

Die Datenmodellierung ist ein wichtiger Schritt bei der Entwicklung eines Informatiksystems. Es geht darum, ein DB-Schema zu finden, so dass

1. alle benötigten Daten im DB-Schema abgespeichert werden können,
2. effizient auf die Daten zugegriffen werden kann und
3. die Datenkonsistenz gewährleistet ist.

In diesem Kapitel studieren wir die grundlegenden Möglichkeiten, wie Entitäten<sup>1</sup> verschiedener Typen (z. B. Autos und ihre Fahrer) zueinander in Beziehung stehen können. Wir untersuchen dann, wie die Tabellenstruktur eines DB-Schemas diese Beziehungsarten abbilden kann. Dabei werden wir wesentlichen Gebrauch machen von den Constraints, welche wir im vorherigen Kapitel eingeführt haben. Auf das Thema der Effizienz wird später im Kap. 7 noch genauer eingegangen. Die Datenkonsistenz wird in den Kap. 9 und 10 ausführlich behandelt.

Ein DB-Schema ist üblicherweise aus sehr vielen Tabellen aufgebaut. In einer rein textuellen Beschreibung eines solchen DB-Schemas sind dann die Beziehungen zwischen den Tabellen nicht mehr klar darstellbar. Deshalb führen wir eine Diagramm-Notation für DB-Schemata ein, mit der wir die Tabellen-Struktur einer relationalen Datenbank graphisch darstellen können. Unsere Notation lehnt sich an die sogenannte Krähenfuss-Notation an.

---

<sup>1</sup>Hier könnten wir statt *Entitäten* auch *Dinge* sagen (falls wir Personen als Dinge betrachten). Häufig wird der Begriff *Entität* als Sammelbegriff verwendet um bspw. Dinge, Eigenschaften und Relationen auf einmal anzusprechen.

3.1 Diagramme für m:1-Beziehungen

In unseren Diagrammen wird jedes Schema (jede Tabelle) durch eine Box dargestellt. Die Kopfzeile der Box besteht aus dem Namen des Schemas. Danach werden die einzelnen Attribute aufgelistet, wobei die Attribute des Primärschlüssels unterstrichen sind. Falls Tabellen einer Datenbank dargestellt werden, so verwenden wir in der Kopfzeile den Namen der jeweiligen Tabelle, meinen damit aber eigentlich das Schema der entsprechenden Relation.

Abb. 3.1 zeigt eine Tabelle mit vier Attributen, wobei die ersten beiden den Primärschlüssel bilden.

Manchmal werden wir die Liste der Attribute weglassen, wenn sie nicht relevant ist. Die entsprechende Darstellung der Tabelle wird in Abb. 3.2 gezeigt.

Abb. 3.3 zeigt die beiden Tabellen aus Beispiel 2.5.

Es fehlt jetzt noch die Darstellung der Verbindung zwischen diesen beiden Tabellen. Die Daten zu Autos und Personen sind ja nicht isoliert, sondern es wird auch abgespeichert, wer der Fahrer eines Autos ist. Dies wird erreicht durch das Attribut FahrerId der Tabelle Autos, welches ein Fremdschlüssel auf die Tabelle Personen ist.

Diese Fremdschlüssel-Beziehung geben wir durch eine Verbindungslinie zwischen den beiden Boxen an, siehe Abb. 3.4.

Eine solche Beziehung heisst *m:1-Beziehung*. Die Zahl 1 besagt dabei, dass es zu jedem Auto *höchstens einen* Fahrer gibt. Auf der anderen Seite sagt der Buchstabe m, dass eine

Abb. 3.1 Darstellung einer einzelnen Tabelle

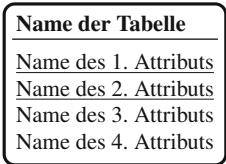


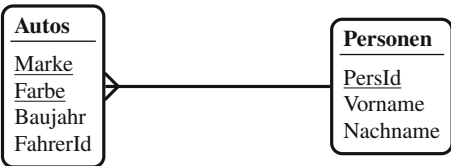
Abb. 3.2 Tabelle ohne Liste der Attribute




Abb. 3.3 Tabellen aus Beispiel 2.5



Abb. 3.4 Fremdschlüssel-Beziehung




Person *mehrere* Autos fahren kann. Wir verwenden die Gabelung (den Krähenfuss)  der Verbindungslinie um auszudrücken, dass es mehrere Autos zu einem Fahrer geben kann. Auf der Seite der Personentabelle gibt es keine Gabelung, da ein Auto eben nicht von mehreren Personen gefahren werden kann.

