abschnitt als Beispiel und ändern wir es so ab, dass die Mitgliedschaft des Entities “Besitzer” nicht obligatorisch ist. Wenn dann ein Besitzer kein Konto besitzt, dann steht in der Attributspalte des entsprechenden Tupels der Relation “Besitzer” zwangsläufig ein Null – Wert (siehe Abbildung 2.20).

Um diese Null – Werte zu vermeiden, erstellt man eine neue Relation, die nur die beiden Primary Keys der in zu Beziehung setzenden Entities als Attribute enthält. Jedes Tupel dieser neuen Relation stellt dann eine Beziehung dar.

Um diese Methode auf das Bankbeispiel anzuwenden, empfiehlt es sich,

Besitzer

| Vorname | Nachname | Adresse | Konto-Nr |
|---|---|---|---|
| Heinz | Müller | hier | 2 |
| Rudolf | Meier | da | NULL |
| Klaus | Schulze | dort | 1 |

Bankkonto

| Konto-Nr | Bankleitzahl | Betrag |
|---|---|---|
| 1 | 1234 | 1 DM |
| 2 | 1234 | 3 DM |

Abbildung 2.20: Null – Werte bei schwachen 1:1 – Beziehungen

für das Entity "Besitzer" ein neues Attribut als Primary Key einzuführen. Wie die neuen Relationen dann aussehen, zeigt Abbildung 2.21.

Da es sich um eine 1:1 – Beziehung handelt, ist jedes Attribut aus `Besitzer-Konto` schon ein Candidate Key.

### 2.6.3 Abbildung von 1:n – Beziehungen

Auch bei 1:n – Beziehungen ist es sinnvoll, bei der Auflösung zwischen starken und schwachen Beziehungen zu unterscheiden.

#### Obligatorische Mitgliedschaft bei dem n – Entity

Mit "n – Entity" meine ich das Entity, welches Beziehungen zu mehreren Entities haben kann (weil es auf der "n – Seite" der 1:n – Beziehung steht)

Zur Auflösung einer solchen Beziehung wandelt man die Entities wie gewohnt in Relationen um und fügt auf der n – Seite der Beziehung noch einen Foreign Key hinzu. Damit ist die 1:n – Beziehung komplett abgebildet. Wenn die Mitgliedschaft bei dem 1 – Entity obligatorisch ist, muss man hier besonders darauf acht geben, dass auf jeden 1 – Tupel mindestens ein Foreign Key zeigt. Abbildung 2.22 verdeutlicht diese Vorgehensweise anhand der bereits angesprochenen 1:n – Beziehung zwischen Verlag und Buch (siehe Abbildung 2.12, Seite 19). Jedes der literarischen Meisterwerke hat ein Attribut namens `Verlag-ID`, dass als Foreign Key auf den entsprechenden Verlag zeigt.

Besitzer

| Kunden-ID | Vorname | Nachname | Adresse |
|---|---|---|---|
| 1 | Heinz | Müller | hier |
| 2 | Rudolf | Meier | da |
| 3 | Klaus | Schulze | dort |

Besitzer-Konto

| Kunden-ID | Konto-Nr |
|---|---|
| 1 | 2 |
| 3 | 1 |

Bankkonto

| Konto-Nr | Bankleitzahl | Betrag |
|---|---|---|
| 1 | 1234 | 1 DM |
| 2 | 1234 | 3 DM |

Abbildung 2.21: Schwache 1:1 – Beziehungen ohne Null – Werte

Verlag

| Verlag-ID | Name | Adresse |
|---|---|---|
| 1 | Reclam | Stuttgart |
| 2 | dtv | München |
| 3 | Telekom | überall |

Buch

| ISBN | Autor | Titel | Verlag-ID |
|---|---|---|---|
| 3150092531 | John Steinbeck | Of Mice and Men | 1 |
| 3423124008 | Goethe | Faust | 2 |
| 3150017874 | Hartmann v. Aue | Gregorius | 1 |
| NULL | einige... | Telefonbuch | 3 |
| 3423125551 | Jostein Gaarder | Sofies Welt | 2 |

Abbildung 2.22: Auflösung einer 1:n – Beziehung in Relationen

**Nicht obligatorische Mitgliedschaft bei dem n – Entity**

Bei nicht obligatorischen Mitgliedschaften ergibt sich wieder das Risiko von Null-Werten, das ja vermieden werden sollte. Deshalb erstellt man hier wie auch bei der 1:1 – Beziehung wieder eine eigene Beziehungsrelation.

### 2.6.4 Abbildung von m:n – Beziehungen

Bei m:n – Beziehungen ist es unabhängig von der Stärke der Beziehung ratsam, wie auch bei allen anderen schwachen Beziehungen eine eigene Beziehungsrelation zu erstellen.

Ich glaube, dass die Abbildung 2.21 auf Seite 27 diese Methode ausreichend verdeutlicht und werde deshalb keine weiteren Beispiele anführen.

## 2.7 Normalisierung

Ziel einer jeden Datenbankentwicklung ist es, ein stabiles Datenmodell zu erhalten, dass die Daten effizient strukturiert und dass zukünftigen Erweiterungen möglichst geringe Hürden in den Weg stellt. Die *Normalisierung* bringt die Datenstrukturen diesem Ziel ein großes Stück näher.

### 2.7.1 Update–, Insert– und Delete – Anomalien

Die im letzten Absatz angesprochene *effiziente Datenstrukturierung* bedeutet, dass die Datenstruktur alltägliche Operationen wie das Ändern, Löschen, Erstellen oder Suchen von Daten so weit wie möglich unterstützt. Die Aktionen dürfen also keine unnötig langen Rechenzeiten beanspruchen und es ist darauf zu achten, dass die Datenstruktur alle in der Realität vorkommenden Zustände ohne "Verrenkungen" abbilden kann.

Die Update–, Insert– und Delete – Anomalien treten nur bei Datenstrukturen auf, die diese Anforderungen nicht erfüllen.

Angenommen, ein kleines Versandhaus würde seine Bestelldaten in einer Datenbank abspeichern, wie sie in Abbildung 2.23 dargestellt ist.

Es gibt für Kunden und für lieferbare Artikel jeweils eine Relation. Gibt ein Kunde eine Bestellung auf, wird ein neues Tupel in der Relation "Bestellung" angelegt, in dem die entsprechende Kunden– und Artikelnummer und die Nummer des Kontos, von dem der zu bezahlende Betrag eingezogen wird, gespeichert werden (um die Kunden vor einem schädlichen Konsumwahn zu

Kunde

| Kunden-Nr | Vorname | Nachname | Adresse |
|---|---|---|---|
| 1 | Heinz | Müller | hier |
| 2 | Rudolf | Meier | da |
| 3 | Klaus | Schulze | dort |

Artikel

| Artikel-Nr | Bezeichnung | Preis |
|---|---|---|
| 1 | Windows 98 | 298 DM |
| 2 | Linux | 8 DM |

Bestellung

| Kunden-Nr | Artikel-Nr | Konto-Nr |
|---|---|---|
| 1 | 2 | 123123 |
| 3 | 1 | 456456 |
| 1 | 1 | 123123 |

Abbildung 2.23: Eine Beispieldatenbank für Bestelldaten

schützen, darf in diesem Versandhaus jede Bestellung nur einen Artikel umfassen).

Bei dem alltäglichen Betrieb dieser Datenbank können u.a. folgende Probleme auftreten:

- **Update – Anomalie:** Die Kontonummer eines Kunden ändert sich. Um diese Änderung in die Datenbank zu übernehmen, muss man die gesamte Relation "Bestellung" nach Tupeln durchsuchen, die diesem Kunden zuzuordnen sind und jedesmal die Kontonummer anpassen. Das Problem ist hier, dass in der Datenbank redundante Daten[22] vorkommen. Das hat zur Folge, dass man sehr viele Werte in der Datenbank ändern muss, wenn sich in der Realität nur ein einziger Sachverhalt ändert. Das macht die Durchführung solcher Änderungen wesentlich aufwändiger und damit langsamer. Außerdem besteht das Risiko einer Integritätsverletzung der Datenbank[23].

- **Insert – Anomalie:** Wenn das Versandhaus einen neuen Kunden in

[22]Gleiche Daten, die doppelt vorkommen.

[23]siehe Kapitel 7

ihre Datenbank aufnimmt, der noch keine Bestellung aufgegeben hat, dann gibt es keine Möglichkeit, die Kontonummer dieses Kunden zu speichern.

- **Delete – Anomalie:** Wenn ein Kunde seinen bestellten Artikel erhalten hat und das Geld dafür eingezogen ist, dann ist eine Bestellung abgeschlossen und man kann das ensprechende Tupel aus der Relation "Bestellung" löschen. War das jedoch die einzige Bestellung, die der Kunde gerade am Laufen hatte, so geht mit diesem Löschvorgang auch die Information über die Kontonummer des Kunden verloren.

Die Ursache für alle drei Probleme ist offensichtlich, dass die Datenbank Zusammenhänge nicht so abbildet, wie sie in der Realität bestehen [4, S. 164f.]. Die Kontonummer, die hier in der Relation "Bestellung" steht, hat in Wirklichkeit nichts mit der eigentlichen Bestellung zu tun, denn sie gehört zum Kunden.

Wie man sieht, führt eine Nichtbeachtung der *Abhängigkeiten* zwischen den Attributen zu Problemen wie den Update–, Insert– und Delete – Anomalien. Bei der *Normalisierung* versucht man, z.B. durch Analyse dieser Abhängigkeiten und darauf folgende Modifizierungen eine Datenstruktur weniger anfällig für solche Probleme zu machen.

Es gibt verschiedene Grade der Normalisierung, in denen sich Datenstrukturen befinden können. Sie werden *Normalformen* genannt.

Die Normalformen "vererben" ihre Eigenschaften. Eine Relation, die in der zweiten Normalform ist, muss also auch in der ersten Normalform sein u.s.w.

### 2.7.2 Die erste Normalform (1NF)

Eine Datenstruktur kann erst in einem relationalen Datenbanksystem implementiert werden, wenn sie sich in der ersten Normalform befindet. Die erste Normalform gehört also zur Definition einer relationalen Datenbank und nimmt damit eine Sonderstellung unter den Normalformen ein. Für die erste Normalform gilt auch nicht das, was ich über Abhängigkeiten zwischen Attributen im Zusammenhang mit Normalisierung gesagt habe.

Eine Datenstruktur befindet sich in der ersten Normalform (1NF), wenn folgendes gilt (vgl. u.a. [1, Kap. 3.1]):

- **Jedes Attribut eines Tupels hat einen eindeutigen Namen.** Innerhalb eines Tupels dürfen sich also keine gleichen Attribute wiederholen. Angenommen, eine Lagerverwaltung möchte die Lagerbestandsdaten von Teilen speichern. Gleiche Teile können an verschiedenen Orten gelagert werden. Nun könnte man auf die Idee kommen, eine Relation wie die in Abbildung 2.24 dargestellte zu verwenden. Diese Relation hat die offensichtlichen Nachteile, dass Nullwerte vorkommen können und dass Teile, die an mehr als zwei Orten gelagert werden, zu Problemen führen. Die Relation steht nicht in der ersten Normalform.