Mit Hilfe eines *not null Constraints* auf dem Fremdschlüsselattribut `FahrerId` können wir verlangen, dass es zu jedem Auto *mindestens* einen Fahrer geben muss. Eine solche Bedingung nennen wir auch *Existenzbedingung*, da sie verlangt, dass gewisse Tupel in der `Personen`-Tabelle existieren müssen. Im Diagramm geben wir eine solche Existenzbedingungen für Personen mit einem senkrechten Strich durch die Verbindungslinie auf der Seite der `Personen`-Tabelle an. Das entsprechende Diagramm ist in Abb. 3.5 dargestellt.

Diese Darstellung drückt zwei Sachverhalte aus:

1. Jedes Auto hat genau einen Fahrer, das heisst mindestens einen und auch höchstens einen Fahrer.
2. Jede Person kann kein, ein oder mehrere Autos fahren.

In unserer Diagramm-Notation können wir die *mindestens ein* Bedingung auch auf der Seite mit mehreren Beziehungen hinzufügen, indem wir den senkrechten Strich mit dem Krähenfuss kombinieren. Wir erhalten so eine Verbindungslinie der Form . Diese Existenzbedingung können wir im relationalen Modell *nicht* durch einen *not null* Constraint auf einem Attribut ausdrücken. Trotzdem ist es wichtig, solche Bedingungen in den Diagrammen anzugeben, da sie die Semantik der modellierten Konzepte verdeutlichen. Nehmen wir also an, wir haben folgende Bedingungen:

1. Jedes Auto hat keinen oder einen Fahrer.
2. Jede Person fährt ein oder mehrere Autos.

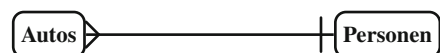
Wir erhalten dazu das Diagramm in Abb. 3.6.

Natürlich kann die *mindestens ein* Bedingung auf beiden Seiten einer m:1-Beziehung verlangt werden. Wir haben dann:

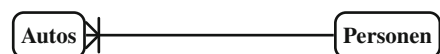
1. Jedes Auto hat genau einen Fahrer.
2. Jede Person kann ein oder mehrere Autos fahren.

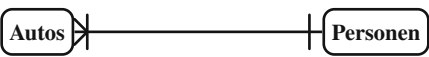
Dies ist in Abb. 3.7 dargestellt.

**Abb. 3.5** Jedes Auto hat genau einen Fahrer



**Abb. 3.6** Jede Person fährt mindestens ein Auto





**Abb. 3.7** Jede Person fährt mindestens ein Auto und jedes Auto hat genau einen Fahrer

**Tab. 3.1** Krähenfuss-Notation

Beschreibung	Symbol
Keine oder eine Beziehung	—
Keine, eine oder mehrere Beziehungen	—<
Genau eine Beziehung	—
Eine oder mehrere Beziehungen	— *

Zusammenfassend gibt es vier Möglichkeiten, wie ein Ende einer Verbindungslinie aussehen kann. Wir geben diese Kombinationen von *höchstens ein* und *mindestens ein* mit ihrer graphischen Darstellung nochmals in Tab. 3.1 an.

### 3.2 Diagramme für m:n-Beziehungen

Im obigen Abschnitt haben wir gesehen, wie wir eine m:1-Beziehung zwischen zwei Konzepten modellieren können. Der allgemeine Fall einer Beziehung zwischen zwei Konzepten ist jedoch nicht die m:1-Beziehung sondern die sogenannte m:n-Beziehung. Betrachten wir ein DB-Schema für eine Bank, welche Kunden und ihre Konten verwalten muss. Dabei soll folgendes gelten:

1. Ein Kunde kann mehrere Konten haben.
2. Ein Konto kann mehreren Kunden gemeinsam gehören.

Es handelt sich also nicht um eine m:1-Beziehung, sondern um eine Beziehung bei der in beide Richtungen jeweils ein Tupel mit mehreren anderen in Relation stehen kann. Wir bezeichnen eine solche Beziehung als *m:n-Beziehung*. Um solche Beziehungen im relationalen Modell zu beschreiben, benötigen wir nicht nur eine Tabelle für die Kunden und eine für die Konten sondern auch noch eine zusätzliche Tabelle um die Beziehung zwischen Kunden und Konten zu modellieren. Wir betrachten folgendes Beispiel.

*Beispiel 3.1.* Die Bank verwaltet Kunden und ihre Konten. Kunden haben eine eindeutige Kundennummer und einen Namen. Zu einem Konto gehört die Kontonummer und der Kontostand. Dies ergibt folgendes DB-Schema für die Verwaltung der Kunden- und Kontendaten

$$\mathcal{S}_{\text{Kunden}} := (\underline{\text{KundenNr}}, \text{Name})$$

$$\mathcal{S}_{\text{Konten}} := (\underline{\text{KontoNr}}, \text{Stand})$$

Die entsprechenden Tabellen könnten beispielsweise die folgenden Daten enthalten:

<b>Kunden</b>		<b>Konten</b>	
<b>KundenNr</b>	<b>Name</b>	<b>KontoNr</b>	<b>Stand</b>
A	Ann	1	1000
B	Tom	2	5000
C	Eva	3	10
D	Bob		

Die Kunden und Konten sollen in einer m:n-Beziehung stehen. Wir nehmen an:

1. Konto Nr. 1 gehört Ann,
2. Konto Nr. 2 gehört Ann und Tom gemeinsam,
3. Konto Nr. 3 gehört Eva und Bob gemeinsam.

Um eine m:n-Beziehung zwischen Kunden und Konten zu modellieren, benötigen wir ein zusätzliches Schema, welches aus dem Primärschlüssel von  $\mathcal{S}_{\text{Kunden}}$  und dem Primärschlüssel von  $\mathcal{S}_{\text{Konten}}$  besteht. Wir setzen also:

$$\mathcal{S}_{\text{KuKo}} := (\underline{\text{KundenNr}}, \underline{\text{KontoNr}})$$

Wir können nun folgende Tabelle abspeichern, welche die obige Relation zwischen Kunden und Konten repräsentiert:

<b>KuKo</b>	
<b>KundenNr</b>	<b>KontoNr</b>
A	1
A	2
B	2
C	3
D	3

Wir wollen uns nun noch die Constraints in diesem DB-Schema überlegen. Es ist klar, dass der Primärschlüssel von  $\mathcal{S}_{\text{KuKo}}$  aus den Primärschlüsseln von  $\mathcal{S}_{\text{Kunden}}$  und  $\mathcal{S}_{\text{Konten}}$  zusammengesetzt sein muss. Andernfalls kann eine m:n-Beziehung nicht dargestellt werden.

Weiter stellen wir fest, dass zu jedem Eintrag in der KuKo-Tabelle die entsprechenden Kunden und Konten existieren müssen. Das heisst,

1. das Attribut `KundenNr` in `KuKo` ist ein Fremdschlüssel auf `Kunden`,
2. das Attribut `KontoNr` in `KuKo` ist ein Fremdschlüssel auf `Konten`.

Diese Fremdschlüssel müssen einen not null Constraint erfüllen, da sie Teil des Primärschlüssels des `KuKo` Schemas sind.

Abb. 3.8 zeigt die Diagramm-Notation der m:n-Beziehung zwischen Kunden und Konten.