`Lagerbestände`

| Teile-Nr | Name | L-Ort | L-Best | L-Ort | L-Best |
|---|---|---|---|---|---|
| 1 | Nagel | L1 | 1000 | L3 | 500 |
| 2 | Schraube | L2 | 200 | ? | ? |
| 3 | Dübel | L3 | 350 | L2 | 20 |

Abbildung 2.24: Eine nicht normalisierte Relation

- **An jedem Schnittpunkt zwischen Zeilen und Spalten in einer Relation muss genau ein atomarer Wert stehen.** Atomare Werte sind Daten, die nicht weiter sinnvoll zerlegt werden können.

  Eine andere Möglichkeit, den Lagerbestand in einer Relation zu speichern, beinhaltet nicht – atomare Werte und ist in Abbildung 2.25 dargestellt (diese Relation ist auch nicht in 1NF).

`Lagerbestände`

| Teile-Nr | Name | L-Ort | L-Best |
|---|---|---|---|
| 1 | Nagel | L1, L3 | 1000, 500 |
| 2 | Schraube | L2 | 200 |
| 3 | Dübel | L3, L2 | 350, 20 |

Abbildung 2.25: Eine (zweite) nicht normalisierte Relation

Eine Relation, die doppelte Attribute und nicht – atomare Werte vermeidet (die also 1NF ist), zeigt Abbildung 2.26.

Lagerbestände

| Teile-Nr | Name | L-Ort | L-Best |
|---|---|---|---|
| 1 | Nagel | L1 | 1000 |
| 2 | Schraube | L2 | 200 |
| 3 | Dübel | L3 | 350 |
| 1 | Nagel | L3 | 500 |
| 3 | Dübel | L2 | 20 |

Abbildung 2.26: Relation in der ersten Normalform

- **Jedes Tupel besitzt einen Primary Key.** Alle Relationen, die in 1NF stehen, besitzen einen Primary Key. Das heißt auch, dass sie mindestens einen Super Key haben, was ja wegen des Begriffs "Relation" eigentlich gar nicht mehr erwähnenswert ist. Die Relation aus der Abbildung 2.26 hat einen Concatinated Primary Key, bestehend aus den Attributen `Teile-Nr` und `L-Ort`.

- **Die Datenstruktur muss von der Art ihrer physischen Speicherung unabhängig sein.** Das schließt zum Einen die bereits erwähnte Beschränkung ein, dass weder die Spalten- noch die Zeilenreihenfolge in einer Relation von Bedeutung sein darf. Außerdem folgt daraus, dass Beziehungen zwischen Tupeln nie über physische Adresszeiger realisiert werden dürfen, sondern immer, wie ich bereits beschrieben habe, über Schlüssel.

  Um dieses Ziel zu erreichen, folgen alle relationalen Datenbanksysteme dem *Dreischichtenmodell*, das erstmals im Jahre 1975 von der *ANSI/SPARC Study Group on Data Base Management Systems* vorgeschlagen wurde. Dieses Modell verlangt, dass die Verwaltung einer Datenbank in drei Ebenen aufgeteilt wird:

  - Physische (interne) Sicht
  - Konzeptionelle (logische) Sicht
  - Externe Ebene (Benutzersicht)

  Diese drei Ebenen sollen auch von drei voneinander getrennten, verschiedenen Programmteilen des DBMS übernommen werden.

Die physische Ebene ist für die Speicherung der Daten im Hauptspeicher und auf den Massenspeichern zuständig und hat die Art und Weise dieser Speicherung vollständig vor der darüberliegenden Schicht zu verstecken.

Die konzeptionelle Ebene ist die logische Gesamtsicht auf die Datenbank. Sie ist genau das, was in diesem Kapitel entwickelt wird.

Die externe Ebene ist dafür zuständig, eine Schnittstelle zwischen dem Benutzer und der Datenbank anzubieten. Sie soll dem Benutzer nur einen kleinen, überschaubaren Teil der Datenbank präsentieren.

Dieses Dreischichtenmodell sorgt dafür, dass die Art der physischen Speicherung der Daten völlig irrelevant für die logische Gesamtsicht auf die Datenbank wird. Deshalb ist das Verwenden von (physischen) Adresspointern in relationalen Datenbanken gar nicht erst möglich. Es bleibt also nur der "Ausweg" über (logische) Schlüssel.

### 2.7.3 Die zweite Normalform (2NF)

Um die zweite Normalform zu verstehen, müssen zuerst die Begriffe *funktionale Abhängigkeit* und *voll funktionale Abhängigkeit* geklärt werden [10, Kap. 9.1.2].

- Eine **funktionale Abhängigkeit** zwischen den Attributen $A$ und $B$ einer Relation besteht dann, wenn sich jedem Wert von $A$ genau ein Wert von $B$ zuordnen lässt. Man sagt: *$B$ ist von $A$ funktional abhängig.*

  Eine solche Abhängigkeit besteht zum Beispiel zwischen den Attributen `Personalausweis-Nr` und `Name` einer Person. Jeder Ausweisnummer lässt sich genau ein Name zuordnen. Der Name ist also funktional von der Ausweisnummer abhängig. Die Ausweisnummer ist aber nicht von dem Namen funktional abhängig, da es ja durchaus verschiedene Personen mit gleichem Namen geben soll, die aber unterschiedliche Ausweisnummern haben.

- Eine **voll funktionale Abhängigkeit** besteht zum Beispiel zwischen den Attributen $A_1$ und $A_2$ und dem Attribut $B$ einer Relation dann, wenn $B$ von dem zusammengesetzten Attribut $A_1$ und $A_2$ funktional abhängig ist, nicht aber von den einzelnen Attributen $A_1$ oder $A_2$.

> So ist beispielsweise der sich auf einem Konto befindende Geldbetrag voll funktional abhängig von der Kontonummer und der Bankleitzahl, da ein Konto erst mit Hilfe dieser beiden Werte eindeutig identifiziert werden kann. Weder die Kontonummer noch die Bankleitzahl alleine reichen aus, um den Geldbetrag zuzuordnen.
>
> Ein Attribut $B$ kann auch von mehr als zwei $A_n$ – Attributen abhängig sein. Auch hier gilt wieder, dass $B$ von keiner Teilmenge der $A_n$ – Attributmenge funktional abhängig sein darf, nur von der ganzen Menge.
>
> Umfasst die Attributmenge nur ein Attribut und ist $B$ von diesem Attribut funktional abhängig, so ist $B$ auch von dieser ein – elementigen Attributmenge voll funktional abhängig.

Eine Relation ist in der zweiten Normalform, wenn jedes ihrer Nicht – Schlüssel – Attribute voll funktional abhängig vom Primary Key ist [1, Kap. 3.2].

Bei Primary Keys, die nur aus einem Attribut bestehen, versteht sich das meist von selbst. Vorsicht ist aber bei Concatinated Primary Keys geboten.

Die in Abbildung 2.26 (Seite 32) dargestellte Relation hat einen Concatinated Primary Key, der aus den Attributen `Teile-Nr` und `L-Ort` besteht. Das Attribut `Name` ist nicht voll funktional von diesem Primary Key abhängig. Deshalb steht die Relation nicht in der zweiten Normalform.

Um die Relation in die zweite Normalform zu überführen, muss man die Attribute, die nicht voll funktional vom Primary Key abhängig sind, in andere Relationen auslagern. Wie das bei dem oben genannten Beispiel aussieht, kann man Abbildung 2.27 entnehmen.

Die Überführung in die zweite Normalform hatte hier offensichtlich den Vorteil, dass die Update–, Insert– und Delete – Anomalien des Attributs `Name` vermieden wurden.

### 2.7.4 Die dritte Normalform (3NF)

Für die dritte Normalform ist der Begriff "transitive Abhängigkeit" von Bedeutung [1, Kap. 3.3].

In einer Relation mit den Attributen $A$, $B$ und $C$ ist das Attribut $C$ transitiv von $A$ abhängig, wenn $B$ von $A$ voll funktional abhängig ist, $A$ aber nicht von $B$, und $C$ von $B$ funktional abhängig ist.

Lagerbestände

| Teile-Nr | L-Ort | L-Best |
|---|---|---|
| 1 | L1 | 1000 |
| 2 | L2 | 200 |
| 3 | L3 | 350 |
| 1 | L3 | 500 |
| 3 | L2 | 20 |

Teile

| Teile-Nr | Name |
|---|---|
| 1 | Nagel |
| 2 | Schraube |
| 3 | Dübel |

Abbildung 2.27: Das Lagerbeispiel in der zweiten Normalform

Wenn man voll funktionale Abhängigkeiten mit einem Pfeil darstellt, sieht das zugehörige ERM wie in Abbildung 2.28 aus.

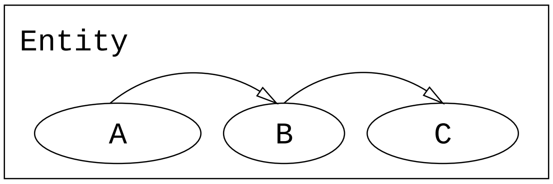


Abbildung 2.28: transitives Abhängigkeitsverhältnis im ERM

Ein konkretes Beispiel für eine transitive Abhängigkeit ist in Abbildung 2.29 dargestellt. In der Entity `Buch` soll die Anschrift des Verlages gespeichert werden. Dabei ergibt sich eine transitive Abhängigkeit zwischen der ISBN (Primary Key) und der Verlagsanschrift über den Verlagsnamen.

Relationen, die in der dritten Normalform stehen, dürfen keine transitiven Abhängigkeiten zwischen ihren Attributen aufweisen.

Um die Beispielentity aus Abbildung 2.29 in die dritte Normalform zu überführen, lohnt es sich wieder, die "störenden" Attribute in neue Entities/Relationen auszulagern (siehe Abbildung 2.30).

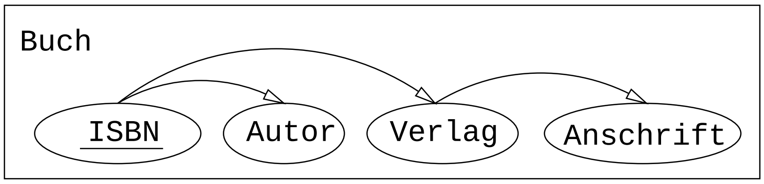


Abbildung 2.29: Ein Beispiel für transitive Abhängigkeit

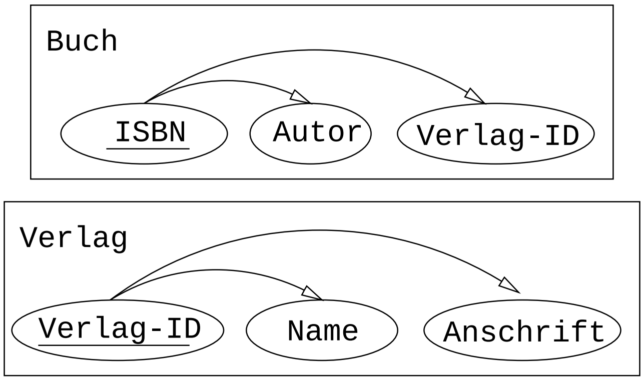


Abbildung 2.30: Entities in der dritten Normalform

### 2.7.5 weitere Normalformen

Es gibt noch weitere Normalformen, namentlich z.B. die Boyce – Codd – Normalform (BCNF), die vierte und die fünfte Normalform, auf die ich hier aber nicht näher eingehen werde.

Diese weiteren Normalformen werden zum Beispiel in [12] beschrieben.

In der Praxis wird meist nur bis zur dritten Normalform normalisiert, da sich bei den weiteren Normalformen das Verhältnis zwischen Aufwand und realem Nutzen verschlechtert.

# Kapitel 3

# Die Schemadefinition mit SQL

Im Kapitel 2 wurde beschrieben, wie man das relationale Datenschema entwickeln kann. Dieses Schema besteht aus einer Anzahl von Relationen, die in Beziehung zueinander stehen können. Dieses Kapitel thematisiert, wie man das logische Schema in eine funktionsbereite Datenbank verwandeln kann.

Dazu muss zuerst die (bisher noch nicht getroffene) Entscheidung fällen, welches DBMS zum Einsatz kommen soll. Es gibt auf dem Markt eine unübersehbare Vielfalt von verschiedenen DBMS, die sich alle durch Preis, Qualität und Funktionsumfang unterscheiden. *dBase*, *Oracle* und *Access* sind einige der populärsten nicht – freien Programmpakete. Für den schmalen Geldbeutel (oder für Anhänger freier Software) empfiehlt sich *MySQL*.

Um die Datenbank zu implementieren, muss man dem DBMS eine Beschreibung des logischen Datenbankschemas übermitteln. Bei der Wahl der hierfür benutzten Sprache geht fast jedes DBMS seine eigenen Wege. Sehr viele Systeme benutzen jedoch den Datendefinitionsteil der Sprache *SQL* dafür [11].

## 3.1 Die Geschichte von SQL

Anfang der siebziger Jahre, kurz nachdem das relationale Datenmodell erstmals vorgestellt worden war, wurde eine IBM – Forschungsgruppe beauftragt, die Effizienz und Benutzbarkeit des neuen Datenmodells zu überprüfen. Im Rahmen dieses Auftrags entwickelten sie ein prototypisches DBMS, das sie *System R* nannten. Dieses DBMS enthielt die erste Version von SQL, die damals noch *SEQUEL2* genannt wurde. 1976 wurde die komplette Sprachspezifikation von SQL im *IBM Journal of R&D* veröffentlicht. Im Jahr 1979 brachte die Firma *Oracle Corporation* das erste kommerzielle DBMS auf den Markt, das SQL unterstützte [11]. 1987 wurde ein ANSI – Standard für SQL eingeführt [10, Kap. 3.1].

Inzwischen hat sich SQL zu einem Quasi – Standard für Abfragesprachen in relationalen Datenbankmanagementsystemen entwickelt.

Es scheint nicht ganz eindeutig zu sein, wofür dei Abkürzung SQL steht. Die beiden am weitesten verbreiteten Versionen sind "Structured Query Language[1]" und "Standard Query Language". Microsoft hat eine eigene Version: "Source Query Language"[2].

## 3.2 Die Struktur von SQL

Entgegen der landläufigen Bezeichnung ist SQL keine reine "Abfragesprache". Die Anweisungen von SQL lassen sich zwei Gruppen zuordnen [4]:

- **Datendefinitionssprache (DDL):** Diese Anweisungen dienen zur Definition des logischen Datenschemas.
- **Datenmanipulationssprache (DML):** Diese Anweisungen dienen zur Manipulation von bestehenden Relationen.

Auf die Datenmanipulationssprache gehe ich in Kapitel 4 ein.

Die folgenden Unterkapitel stellen die Möglichkeiten der Datendefinitionssprache vor.

## 3.3 Erstellen von Relationen

In SQL kann man neue Relationen mit dem Befehl `create table` erstellen nach dem Schema[3] [10, Kap. 3.4.1]:

```
create table name (attr1 dom1 [, attr2 dom2 [,[...]]])
```

[1] Query Language = Abfragesprache

[2] Aus einer Werbung für den Microsoft – SQL Server

[3] SQL achtet, zumindest bei den Schlüsselwörtern, nicht af die Groß- oder Kleinschreibung. Deshalb trifft man auch sehr oft auf GROSSGESCHRIEBENE SQL–Anweisungen.

`name` ist der Name der Relation. `attrx` sind die Namen der Attribute, die in dieser Relation vorkommen. SQL ist nicht voll relational, d.h. es gibt einige Abweichungen zum relationalen Modell. So ist das auch hier, denn die Reihenfolge in der man die Attribute bei diesem Befehl angibt, ist für andere Befehle, wie z.B. `insert`, von Bedeutung. `domx` bezeichnet die Domäne, die dem jeweiligen Attribut zugeordnet werden soll.

Um beispielsweise die Relation aus Abbildung 3.1 zu erstellen, könnte man folgenden Befehl verwenden:

```
create table Person
    ( Kunden-ID smallint,
      Vorname   char(12),
      Nachname  char(20),
      Adresse   char(35) )
```

Dieses Beispiel zeigt, dass man SQL – Anweisungen zur besseren Übersichtlichkeit auch auf mehrere Zeilen aufteilen kann.

Person

| Kunden-ID | Vorname | Nachname | Adresse |
|---|---|---|---|

Abbildung 3.1: Beispielrelation für die `create table` – Anweisung

## 3.4 Erstellen von Views

### 3.4.1 Das View – Konzept

Wie bereits in dem Kapitel 2.7.2 beschrieben, gibt es in einem DBMS drei Ebenen, die alle eine unterschiedliche Sicht auf die Daten haben [1, Kap. 3.4.1]. Die konzeptionelle Ebene enthält die logische Gesamtsicht auf die Daten. Es gibt aber noch eine darüber liegende Ebene, die jedem Benutzer einen auf ihn zugeschnittenen Teil der Datenbank präsentieren soll. Das hat hauptsächlich den Vorteil, dass die Arbeit mit der Datenbank benutzerfreundlicher wird, da der Anwender nur die Daten auf dem Bildschirm sieht, die er wirklich zu seiner Arbeit beötigt. Um diese benutzerspezifischen Sichten auf die Datenbank zu ermöglichen, gibt es in SQL die sogenannten *Views*. Views kann man sich vorstellen als spezielle Relationen, die aber nur auf der externen Ebene existieren. Sie repräsentieren Teile aus einer oder mehreren Relationen der konzeptionellen Ebene.

### 3.4.2 Die `create view` – Anweisung

Views werden wie folgt definiert [10, Kap. 3.4.3]:

```
create view name ( attr1 [, attr2 [, [...]]]) as
  <select-Anweisung>
```

`name` ist hier wieder der Name der zu erstellenden View. `attrx` sind die Attribute, die die View – Relation haben soll. Die `select-Anweisung` gibt dem DBMS Aufschluss darüber, welche Daten in der View enthalten sein sollen. Die genaue Funktion von `select` – Anweisungen wird in Kapitel 4.3.1 beschrieben.

Um beispielsweise aus der in Abbildung 3.2 dargestellten Relation eine View zu erstellen, die nur die Kundennummern und die Telefonnummern aller Kunden enthält, könnte man folgende Anweisung benutzen:

```
create view Tel-Nummern ( Kunden-ID, Telefonnummer ) as
select Kunden-ID, Telefonnummer from Person
```

Person

| Kunden-ID | Vorname | Nachname | Adresse | Telefonnummer |
|---|---|---|---|---|

Abbildung 3.2: Relation für das View – Beispiel

Tel-Nummern

| Kunden-ID | Telefonnummer |
|---|---|

Abbildung 3.3: Die View `Tel-Nummern`

## 3.5 Der Befehl `grant`

Der SQL – Befehl `grant` erlaubt es, den Benutzern der Datenbank bestimmte Zugriffsrechte auf Relationen, Views und deren Attribute zu erteilen [10, Kap. 3.4.2]. So lässt sich festlegen, dass ein Benutzer bestimmte Daten nur lesen, nicht aber verändern darf. Andere Daten wiederum darf er nicht einmal lesen. Zusammen mit dem View – Konzept lässt sich so eine detaillierte Rechtevergabe an jeden Benutzer realisieren.

Die Möglichkeit einer solchen Rechtevergabe ist auch unbedingt notwendig. In einer Datenbank werden oft riesige Mengen an Daten gespeichert, auf die aus Gründen des Datenschutzes nicht jeder Zugriff haben soll.

# Kapitel 4

# Relationale Abfragesprachen

Jedes DBMS bietet der Außenwelt (das ist entweder der Benutzer oder ein Anwendungsprogramm) Möglichkeiten, die gespeicherten Informationen zu erfahren. Die dazu erforderliche Interaktion mit dem DBMS findet über sogenannte *Abfragesprachen* statt. Es gibt eine Vielzahl verschiedener Abfragesprachen. Alle lassen sich jedoch einer von drei Gruppen zuordnen [3, Kap. 3]:

- **Sprachen auf Basis der relationalen Algebra.** Diese Sprachen bieten dem Nutzer die Möglichkeit, die Operationen der relationalen Algebra auf die Datenbank anzuwenden. Die relationale Algebra umfasst die aus der Mathematik bekannten Mengenoperationen und einige weitere Operationen. Einen Einblick in die relationale Algebra gibt Kapitel 4.1.

- **Relationale Kalkülsprachen.** Das *relationale Kalkül* basiert im Wesentlichen darauf, vorhandene Relationen unter Anwendung bestimmter Regeln (*Bedingungen*) auf neue Relationen abzubilden. *Abbildungsorientierte Sprachen* (*mapping – oriented languages*) sind ein Versuch, relationale Kalkülsprachen für den Benutzer einfacher erlernbar zu machen. Der Datenmanipulationsteil von *SQL* ist eine solche Sprache. Auf das relationale Kalkül und SQL gehe ich ab Kapitel 4.2 ein.

- **Abfrage nach Muster (*query by example*).** Bei diesen Sprachen findet die Abfrage dadurch statt, dass der Benutzer in ein vorgefertigtes Formular einige Werte einträgt und das DBMS die fehlenden Werte nach bestem Vermögen automatisch ergänzt. Diese Art von Sprache ist sehr benutzerfreundlich, jedoch bereitet die Entwicklung einer solchen Sprache viel Mühe und sie sind im Vergleich zu den anderen beiden Spracharten wenig leistungsfähig, da sie bei komplexen Anfragen schnell versagen.

## 4.1 Die relationale Algebra

Genauso, wie die Mathematik Operationen zur Manipulation von Zahlen anbietet (Addition, Multiplikation...), gibt es auch in der Theorie relationaler Datenbanken einige "Grundrechenarten", mit denen Relationen verändert werden können. Diese entstammen zum Teil aus der Mengenlehre der Mathematik:

- Vereinigung
- Durchschnitt
- Differenz

Der Grund dafür ist, dass eine Relation auch als eine *Menge* von Tupeln aufgefasst werden kann [2, S. 29].

Andere Operationen kommen jedoch so nicht in der Mengenlehre vor:

- Umbenennung
- Selektion
- Projektion
- Kartesiches Produkt
- Division
- Verbindung (Join)

Im Folgenden werde ich jede Rechenoperation in einem eigenen Unterkapitel beschreiben.