Wir müssen uns nun noch überlegen, welche Existenzbedingungen für die `KuKo`-Tabelle gelten sollen. Das heisst, wir müssen folgende Fragen beantworten:

1. Hat jeder Kunde ein Konto?
2. Muss jedes Konto einem Kunden gehören?

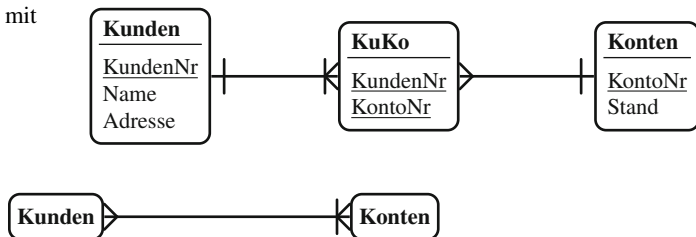
Wir beantworten die erste Frage mit *ja*. Wer kein Konto hat, kann auch kein Kunde sein. Die zweite Frage verneinen wir. Unsere Bank ist etwas altmodisch und lässt nachrichtenlose Vermögen (d. h. Konten ohne bekannte Kundenbeziehung) zu. Wir fügen im Diagramm also noch die Existenzbedingung für `KuKo` auf der Kunden-Seite ein. Das vollständige Diagramm des DB-Schemas für unsere Bank wird in Abb. 3.9 gezeigt.

In diesem Beispiel enthält die Tabelle `KuKo` keine weiteren Daten ausser den Primärschlüsseln für Kunden und Konten. Im Prinzip können wir somit die Box für dieses Schema bei der Diagrammdarstellung des DB-Schemas weglassen. Dies führt dann zu der abgekürzten Schreibweise in Abb. 3.10. Dabei ist jedoch zu beachten, dass die Existenzbedingungen, welche im vollständigen Schema (siehe Abb. 3.2) bei der `KuKo`-Box angegeben waren, nun bei den Boxen für Kunden, beziehungsweise Konten, eingezeichnet sind.

**Abb. 3.8** m:n-Beziehung



**Abb. 3.9** m:n-Beziehung mit Existenzbedingungen



**Abb. 3.10** Abgekürzte Darstellung einer m:n-Beziehung

Bisher haben wir nur binäre Beziehungen betrachtet, das heisst Beziehungen zwischen *zwei* Konzepten. Für den allgemeinen Fall müssen wir auch Beziehungen betrachten, welche zwischen drei und mehr Konzepten bestehen. Dazu studieren wir folgendes Beispiel.

*Beispiel 3.2.* Wir betrachten ein DB-Schema für eine Universität. Dieses soll Studierende, Professoren und Vorlesungen verwalten können. Wir treffen folgende Annahmen. Studierende haben eine Matrikelnummer und einen Namen. Professoren haben eine Personalnummer und einen Namen. Vorlesungen sind durch ihre Bezeichnung und ihr Semester identifiziert. Zusätzlich wird zu jeder Vorlesung der entsprechende Hörraum angegeben. Dies ergibt folgendes DB-Schema:

$$\mathcal{S}_{\text{Studierende}} := (\text{MatNr}, \text{Name})$$

$$\mathcal{S}_{\text{Professoren}} := (\text{PersNr}, \text{Name})$$

$$\mathcal{S}_{\text{Vorlesungen}} := (\text{Bezeichnung}, \text{Semester}, \text{Raum})$$

Wir können nun Prüfungen als ternäre (3-stellige) Beziehung zwischen diesen Konzepten modellieren. Ein Professor prüft seine Studierenden über eine Vorlesung. In diesem Fall wird die Beziehung auch noch zusätzliche Attribute haben (neben den Primärschlüsseln der in Beziehung stehenden Konzepte), nämlich das Datum und das Resultat (die Note) der Prüfung. Das ergibt folgendes Schema:

$$\mathcal{S}_{\text{Prüfungen}} := (\text{MatNr}, \text{PersNr}, \text{Bezeichnung}, \text{Semester}, \text{Datum}, \text{Note}).$$

Dieses Beispiel zeigt unter anderem:

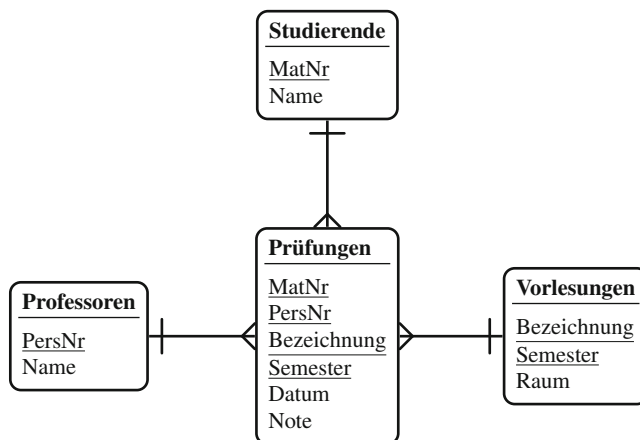
1. Beziehungen können zwischen mehr als zwei Konzepten bestehen.
2. Wenn ein Primärschlüssel aus mehreren Attributen besteht, so muss der ganze Primärschlüssel in der Beziehungstabelle vorkommen. Im Beispiel enthält  $\mathcal{S}_{\text{Prüfungen}}$  die Attribute *Bezeichnung* und *Semester*, um eine Vorlesung zu identifizieren.
3. Beziehungen können zusätzliche Attribute haben.

Das DB-Schema für die Universität ist in Abb. 3.11 graphisch dargestellt.

---

### 3.3 Diagramme für 1:1-Beziehungen

In den vorangehenden Abschnitten haben wir m:1- und m:n-Beziehungen betrachtet. Natürlich gibt es daneben auch noch 1:1-Beziehungen, welche wir in diesem Abschnitt studieren werden.



**Abb. 3.11** Ternäre Beziehung

Als Beispiel für eine 1:1-Beziehung betrachten wir die Mitglieder der Schweizer Regierung (diese heissen Bundesrat bzw. Bundesrätin) sowie deren Departemente<sup>2</sup>. Jedes Regierungsmitglied steht *genau einem* Departement vor und und das wiederum wird von *genau* diesem Bundesrat oder dieser Bundesrätin geführt. Diese Situation können wir *nicht* mit Hilfe von Fremdschlüssel-Attributen beschreiben. Betrachten wir folgendes DB-Schema:

$$\mathcal{S}_{\text{Regierung}} := (\text{BrId}, \text{DepId})$$

$$\mathcal{S}_{\text{Departemente}} := (\text{DepId}, \text{BrId}).$$

Dabei ist im Schema  $\mathcal{S}_{\text{Regierung}}$  das Attribut  $\text{DepId}$  ein Fremdschlüssel auf  $\mathcal{S}_{\text{Departemente}}$ , und im Schema  $\mathcal{S}_{\text{Departemente}}$  ist  $\text{BrId}$  in ein Fremdschlüssel auf  $\mathcal{S}_{\text{Regierung}}$ . Zu diesem DB-Schema gibt es die folgende Instanz:

Regierung		Departemente	
BrId	DepId	DepId	BrId
A	Y	Y	C
B	Y	Z	A
C	Z		

Aus zwei Gründen entspricht dies nicht einer 1:1-Beziehung.

<sup>2</sup>Ein Departement in der Schweiz ist vergleichbar mit einem Ministerium in anderen Ländern.



1. Die Bundesrätinnen A und B haben beide Y als Departement eingetragen.
2. Das Departement Z wird von der Bundesrätin A geführt, diese hat jedoch Y als Departement eingetragen.

Wir können eine 1:1-Beziehung korrekt modellieren, indem wir beiden Tabellen einen gemeinsamen Primärschlüssel geben. Ein korrektes Schema für eine 1:1-Beziehung sieht also wie folgt aus:

$$\begin{aligned}\mathcal{S}_{\text{Regierung}} &:= (\underline{\text{BrDepId}}, \text{Name}) \\ \mathcal{S}_{\text{Departemente}} &:= (\underline{\text{BrDepId}}, \text{Bezeichnung}).\end{aligned}$$

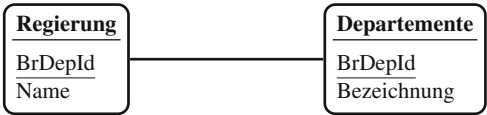
Abb. 3.12 zeigt die graphische Darstellung dieses DB-Schemas.  
Folgende Tabellen sind eine zulässige Instanz dieses DB-Schemas.