### 4.1.1 Umbenennung

Um die Operationen der Mengenlehre auf Relationen anwenden zu können, müssen diese *vereinigungskompatibel* sein. D.h., sie müssen die gleiche Anzahl von Attributen haben, die gleiche Namen und gleiche Domains haben. Die Relationen müssen also den gleichen *Degree* besitzen.

Die *Umbenennung* ist ein oft vergessenes, hierzu aber unschätzbares Hilfsmittel [9, S. 65]. Es erlaubt, die Namen von Attributen zu ändern (siehe Abbildung 4.1).

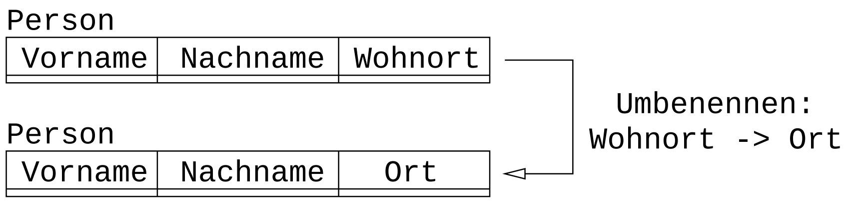


Abbildung 4.1: Ein Beispiel für *Umbenennung*

### 4.1.2 Vereinigung

Die *Vereinigung* (auch: Verbindung, Union) vereinigt die Tupelmengen zweier vereinigungskompatibler Relationen. Die Ergebnisrelation hat den gleichen Degree wie die Ausgangsrelationen und enthält alle Tupel, die in mindestens einer Ausgangsrelation vorkommen. Aufgrund der Bedingungen für die erste Normalform werden in der Ergebnisrelation doppelt vorkommende Tupel vermieden (siehe Abbildung 4.2).

### 4.1.3 Durchschnitt

Beim *Durchschnitt* (engl: *Intersection*) wird die Schnittmenge der Tupelmengen zweier vereinigungskompatibler Relationen gebildet. Die Ergebnisrelation hat den gleichen Degree wie die Ausgangsrelationen und enthält alle Tupel, die in beiden Ausgangsrelationen gleichermaßen enthalten sind (siehe Abbildung 4.3).

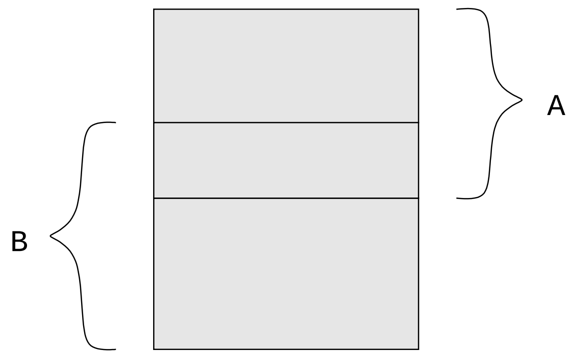


Abbildung 4.2: *Vereinigung*: A vereinigt B

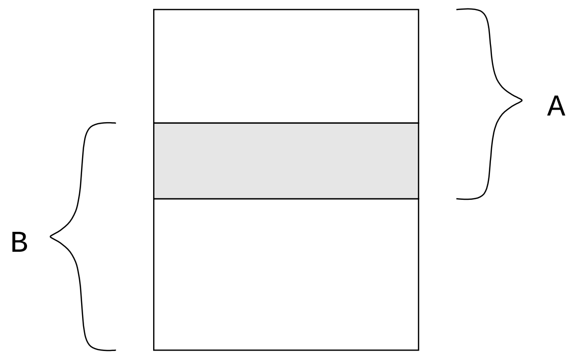


Abbildung 4.3: *Durchschnitt*: A geschnitten B

### 4.1.4 Differenz

Die Ergebnisrelation bei der *Subtraktion* ist die Differenz der beiden (vereinigungskompatiblen) Ausgangsrelationen. Bei der Differenz $A - B = C$ enthält $C$ alle Tupel, die in $A$ enthalten sind, die aber nicht Element von $B$ sind (siehe Abbildung 4.4).

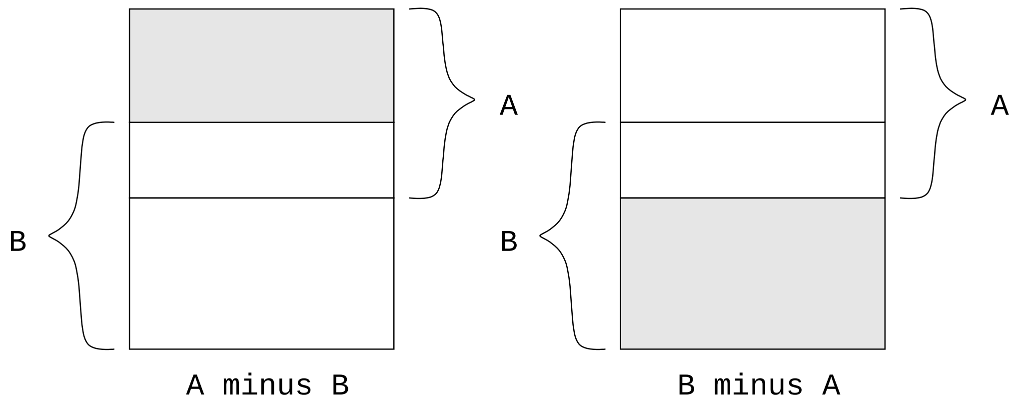


Abbildung 4.4: *Differenz*: $A$ minus $B$ und $B$ minus $A$

### 4.1.5 Selektion

Bei der *Selektion* (auch: *Auswahl*; engl: *Restriction*) wählt man nach einem bestimmten Kriterium Tupel aus einer Relation aus. Die Ergebnisrelation enthält also eine Teilmenge der Tupel der Ausgangsrelation (siehe Abbildung 4.5). Es ist beispielsweise möglich, aus der Relation `Person` (siehe Abbildung 4.1) alle Personen zu extrahieren, die mit Nachnamen "Schuhmacher" heißen, indem man eine Selektion mit dem Kriterium `Nachname = ''Schuhmacher''` durchführt. Das Kriterium ist eine mathematische Aussage und kann mit den Logik – Operatoren wie zum Beispiel "`und`", "`oder`" oder "`nicht`" verknüpft werden.

### 4.1.6 Projektion

Die *Projektion* (engl. *Projection*) wird benutzt, um aus einer Relation bestimmte Attribute zu extrahieren (siehe Abbildung 4.6). Zur Bestimmung der Attribute muss man deren Namen angeben.

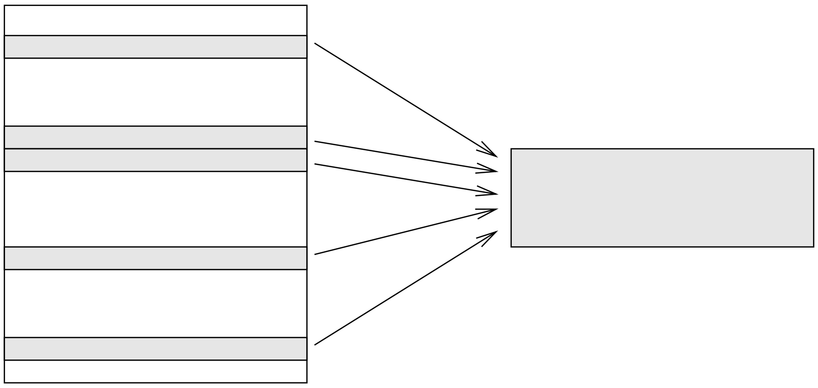

Abbildung 4.5: Selektion

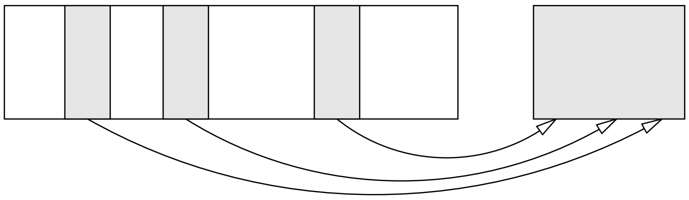

Abbildung 4.6: Projektion

### 4.1.7 Kartesisches Produkt

Das *kartesische Produkt* kombiniert die Tupel zweier Relationen auf alle möglichen Arten miteinander. Der *Degree* der entstehenden Relation ist die Summe der Degrees der Ausgangsrelationen. Die Kardinalität der Ergebnisrelation ist das Produkt der Kardinalitäten der Ausgangsrelationen.

Abbildung 4.7 zeigt das kartesische Produkt anhand eines allgemeinen Beispieles.

### 4.1.8 Division

Die *Division* ist die Umkehroperation zum kartesischen Produkt. Das Ergebnis der Operation $C : B = A$ ist also genau die Relation, die, mit $B$ multipliziert, die größtmögliche Teilmenge von $C$ ergibt[1]. Im "Idealfall" gilt also:

$$C = A \times B$$

Alle anderen Möglichkeiten entsprechen einer Division mit Rest, wie man sie aus dem Bereich der Ganzen Zahlen ($Z$) kennt. Dann gilt:

$$C = A \times B + R$$

wobei $R$ der "Rest" ist, also die Menge der Tupel, die Element von $C$ sind, die aber nicht Element der Relation $A \times B$ sind.

#### Beispiel für Division

In der Datenbank einer Schule gibt es zwei Relationen. Sie beschreiben alle Kurse, die an der Schule laufen und alle Schüler der Schule (siehe Abbildung 4.8).

Der Schulleiter möchte wissen, welche Schüler alle angebotenen Kurse belegen.

Lösung:

[1] Das ist eine leicht eingeschränkte Definition der Division. Bei der "richtigen" Division kann man angeben, unter Berücksichtigung welcher Attribute dividiert werden soll. Das erspart z. B. die Projektion, die ich in dem Beispiel zur Division anwende. Eine exakte Definition der Division erfordert eine Einführung weiterer Begriffe, die hier den Rahmen sprengen würde (genaue Informationen: siehe [1, Seite 70ff.]).

A

| | | | | | |
|---|---|---|---|---|---|
| $a_{11}$ | $a_{12}$ | $a_{13}$ | . | . | $a_{1p}$ |
| $a_{21}$ | $a_{22}$ | $a_{23}$ | . | . | $a_{2p}$ |
| . | . | . | | | . |
| . | . | . | | | . |
| $a_{m1}$ | $a_{m2}$ | $a_{m3}$ | . | . | $a_{mp}$ |

B

| | | | | | |
|---|---|---|---|---|---|
| $b_{11}$ | $b_{12}$ | $b_{13}$ | . | . | $b_{1q}$ |
| $b_{21}$ | $b_{22}$ | $b_{23}$ | . | . | $b_{2q}$ |
| . | . | . | | | . |
| . | . | . | | | . |
| $b_{n1}$ | $b_{n2}$ | $b_{n3}$ | . | . | $b_{nq}$ |

A x B

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_{11}$ | $a_{12}$ | $a_{13}$ | . | . | $a_{1p}$ | $b_{11}$ | $b_{12}$ | $b_{13}$ | . | . | $b_{1q}$ |
| $a_{11}$ | $a_{12}$ | $a_{13}$ | . | . | $a_{1p}$ | $b_{21}$ | $b_{22}$ | $b_{23}$ | . | . | $b_{2q}$ |
| . | . | . | | | . | . | . | . | | | . |
| . | . | . | | | . | . | . | . | | | . |
| $a_{11}$ | $a_{12}$ | $a_{13}$ | . | . | $a_{1p}$ | $b_{n1}$ | $b_{n2}$ | $b_{n3}$ | . | . | $b_{nq}$ |
| $a_{21}$ | $a_{22}$ | $a_{23}$ | . | . | $a_{2p}$ | $b_{11}$ | $b_{12}$ | $b_{13}$ | . | . | $b_{1q}$ |
| $a_{21}$ | $a_{22}$ | $a_{23}$ | . | . | $a_{2p}$ | $b_{21}$ | $b_{22}$ | $b_{23}$ | . | . | $b_{2q}$ |
| . | . | . | | | . | . | . | . | | | . |
| . | . | . | | | . | . | . | . | | | . |
| $a_{21}$ | $a_{22}$ | $a_{23}$ | . | . | $a_{2p}$ | $b_{n1}$ | $b_{n2}$ | $b_{n3}$ | . | . | $b_{nq}$ |
| | | . | | | | | | . | | | |
| | | . | | | | | | . | | | |
| $a_{m1}$ | $a_{m2}$ | $a_{m3}$ | . | . | $a_{mp}$ | $b_{11}$ | $b_{12}$ | $b_{13}$ | . | . | $b_{1q}$ |
| $a_{m1}$ | $a_{m2}$ | $a_{m3}$ | . | . | $a_{mp}$ | $b_{21}$ | $b_{22}$ | $b_{23}$ | . | . | $b_{2q}$ |
| . | . | . | | | . | . | . | . | | | . |
| . | . | . | | | . | . | . | . | | | . |
| $a_{m1}$ | $a_{m2}$ | $a_{m3}$ | . | . | $a_{mp}$ | $b_{n1}$ | $b_{n2}$ | $b_{n3}$ | . | . | $b_{nq}$ |

Abbildung 4.7: Kartesisches Produkt: $A \times B$

Schüler

| Schüler-ID | Kurs-ID |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 1 | 3 |
| 3 | 1 |
| 2 | 3 |
| 2 | 2 |
| 1 | 1 |
| 3 | 3 |

Kurs

| Kurs-ID | Bezeichnung | Raum |
|---|---|---|
| 1 | PW | 112 |
| 2 | Kunst | 110 |
| 3 | Physik | 101 |

Abbildung 4.8: Beispieldatenbank einer Schule (Auszug :-) )

```
Division (Schüler, Projektion (Kurs, Kurs-ID))
```

Es wird also die Relation `Schüler` durch die von allen Attributen außer `Kurs-ID` "bereinigte" Relation `Kurs` dividiert.

Das Ergebnis dieser Operation zeigt Abbildung 4.9.

Ergebnis

| Schüler-ID |
|---|
| 1 |
| 2 |

Abbildung 4.9: Das Ergebnis der Division

### 4.1.9 Verbindung (Join)

Der *Join* (auch: *Verbund*, *Verbindung*) ist die wohl am meisten benutzte Rechenoperation der relationalen Algebra. Ein Join dient wie das kartesische Produkt dazu, zwei Relationen zu einer Relation mit höherem Degree zu verknüpfen. Anders als beim Produkt ist es beim Join Voraussetzung, dass es in beiden Relationen je ein Attribut mit gleichem Wertebereich gibt. Beim Join findet eine Verknüpfung zwischen zwei Tupeln nur dann statt, wenn diese beiden Attribute den gleichen Wert haben.

In der Ergebnisrelation des Joins gibt es also zwei Attribute, die genau die gleichen Werte haben. Das wird bei einer speziellen Variante des Joins, dem *Natural Join* (natürlicher Verbund), vermieden, da er bei der Verknüpfung der Tupel je eines der Attribute, nach denen verknüpft wird, weg lässt.

Eine allgemeinere Form des Joins ist der $\theta$ – *Join*[2]. $\theta$ repräsentiert hier einen der mathematischen Vergleichsoperatoren $=, <, >, \leq, \geq$ oder $\neq$. Die Tupel werden nur verknüpft, wenn der Ausdruck $attr1\ \theta\ attr2$ wahr ist.

Der Join ist eigentlich keine Grundrechenart in der relationalen Algebra, denn er lässt sich in jedem Fall auch definieren als die Projektion aus einer Selektion aus dem Produkt der Ausgangsrelationen [1, S. 68].

Die Abbildung 4.10 zeigt die verschiedenen Join – Arten an dem Beispiel aus Abbildung 4.8. Die Joins wurden bezüglich der Attribute `Kurs-ID` aus beiden Relationen durchgeführt.

## 4.2 Das relationale Kalkül

Eine Datenbankabfrage mit dem *relationalen Kalkül* hat grundsätzlich die Form einer Mengendefinition [2, S. 34]. Das kann zum Beispiel so aussehen:

$$\{\, t \mid A(t) \,\}$$

$t$ repräsentiert hier genau ein Tupel und wird deshalb *Tupelvariable* genannt. $t$ ist nur Element des Ergebnisses, wenn die Aussage $A(t)$ für $t$ wahr ist. $A(t)$ wird deshalb auch *Bedingung* genannt. Die oben angegebene Menge ist also die Menge aller Tupel, für die $A(t)$ eine wahre Aussage ist.

Das vereinfacht die Datenbankabfrage insofern, als dass man dem Computer nur noch sagen muss, *was* man haben möchte. Der Computer weiß selber, *wie* er das Ergebnis errechnen kann. Bei der relationalen Algebra war das noch nicht so.

### 4.2.1 Der Aufbau der Bedingung

$A(t)$ ist eine Aussage in der Prädikatenlogik erster Stufe [4, S. 141]. Sie kann *Bereichsaussagen*, *Vergleichsaussagen*, logische Operatoren und Quantoren enthalten [1, Kap. 4.2.2].

[2] $\theta$: der griechische (kleine) Buchstabe "Theta"

Join

| Schüler-ID | Kurs-ID | Kurs-ID | Bezeichnung | Raum |
|---|---|---|---|---|
| 1 | 2 | 2 | Kunst | 110 |
| 2 | 1 | 1 | PW | 112 |
| 1 | 3 | 3 | Physik | 101 |
| 3 | 1 | 1 | PW | 112 |
| 2 | 3 | 3 | Physik | 101 |
| 2 | 2 | 2 | Kunst | 110 |
| 1 | 1 | 1 | PW | 112 |
| 3 | 3 | 3 | Physik | 101 |

Natural Join

| Schüler-ID | Kurs-ID | Bezeichnung | Raum |
|---|---|---|---|
| 1 | 2 | Kunst | 110 |
| 2 | 1 | PW | 112 |
| 1 | 3 | Physik | 101 |
| 3 | 1 | PW | 112 |
| 2 | 3 | Physik | 101 |
| 2 | 2 | Kunst | 110 |
| 1 | 1 | PW | 112 |
| 3 | 3 | Physik | 101 |

Abbildung 4.10: Join und Natural Join

### Die logischen Operatoren

Die logischen Operatoren $\vee$ (*oder*) und $\wedge$ (*und*) verknüpfen jeweils zwei Aussagen und formen damit eine neue Aussage. Der Operator $\neg$ negiert die nachfolgende Aussage.

### Bereichsaussagen

Bereichsaussagen haben folgende Form:

$$t.R$$

$t$ ist eine Tupelvariable und $R$ ist der Bezeichner für eine Relation. Die Aussage ist genau dann wahr, wenn der Wert der Tupelvariablen ein Tupel von $R$ ist [1, Kap. 4.2.2]. $t.R$ ist also äquivalent zu $t \in R$.

Das folgende Kalkül führt also zu einer originalgetreuen Kopie der Relation $R$:

$$\{\, t \mid t.R \,\}$$

Zusammen mit den logischen Operatoren lassen sich Relationen (hier: $R$ und $S$) auch vereinigen ($\{\, t \mid t.R \vee t.S \,\}$) oder schneiden ($\{\, t \mid t.R \wedge t.S \,\}$).

Folgendes Kalkül liefert eine leere Relation:

$$\{\, t \mid \neg t.R \,\}$$

### Vergleichsaussagen

Vergleichsaussagen verknüpfen die Attributwerte von Tupelvariable mit den Vergleichsoperatoren $=$, $\neq$, $<$, $>$, $\leq$ oder $\geq$ [1, Kap. 4.2.2]. Den Attributwert des Attributes mit dem Namen `Geb_Jahr` erreicht man durch die Tupelvariable, in dem man den Attributnamen in Klammern hinter den Namen der Tupelvariable schreibt.

So findet man in der Relation `Person` alle, die nach 1980 geboren sind, folgendermaßen:

$$\{\, t \mid t.Person \wedge (t(Geb\_Jahr) > 1980) \,\}$$

### Der Quantor $\exists$

Der Quantor $\exists$ wird *Existenzquantor* genannt. Er hat die Form

$$\exists_{t.R}(t(Zahl) = 42)$$

In Worten bedeutet dieser Ausdruck: "Es existiert ein Tupel in der Relation $R$, dessen Attribut `Zahl` den Wert 42 hat."

Allgemein besagt der Existenzquantor also, dass eine bestimmte Menge mindestens ein Element hat, für das eine bestimmte Aussage wahr ist. Da die Aussage von dem Menge abhängt, muss sie natürlich immer mit angegeben werden. Das geschieht meist unter dem Quantor oder als Index direkt dahinter. Die Angabe hat oft die Form $x \in X$ und gibt damit auch gleich an, von welcher Variablen die Aussage abhängig ist. Die Aussage steht hinter dem Quantor auf gleicher Höhe.

Beispiel: "Es gibt mindestens ein Element der Menge $X = \{1; 2; 3; 4; 5; 6\}$, dass die Gleichung $5 + x = 10$ erfüllt."

$$\exists_{x \in X}(5 + x = 10)$$

Folgende Aussage ist nicht wahr:

$$\exists_{x \in X}(5 + x = 12)$$

### Der Quantor $\forall$

Der *Universalquantor* $\forall$ hat den gleichen Aufbau wie der Existenzquantor, nur seine Bedeutung ist unterschiedlich.

$$\forall_{x \in X}(A(x))$$

Dieser Ausdruck heißt: "Für alle Elemente aus $X$ ist der Ausdruck $A(x)$ wahr."

### Negierung der Quantoren

Bei Negation der Quantoren bestehen folgende Äquivalenzen:

$$\neg\Big(\exists_{x.X}(A(x))\Big) \Longleftrightarrow \forall_{x.X}(\neg A(x))$$

$$\neg\Big(\forall_{x.X}(A(x))\Big) \Longleftrightarrow \exists_{x.X}(\neg A(x))$$

## 4.3 Die Datenbankabfrage mit SQL

Wie bereits in Kapitel 3 erwähnt, besteht die Sprache SQL aus zwei Teilen: der DDL (*data definition language*, Datendefinitionssprache) und der DML (*data manipulation language*, Datenmanipulationssprache). Die DDL habe ich in Kapitel 3 vorgestellt.

Die DML, Datenmanipulationssprache, enthält Befehle zur Datenbankabfrage und zur Manipulation der gespeicherten Datenbestände.