Regierung		Departemente	
BrDepId	Name	BrDepId	Bezeichnung
1	Berset	1	EDI
2	Maurer	3	EJPD

Berset steht also dem EDI<sup>3</sup> vor. Für Mauer gibt es jedoch keinen Eintrag in der Departemente-Tabelle, und auch wer das EJPD<sup>4</sup> führt, ist in dieser Instanz nicht ersichtlich. Um diese Situation zu verhindern, können wir im Diagramm noch Existenzbedingungen hinzufügen. Diese bedeuten dann, dass es zu jedem Eintrag in der Regierung-Tabelle ein entsprechendes Tupel in der Departemente-Tabelle geben muss und umgekehrt. Damit ist obige Instanz ausgeschlossen.

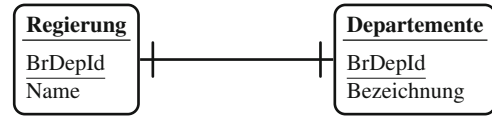
Die entsprechende Integritätsbedingung lautet, dass in beiden Tabellen der Primärschlüssel gleichzeitig ein Fremdschlüssel ist auf die Tabelle, zu der die 1:1-Beziehung besteht. Das heisst, dass der Primärschlüssel BrDepId im Schema Regierung gleichzeitig ein Fremdschlüssel auf  $\mathcal{S}_{\text{Departemente}}$  und der Primärschlüssel BrDepId in Akten gleichzeitig ein Fremdschlüssel auf  $\mathcal{S}_{\text{Angestellte}}$  ist. Abb. 3.13 zeigt das entsprechende Diagramm.

Abb. 3.12 1:1-Beziehung

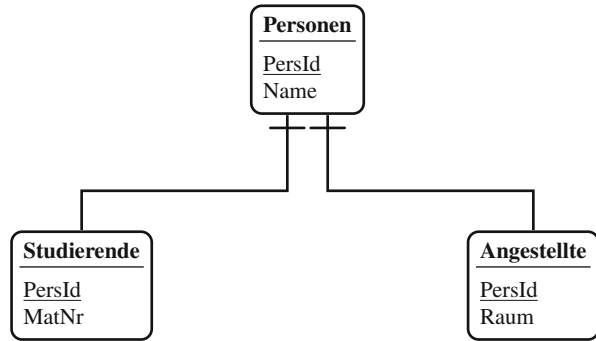


<sup>3</sup>Eidgenössisches Departement des Inneren.  
<sup>4</sup>Eidgenössisches Justiz- und Polizeidepartement.

**Abb. 3.13** 1:1-Beziehung mit Existenzbedingungen



**Abb. 3.14** Vererbung



Eine wichtige Anwendung von 1:1-Beziehungen ist die Modellierung von *Vererbung* (auch *Spezialisierung* genannt). Studieren wir ein DB-Schema für eine Universität. An der Uni gibt es Personen, welche wir durch folgendes Schema beschreiben:

$$\mathcal{S}_{\text{Personen}} := (\underline{\text{PersId}}, \text{Name}).$$

Nun wollen wir das Konzept *Personen* spezialisieren zu *Angestellte* und *Studierende*. Studierende sind Personen, die eine Matrikelnummer haben. Das heisst, sie erben alle Attribute von *Personen* und haben ein zusätzliches Attribut *MatNr*. Angestellte haben keine Matrikelnummer, dafür ist jedem Angestellten ein Büro zugeteilt, dessen Raumnummer gespeichert werden soll. Das heisst Angestellte erben ebenfalls die Attribute von *Personen* und besitzen ein zusätzliches Attribut *Raum*. Mit Hilfe von 1:1-Beziehungen können wir das durch folgende Schemata beschreiben:

$$\mathcal{S}_{\text{Studierende}} := (\underline{\text{PersId}}, \text{MatNr})$$

$$\mathcal{S}_{\text{Angestellte}} := (\underline{\text{PersId}}, \text{Raum}).$$

In beiden Schemata ist der Primärschlüssel *PersId* gleichzeitig Fremdschlüssel auf  $\mathcal{S}_{\text{Personen}}$ . Dies garantiert, dass jeder Studierende und jeder Angestellte auch tatsächlich eine Person ist. Abb. 3.14 zeigt die graphisch Darstellung der Vererbung.