Bezüglich der Datenbankabfrage ist SQL eine *abbildungsorientierte Sprache* (*mapping – oriented language*) [4, Kap. 5.4]. Solche Sprachen entstammen der Überlegung, dass Abfragesprachen auch für den nicht – Mathematiker verständlich und anwendbar sein sollten. Deshalb hat man versucht, mathematische Konstrukte wie die Quantoren zu vermeiden. Eine Abfrage hat immer die Form einer Abbildung, wie man sie aus der Mathematik kennt[3]. Ansonsten ähneln diese Sprachen sehr dem relationalen Kalkül.

### 4.3.1 Die `select` – Anweisung

Der für die Datenbankabfrage wichtigste SQL – Befehl ist der ***`select`** – Befehl*. Er hat grundsätzlich die folgende Form:

```
select <Attribute>
from <Relationen>
where <Auswahlkriterium>;
```

Wie man sieht, ist eine ***`select`** – Anweisung* dreigeteilt.

#### Die `select` – Klausel

Der erste Teil wählt die Attribute aus, die das Ergebnis enthalten soll. Diese Attribute müssen dem DBMS aus den Relationen in der `from` – Klausel bekannt sein. Dadurch steht der Wertebereich der Abbildung fest.

Die Attribute werden angegeben durch ihren Namen. Falls mehrere Attribute aufgelistet sind, werden sie durch Kommata getrennt.

[3]Eine Abbildung hat immer einen Definitionsbereich, einen Wertebereich und eine Zuordnungsvorschrift.

### Die `from` – Klausel

Im zweiten Teil kann man angeben, aus welchen Relationen die Tupel entnommen werden sollen. Damit ist dann auch gleichzeitig der Definitionsbereich der Abbildung festgelegt.

In der `from` – Klausel kann man mehrere Relationen angeben, die durch Kommata getrennt werden. Jede Nennung einer Relation erzeugt eine Tupelvariable, auf die in der `select`– oder `where` – Klausel zurückgegriffen werden kann. Es ist auch möglich, vom Relationennamen abweichende Namen für die Tupelvariable festzulegen. Dadurch kann man mehrere Tupelvariable für ein und die selbe Relation erzeugen. Das ist nötig, wenn man zwei Tupel einer Relation vergleichen will.

```
select K1.Kurs-ID, K2.Kurs-ID
from Kurs K1, Kurs K2
where K1.Raum = K2.Raum;
```

Diese SQL – Anweisung findet aus der Relation `Kurs` alle Kurspaare, die den gleichen Raum benutzen (siehe Abbildung 4.8 auf Seite 53). Dazu werden zwei Tupelvariable (`K1`, `K2`) definiert, die sich beide auf die Relation `Kurs` beziehen.

Wie man schon an dem Beispiel sieht, greift man unter SQL so auf die Attribute von Tupeln zu, wie man es von vielen Programmiersprachen (BASIC, Pascal, C, Java... (T–ASM)) beim Zugriff auf Felder von Strukturen gewohnt ist[4].

### Die `where` – Klausel

Der `where` – Teil ist meistens der wichtigste und komplizierteste Teil einer `select` – Anweisung, denn er enthält die Bedingungen für Tupel in der Ergebnismenge. Hier findet sich also die *Abbildungsvorschrift*. Sie ist vergleichbar mit der Bedingung des relationalen Kalküls.

Wie im relationalen Kalkül können hier die mathematischen Vergleichsoperatoren und die logischen Verknüpfungen (`and`, `or`, `not`) angewandt werden.

[4] Auf das Attribut mit dem Namen `Attribut` der Relation `Relation` greift man so zu: `Relation.Attribut` (vgl. `K1.Raum`)

SQL erlaubt hier auch Mengenangaben. Anders als bisher kommen hier nicht mehr geschweifte Klammern (`{}`) bei der Kennzeichnung der Mengenangabe zum Einsatz, sondern runde Klammern (`()`). Man kann die Mengen explizit angeben oder auch einfach eine weitere `select` – Anweisung in die Klammern schreiben. Dann stellt das Ergebnis der *eingeschachtelten SQL – Anweisung* die Menge dar. Das ermöglicht eine tiefe Verschachtelung von SQL – Anweisungen.

SQL hat auch viele Möglichkeiten, mit diesen Mengen zu operieren, zum Beispiel durch den `in` – Operator, der äquivalent zu dem mathematischen Operator $\in$ ist [4, S. 149].

Ein Beispiel für Mengenangaben in der `where` – Klausel:

```
select Kurs-ID
from Kurs
where Raum in ( 101, 102, 103 );
```

ist gleichbedeutend mit:

```
select Kurs-ID
from Kurs
where Raum=101 or Raum=102 or Raum=103;
```

Dies sit nur ein kleiner Einblick in die große Menge von Möglichkeiten, die SQL mit der `select` – Anweisung bietet.

### 4.3.2 Andere SQL – Anweisungen

Weitere wichtige SQL – Anweisungen, die der DML zuzuordnen sind, sind `update`, `insert` und `delete`. Ich werde auf diese Befehle nicht näher eingehen, sondern nur jeweils ein einfaches Beispiel angeben (Alle Beispiele sind angelehnt die Datenbank aus Abbildung 4.8 auf Seite 53).

#### Anweisung `update`

Der Kurs "Kunst" heißt ab sofort "Bildende Kunst".

```
update Kurs
set Bezeichnung = 'Bildende Kunst'
where Kurs-ID = 2;
```

**Anweisung** `insert`

Es gibt einen neuen Kurs mit dem Namen “Mathematik”.

```
insert into Kurs
values < 4, 'Mathematik', 333 >;
```

Mit den Zeichen < und > kann man in SQL Tupel kennzeichnen.

**Anweisung** `delete`

Schüler `1` wählt den Kunstkurs ab.

```
delete from Schüler
where Schüler-ID=1 and Kurs-ID=2;
```

# Kapitel 5

# Das objektorientierte Datenbankmodell

Dieses Kapitel erhebt keinesfalls den Anspruch, auch nur eine annähernd vollständige Beschreibung des objektorientierten Datenmodells zu geben. Es soll vielmehr einen kurzen Einblick in die Ursprünge und Grundgedanken hinter diesem Datenmodell liefern.

## 5.1 Ursachen für die Entwicklung des OO – Modells

### 5.1.1 Probleme mit dem alten Datenmodell

Einige Jahre nach der Entwicklung des relationalen Datenmodells Anfang der siebziger Jahre hat die Untersuchung von Datenmodellen einige Fortschritte gemacht und es hat sich u.a. gezeigt, dass das "flache" [2, S. 57] Relationenmodell für sehr komplizierte, hochstrukturierte Datenstrukturen nur bedingt geeignet ist. Das sieht man z.B. schon daran, dass das Objekt "Adresse" in einer Relation, die Personendaten speichert, in seine Einzelkomponenten wie z.B. Straße, Hausnummer und Postleitzahl aufgeteilt werden muss. Das Objekt verliert dabei seine *Objektidentität*.

### 5.1.2 Die Verbreitung der OO – Programmierung

Um die Mitte der achtziger Jahre wurde eine neue Art zu programmieren, die *Objektorientierte Programmierung*, sehr populär. OO – Programmierung nimmt Abschied von dem alten Konzept, dass Prozeduren passive Objekte bearbeiten. Stattdessen geht man dazu über, die Objekte aktiv werden zu lassen. Jedes Objekt wird jetzt selber mit einem Satz von *Methoden* ausgestattet, welche die internen Daten verwalten und sie vollständig einkapseln sollen. D.h., dass jede Kommunikation mit den Objekten, jedes Auslesen und Setzen der Eigenschaften, über Methodenaufrufe erfolgen soll.

Objekte mit gleicher interner Datenstruktur und gleichem Verhalten gehören zu einer *Klasse*. Klassen kann man also als Prototypen von Objekten auffassen.

Klassen können ihre Eigenschaften und ihre Methoden an neue "Kindklassen" vererben. Die Kindklassen (Subklassen) können neue, von der Elternklasse (Superklasse) unabhängige Eigenschaften und Methoden bilden, sie können aber auch geerbte Eigenschaften und Methoden verändern. Dieses Neudefinieren wird *Overriding* genannt.

Objektorientierte Programmierung zeichnet sich u.a. dadurch aus, dass bestehende Objekte leicht ersetzt oder durch neue Objekte ergänzt werden können. Diese Eigenschaft vermindert den Wartungsaufwand von Software.

## 5.2 Die Entwicklung des OO – Datenmodells

Die schnelle Ausbreitung der objektorientierten Programmierung regte die Experten dazu an, die Herangehensweise der OO – Programmierung auch auf Datenbanken zu übertragen. Man wollte ein Datenmodell entwickeln, bei dem die Objekte ähnlich wie in der OO – Programmierung auf einer Vererbungshierarchie basieren. Weiterhin sollte das Modell die Einkapselung von Objekten durch Methoden sowie das Overriding (Überladen) von Methoden unterstützen.

Das führte zu einem Datenmodell, das z.B. die bereits angesprochene Objektidentität vollkommen wahrt.

Eine ausführlichere Behandlung des OO – Datenmodells würde hier den Rahmen sprengen. Eine kurze und prägnante Darstellung objektorientierter Datenmodelle gibt [2] im Kapitel 3.

Als kurze Demonstration soll hier nur eine Datenstruktur definiert werden, die Personen beschreibt und die wohl selbsterklärend ist (entnommen aus [9, S. 53]):

```
TYPE TUPLE ( Name: TUPLE ( Vorname: STRING,
                           Nachname: STRING ),
             Adresse: TUPLE ( PLZ: INTEGER,
                              Ort: STRING,
                              Straße: STRING,
                              Hausnummer: STRING ),
             Hobbies: SET ( Hobby: STRING ),
             Geburtsdatum: DATE )
```

Anmerkung: Das Attribut `Hobbies` ist hier definiert als eine Menge (beliebig vieler) Strings.

# Kapitel 6

# Datenschutz

Datenbanken verwalten oft riesige Mengen an Informationen. Oft werden sogar die gesamten Informationen eines Unternehmens einer Datenbank anvertraut.

Es ist ein vorrangiges Ziel von Datenbanken, jedem Mitarbeiter die Daten, die er zur Arbeit benötigt, auf einfache und effiziente Art und Weise bereitzustellen. Allerdings sollen nicht alle Benutzer der Datenbank gleichermaßen Zugriff auf alle Daten haben.

Beispielsweise ist es nicht wünschenswert, wenn ein Kunde, der über das Internet den Produktkatalog einer Firma durchblättert, gleich einen tiefen Einblick in die Firmeninterna erhält. Das würde den Kunden einerseits mit einem riesigen Berg von Informationen überhäufen, die er eigentlich gar nicht haben wollte. Andererseits würde die Firma so Firmengeheimnisse preisgeben, die sie eigentlich geheim halten wollte. Man muss also bestimmte Teile der Datenbank vor bestimmten Benutzergruppen "verstecken".

Auch wenn der Kunde ausschließlich Zugang zum Produktkatalog hat, soll er nicht nach gutdünken z.B. den Preis eines Produktes ändern können.

## 6.1 Zugriffsrechte

Wie man an dem genannten Beispiel sieht, ist es sinnvoll, jedem Benutzer einen gewissen Satz an Zugriffsrechten auf bestimmte Objekte der Datenbank einzuräumen. Beim Zugriff auf ein Objekt sind u.a. folgende möglichen Rechte sinnvoll [3, S. 98]:

- Selektierung + Aktualisierung (Lesen + Schreiben)

- nur Selektierung (nur Lesen)
- kein Zugriff (das Objekt ist vor dem Benutzer versteckt)

Aus der Möglichkeit, Zugriffsrechte zu erteilen, ergibt sich noch ein weiteres Recht, das ein Benutzer an einem Objekt haben kann: das Recht, Zugriffsrechte zu erteilen. Dieses Recht kann, wie alle anderen Rechte auch, für jedes Objekt gesondert vergeben werden.

Rechte müssen nicht unbedingt an einen bestimmten Benutzer gebunden sein. Die meisten DBMS erlauben es, Rechte an die "Welt" (oder Allgemeinheit) zu vergeben. Auf solche Objekte darf dann grundsätzlich jeder Benutzer der Datenbank zugreifen.

Um die bei sehr vielen Objekten und sehr vielen Benutzern sehr schnell wachsende Komplexität der Rechtevergabe für eine Datenbank zu vereinfachen, gibt es die sogenannte *Autorisierungsmatrix*. Diese Matrix hat für jedes Objekt der Datenbank eine Spalte und für jeden Benutzer eine Zeile. An jedem Schnittpunkt zwischen Spalte und Zeile werden dann die Rechte eingetragen, die der jeweilige Benutzer an dem entsprechenden Objekt hat [3, S. 100].

## 6.2 Benutzersichten

Ein sehr elegantes Konzept zur Realisierung von Datenschutz ist das bereits von SQL unter dem Namen "Views" bekannte Konzept der Benutzersichten. Dieses Konzept erlaubt es, dem Benutzer nur einen Teil der Daten einer Datenbank zu präsentieren. So kann man z.B. gezielt bestimmte Attribute einer Relation vor dem Benutzer verbergen. Views gestatten sogar eine von Attributwerten abhängige Auswahl der Daten. Man kann z.B. ausgehend von der in Abbildung 6.1 dargestellten Relation eine View definieren, die einem Kunden nur die von ihm aufgegebenen Bestellungen zeigt.

## 6.3 Authentifizierung

Wenn man ein so detailliertes Rechtevergabesystem aufgebaut hat, muss man natürlich auch darauf achten, dass sich jeder Benutzer eindeutig bei dem DBMS authentifizieren kann. Das ganze Konzept würde zusammenbrechen, wenn sich jeder beliebige Benutzer als Datenbankadministrator einloggen könnte und damit automatisch dessen Rechte erhalten würde.

Bestellungen

| Auftrags-ID | Kunden-ID | Artikel | bezahlt |
|---|---|---|---|
| 1 | 2 | Kühlschrank | ja |
| 2 | 1 | Hammer | nein |
| 3 | 3 | Telefonbuch | ja |
| 4 | 1 | Nagel | nein |

Abbildung 6.1: Beispielrelation für Datenschutz durch Views

Der klassische Weg, das zu erreichen, geht über *Passwörter*. Der Benutzer vereinbart mit dem DBMS ein geheimes Wort, welches er immer, wenn er mit der Datenbank arbeiten möchte, eingibt. Daran, dass der Benutzer das Passwort kennt, erkennt das DBMS den Benutzer.

Passwörter haben den Vorteil, dass sie übertragbar sind. So kann sich auch ein Benutzer, der zuvor noch nie mit der Datenbank gearbeitet hat, erfolgreich einloggen.

Es gibt jedoch einige Nachteile von Passwörtern. Sie werden zum Beispiel leicht vergessen. Um dem Vergessen entgegenzuwirken, benutzen viele Menschen Passwörter, die sie sich leicht merken können, wie zum Beispiel ihr Geburtsdatum, den Namen einer ihnen nahestehenden Person oder Trivialpasswörter wie "geheim", "passwort" oder "333". Das macht Passwörter leicht "knackbar". Es ist wirklich der Auslöser für sehr viele Angriffe auf Computersysteme, dass die Zugangspasswörter sehr leicht zu erraten sind. Ein weiterer Nachteil ist, dass viele Menschen es als unbequem empfinden, wenn sie jedes Mal ihr Passwort eingeben müssen, um mit einem Computer zu arbeiten.

Um diesen Nachteilen entgegenzuwirken, hat man neue Techniken zur Identifizierung von Benutzern entwickelt. Die wohl am weitesten verbreitete Alternative zu Passwörtern sind Chipkarten, von denen jeder Benutzer ein Exemplar hat. Dadurch, dass ein Anwender seine Chipkarte in das entsprechende Lesegerät steckt, kann er sich bei dem Computer anmelden und identifizieren.

Chipkarten haben jedoch auch Nachteile, die z.T. aus der menschlichen Bequemlichkeit und Vergesslichkeit erwachsen. So kann es passieren, dass ein Benutzer seine Chipkarte am Arbeitsplatz liegen lässt oder sonstwo verlegt. Mit der verlorenen Chipkarte kann sich nun jeder andere Mensch Zugang zu dem Computersystem verschaffen. Außerdem ist es umständlich, jedes Mal die Chipkarte aus der Tasche zu "kramen", wenn man mit dem Computer arbeiten möchte.

Eine Vermeidung dieser Nachteile bringt nur eine Technik, die den Benutzer ohne sein Zutun erkennen kann. Der Computer sollte seinen Benutzer also anhand seiner natürlichen Merkmale erkennen können[1]. Der Wissenschaftszweig, der sich mit der Erkennung menschlicher Merkmale durch den Computer beschäftigt, ist die *Biometrik*. Ziel ist es, Geräte zu entwickeln, die Menschen zuverlässig anhand von Gesicht, Retina, Stimme oder Fingerabdruck unterscheiden können. Die bisher erzielten Ergebnisse sind jedoch zum großen Teil noch ein gutes Stück von der Praxistauglichkeit entfernt, was Fehlerkennungsquote, Zuverlässigkeit und Preis betrifft. Gesichtserkennungssysteme lassen sich durch Fotos täuschen und Fingerabdruckscanner durch Wachsabdrücke. Alle mir bekannten professionellen Lösungen kommen nicht ohne zusätzlichen Schutz durch Passwörter oder Chipkarten aus. Mit neuen Forschungsergebnissen und wachsender Rechenleistung von Computern kann sich das aber durchaus innerhalb der nächsten Jahre ändern.

[1]... wenn sich der Benutzer keinen Chip implantieren lassen möchte.

# Kapitel 7

# Datenbanksicherheit und –integrität

Wie ich bereits in Kapitel 6 beschrieben habe, können die Informationen, die in einer Datenbank gespeichert sind, u.U. von immenser Bedeutung für ein Unternehmen sein. Da die Daten aber auf EDV – Anlagen gespeichert sind, unterliegen sie prinzipiell der Gefahr der Zerstörung. Diese Zerstörung kann mutwillig (z.B. durch einen frustrierten Mitarbeiter) oder zufällig entstehen. Letzteres muss nicht immer durch eine Naturkatastrophe herbeigeführt werden. Es reicht, wenn jemand eine Tasse Kaffee unglücklich über dem Unternehmensserver entleert, vor Schreck wegrennt und dabei über die Stromversorgungsleitung stolpert... Oftmals sind es aber einfach nur Fehler in der Software oder Hardwareausfälle, die zu einem Datenverlust führen.

Solche Horrorszenarien wie die eben beschriebenen gefährden die *Datensicherheit*. Die Datensicherheit gibt also an, wie sicher die Daten vor der Zerstörung sind [3, S. 106].

Unter *Integrität* versteht man die Richtigkeit und Gültigkeit der Daten. Eine Datenbank verliert ihre Integrität, wenn die gespeicherten Daten nicht mehr korrekt sind, wenn sie nicht mehr die realen Verhältnisse abbilden.

Beide Begriffe sind also miteinander verwandt. Man hat auch Techniken und Strategien entwickelt, die helfen sollen, die Datensicherheit und die Datenintegrität gleichermaßen aufrechtzuerhalten.

## 7.1 Backup, Restore

Das *Backup* besteht daraus, dass man die wichtigen Daten regelmäßig an einen anderen Ort kopiert. Ziel ist es, im Falle der Zerstörung der Daten am Ursprungsort immer eine Kopie zur Hand zu haben. Man kann dann mit den gesicherten Daten ein sogenanntes *Restore* machen. Dabei kopiert man die gesicherten Daten zurück an den Ursprungsort und hat so wieder den Zustand hergestellt, der zum Zeitpunkt des Backups herrschte.

Speichermedien für Backups sind üblicherweise Magnetbänder, da sie sich durch eine hohe Zuverlässigkeit, hohe Speicherkapazität und einen geringen Preis auszeichnen. Der Nachteil von Magnetbändern, die hohe Zugriffszeit beim wahlfreien Lesen, spielt bei Backups keine Rolle.

Alle großen Firmen führen regelmäßig Backups in großem Umfang durch. Als Veranschaulichung soll der in Abbildung 7.1 dargestellte Bericht eines Systemadministrators in einem Unternehmen dienen. Er erklärt dort, welche Backupstrategie in seinem Linux – Netzwerk zum Einsatz kommt. Man kann erkennen, dass das Unternehmen immer mehrere Kopien des gesamten (veränderlichen) Datenbestandes führt und dass es jederzeit möglich ist, nahezu jeden beliebigen Systemzustand des letzten Monats wiederherzustellen.

## 7.2 Logging

Während eines Backups darf die Datenbank nicht verändert werden, da das Backup ansonsten inkonsistent wird, d.h. es enthält nicht mehr *einen* definierten Zustand der Datenbank, sondern eine Mischung aus *mehreren* Zuständen. Das ist natürlich nicht wünschenswert.

Große Datenbanken müssen aber in der Regel rund um die Uhr für Anfragen und Veränderungen zur Verfügung stehen. Deshalb kann man es sich meistens nicht leisten, mehr als einmal am Tag ein Backup durchzuführen (und das meistens tief in der Nacht, wenn der Ansturm auf die Datenbank am geringsten ist). Um trotzdem nach einem Systemabsturz dem letzten Zustand, in dem die Datenbank noch ihre Integrität besaß, möglichst nahe zu kommen, unterstützen viele DBMS das *Logging*. Dabei werden in einer speziellen Datei, dem *Logfile*, der Termin des letzten vergangenen Backups und Informationen über alle nachfolgenden Änderungen der Datenbank festgehalten. Wenn jetzt nach einer Störung ein Restore des letzten Backups gemacht wurde, kann das DBMS anhand der Informationen aus dem Logfile

```
Message-ID: <8f22c4$ius$1@news.inf.uc3m.es>
From: "Peter T. Breuer" <ptb@oboe.it.uc3m.es>
Newsgroups: comp.os.linux.misc
Subject: Re: Why partition a Disk?
Date: 6 May 2000 21:21:08 GMT

Rick Hoffman <hoffmyster@netzero.net> wrote:
> If you are so inclined can you tell me if you backup your Linux
> files and if so how do you do it?

Err, well, yes, several gigabytes a day. You probably don't want to
know this.

Every day incremental changes since  the last backup are gathered by a
central backup client served from the individual machines involved. Every
week a "full" backup is done to a different backup client. 6 weeks of
daily incrementals are kept. One month of weekly full backups. Every
6 weeks cdroms should be cut of the full backups, but it's a pain.

The full backups cover only the user home areas and the /boot, /var, /etc
directories. The rest of the systems are supposed never to vary, and
are md5summed every day. Any change in them is detected and repaired with
reference to a central site (and the central site is duplicated in all
the remotes anyway).

[...]

The remote backups are done using find, tar, and ssh (plus gzip/bzip2).
It's a client/server script system.  The tar stream is checksummed on
the way out and on the way in, and the sums must match, or the backup is
repeated.

If the receiving area overflows, an old backup will be discarded,
weighting by age and size to find the removal candidate.

[...]

Another technique is to run a high-latency mirror. Say updated every
day or every week with rsync. The mirror area can be a compressed
file system .. it's not bad compression with e2compr.

Peter
```

Abbildung 7.1: Beispiel für Backup – Strategien von Firmen (gekürzt) die Datenbank "nachfahren". D.h. es führt alle im Logfile aufgezeichneten Veränderungen erneut aus. So lässt sich die Datenbank wieder in einen Zustand versetzen, der einem Zeitpunkt kurz vor der Zerstörung der Daten entspricht.

*

Selbst bei Anwendung des Loggings würde es bei einer Integritätsverletzung der Datenbank ziemlich lange dauern, bis die Datenbank wieder zur Verfügung steht, da erst ein langwieriges Restore durchgeführt werden muss und danach das Logfile abzuarbeiten ist.

Daher sind Methoden wünschenswert, die die Integrität einer Datenbank schon während der Laufzeit prüfen, bzw. verhindern, dass sie überhaupt erst ihre Integrität verlieren kann.

Dazu ist es nötig, schon während des Entwurfs einer Datenbank folgende Dinge zu tun [5]:

- Aufstellen von *Integritätsbedingungen*
- Festlegen von *Transaktionen*

Auf Beides werde ich in den folgenden Unterkapiteln eingehen.

## 7.3 Integritätsbedingungen

Es gibt bestimmte Aussagen, die man schon während der Entwurfsphase über die Daten in einer Datenbank treffen kann und anhand deren Wahrheitsgehalt man die Integrität einer Datenbank bestimmen kann. Diese Aussagen nennt man *Integritätsbedingungen*. Man unterscheidet drei verschiedene Arten von Integritätsbedingungen [5, S. 5]:

- **Statische Integritätsbedingungen.** Statische Integritätsbedingungen (IB) betreffen den *Zustand* einer Datenbank. Die Gesamtheit aller statischen IB definiert die Menge der Zustände, in denen sich eine Datenbank befinden darf.

  Jeder Wertebereich eines Attributs ist eine statische Integritätsbedingung (z.B. darf eine Monatszahl nur zwischen 1 und 12 liegen). Es gibt aber auch komplexere statische Integritätsbedingungen, wie zum Beispiel: "Jeder Schüler belegt mindestens Mathematik, Deutsch und eine Fremdsprache."

- **Transitionale Integritätsbedingungen.** Statische IB schränken die Menge der *möglichen* Zustände auf die Menge der *erlaubten* Zustände ein. Analog geschieht das bei den transitionalen IB mit den *Zustandsübergängen.*

  Eine transitionale IB ist zum Beispiel: "Bei der Änderung der Klassenstufe eines Schülers darf diese nur um eins größer werden."

- **Dynamische Integritätsbedingungen.** Dynamische IB schränken die Menge der möglichen *Zustandsfolgen* ein. Der Unterschied zu den transitionalen IB ist, dass sich dynamische IB auf mehr als zwei Zustände beziehen können.

  Eine dynamische IB wäre: "Ein Schüler besucht zuerst die Grundschule. Dann folgt entweder das Gymnasium oder die Gesamtschule. Erst danach kann er an einer Universität studieren.[1]"

  Wie man sieht, können sprachlich ausformulierte dynamische IB schnell sehr lang und komplex werden. Um sie trotzdem formulieren zu können, gibt es *temporale Formeln* und *Transitionsgraphen*, auf die ich hier aber nicht näher eingehen werde. In [5, Kapitel 2+3] werden beide Darstellungsformen sehr detailliert beschrieben.

## 7.4 Transaktionen

Eine *Transaktion* ist eine Anzahl von Operationen an der Datenbank, die als atomar betrachtet werden. Es gilt: Entweder werden alle Operationen einer Transaktion ausgeführt oder keine. In allen anderen Fällen verliert die Datenbank ihre Integrität. Eine Transaktion könnte zum Beispiel das Umbuchen eines Geldbetrages von einem Konto auf ein anderes sein. Diese Transaktion besteht aus mindestens zwei Operationen: Das Subtrahieren des Betrages von einem Konto und das Addieren des gleichen Betrages auf ein anderes Konto. Einzeln ausgeführt ergeben die Operationen keinen Sinn.

Die Transaktionen überführen die Datenbank grundsätzlich von einem integren Zustand in einen anderen Zustand der Integrität.

Das DBMS unterstützt spezielle Kommandos, die es erlauben, eine Transaktion zu beginnen und zu beenden oder abzubrechen. Bei einem Abbruch

[1] Ich bin mir durchaus bewußt, dass nach der Grundschule auch z.B. die Hauptschule folgen kann. Das Beispiel ist sehr vereinfacht und deshalb nicht ganz realitätsgetreu.

macht das DBMS automatisch alle zu der Transaktion gehörenden Aktionen rückgängig.

## 7.5 Integritätsüberwachung

Es gibt drei verschiedene Möglichkeiten, anhand von Integritätsregeln und / oder Transaktionen die Integrität einer Datenbank zu überwachen [5, Kap. 1.2].

### 7.5.1 Erste Möglichkeit

Die erste Möglichkeit besteht darin, in dem DBMS einen universellen Monitor einzusetzen, der die Einhaltung sämtlicher Integritätsregeln ständig überprüft und bei deren Verletzung sofort eingreift.

Alle Anwendungsprogramme oder Benutzer können jetzt mit beliebigen Transaktionen auf die Datenbank zugreifen. Eine Integritätsverletzung kann nicht auftreten, weil das der Monitor verhindern würde.

Diese Möglichkeit der Integritätsüberwachung ist zur Zeit noch nicht praktikabel, weil der Monitor aufgrund seiner Universalität nicht effizient genug arbeiten kann. Die Rechenleistung heutiger Computer scheint noch nicht auszureichen, um einen solchen Monitor zu realisieren.

### 7.5.2 Zweite Möglichkeit

Das entgegengesetzte Extrem zum universellen Monitor, das aber trotzdem nicht auf eine zentrale Überwachung der Integrität verzichtet, ist die Einrichtung eines Wrappers um das DBMS herum, der Anwendungsprogrammen nur die Möglichkeit anbietet, bestimmte, fest definierte Transaktionen auszuführen. Anwendungsprogramme können also nicht mehr direkt die Funktionen des DBMS nutzen, sondern sie müssen auf die ausgewählten Transaktionen zurückgreifen. Diese Transaktionen wurden sorgfältig auf ihre "Integritätstreue" hin geprüft. Da die Anwendungen nicht anderes mit der Datenbank machen können, als die vorgefertigten Transaktionen auszuführen, kann auch bei dieser Möglichkeit keine Integritätsverletzung entstehen.

Die Integritätsprüfung durch die Transaktionen kann wesentlich effizienter erfolgen als durch einen universellen Monitor.

Ein Nachteil dieser Möglichkeit ist, dass sie einen sehr hohen Aufwand beim Entwurf und bei der Implementierung der Datenbank erfordert, da ja eine ganz neue Schnittstellenschicht geschaffen werden muss.

### 7.5.3 Dritte Möglichkeit

Bei der dritten Möglichkeit verzichtet man ganz auf eine zentrale Integritätsüberwachung. Die Sicherung der Integrität ist Aufgabe der Anwendungsprogramme. Diese müssen z.B. durch Plausibilitätsprüfungen von Benutzereingaben sicherstellen, dass keine falschen Informationen an das DBMS gelangen.

Diese Möglichkeit wird — wahrscheinlich aus Effizienzgründen — am häufigsten angewendet. Sie stellt aber auch den geringsten Schutz gegenüber Integritätsverletzungen dar.

*

In der Praxis werden auch oft Kombinationen zwischen den drei genannten Möglichkeiten benutzt.

# Literaturverzeichnis


[1] *Ernst Grill*: Relationale Datenbanken: vom logischen Konzept zur physischen Realisierung; Ziele – Methoden – Fallstudie; Hallbergmoos: AIT, Angewandte Informations – Technik – Verl. – GmbH, 1990; ISBN 3-926571-05-5

[2] *Gottfried Vossen*: Datenbank – Theorie; Bonn: Internat. Thomson Publ., 1995; Reihe: TAT, Thomson's Aktuelle Tutorien; ISBN 3-8266-0126-2

[3] *Wolfgang R. Diemer*: Relationale Datenbanken kurz und bündig: Sicherheit, Integrität und Unabhängigkeit in der Datenverwaltung; Würzburg: Vogel, 1989; (Reihe: CHIP Wissen) ISBN 3-8023-0797-6

[4] *G. Schlageter / W. Stucky*: Datenbanksysteme: Konzepte und Modelle; Stuttgart: Teubner, 1983 (Leitfäden der angewandten Mathematik; Bd. 37) (Teubner – Studienbücher: Informatik) ISBN 3-519-12339-8

[5] *Udo Lipeck*: Dynamische Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung; Berlin, Heidelberg, New York, London, Paris, Tokyo: Springer, 1989 (Informatik – Fachberichte Bd. 209) ISBN 3-540-51130-X

[6] *Alfred Moos / Gerhard Daues*: Datenbank – Engineering: Analyse, Entwurf und Implementierung relationaler Datenbanken mit SQL; Braunschweig, Wiesbaden: Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1997 (vieweg Ausbildung und Studium) ISBN 3-528-15183-8

[7] *Newsgoups*[2]: Die genauen Quellen sind bei den Zitaten vermerkt.


[2]**Anmerkung zu Newsgroups:** Ich bin mir der eingeschränkten Glaubwürdigkeit von vielen Newsgroup – Postings bewusst. Deshalb liegt das Schwergewicht meiner Quellen

[8] `Message-ID: 38C4C9E5.89EF3074@mem.unibe.ch` von 7.3.2000

[9] *Andreas Heuer*: Objektorientierte Datenbanken: Konzepte, Modelle, Systeme; Bonn, München, Paris: Addison – Wesley, 1992 ISBN 3-89319-315-4

[10] *Herman Sauer*: Relationale Datenbanken: Theorie und Praxis; Bonn; Reading, Massachusetts [u.a.]: Addison – Wesley – Longman, 1998 ISBN 3-8273-1381-3

[11] *Oracle7 Server SQL Language Reference Manual*: History of SQL; (http://info-it.umsystem.edu/oracle/svslr/svslr.2.0013.html)

[12] `http://www.databases.about.com`; Artikel: *Database Normalisation – Definitions*

auch bei den "konventionellen" Büchern. Newsgroups bieten aber die einzigartige Möglichkeit zu erfahren, wie die Datenbanktheorie in die Praxis umgesetzt wird und auf welche Probleme man dabei stößt. Deshalb wäre es meiner Meinung nach ein Verlust, die vielen Beiträge aus den Newsgroups zu ignorieren.

# Abbildungsverzeichnis