Hier gibt es eine wichtige Asymmetrie. Jeder Studierende ist eine Person und auch jeder Angestellte ist eine Person. Jedoch muss nicht jede Person ein Studierender oder ein Angestellter sein. Die Existenzbedingungen gelten also (wie im Diagramm dargestellt) nur in eine Richtung der 1:1-Beziehungen. Im obigen Beispiel nennen wir deshalb die Relation

Personen *Basisrelation*. Die Relationen Studierende und Angestellte heissen *abgeleitete Relationen*.

Es gibt zwei grundlegende Bedingungen, welche wir an eine Vererbungsbeziehung stellen können:

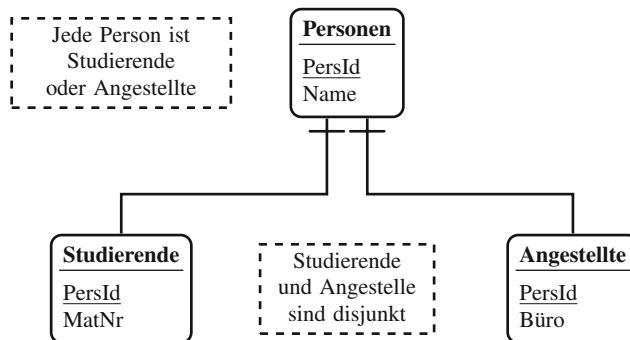
**Totalität** Die Vererbungsrelation ist *total*, wenn es für jedes Tupel in der Basisrelation ein entsprechendes Tupel in mindestens einer der abgeleiteten Relationen gibt. In unserem Beispiel heisst das, es gibt keine Entitäten die nur Personen sind. Jede Person muss Studierender oder Angestellter sein. In der objektorientierten Terminologie könnten wir sagen, Personen ist abstrakt.

**Disjunktheit** Die abgeleiteten Relationen heissen *disjunkt*, falls es kein Tupel in der Basisrelation gibt, zu welchem in mehr als einer abgeleiteten Relation ein entsprechendes Tupel existiert. In unserem Beispiel heisst das, es gibt keine Personen, die sowohl Studierende als auch Angestellte sind.

Totalität und Disjunktheit können wir nicht direkt in unserer graphischen Notation darstellen. Wir können aber entsprechende Kommentare hinzufügen, siehe Abb. 3.15.

Eine weitere wichtige Anwendung von 1:1-Beziehungen ist die Vermeidung von zu vielen Null Werten. Betrachten wir ein Schema um Personen zu verwalten, welches Attribute enthält für den Namen (Name) und das Geburtsdatum (GebDatum) der Person, sowie (falls die Person Brillenträger ist) für die Korrektur der Brille (Brille). Dieses Schema hat also die Form:<sup>5</sup>

$$\mathcal{S}_{\text{Personen}} := (\text{PersId}, \text{Name}, \text{GebDatum}, \text{Brille}).$$



**Abb. 3.15** Vererbung mit Totalität und Disjunktheit

<sup>5</sup>Wir vereinfachen hier das Beispiel und geben die Korrektur für das linke und rechte Auge nicht separat an.

Eine mögliche Instanz dieses Schemas ist:

Personen			
PersId	Name	GebDatum	Brille
1	Eva	19710429	Null
2	Tom	19720404	Null
3	Eva	19680101	-3.5
4	Ann	19841214	Null
5	Bob	20140203	Null

Natürlich gibt es viele Personen, die *keine* Brille tragen und somit ist der Wert des Attributs *Brille* in vielen Tupeln Null.

Wir werden später sehen, dass es sich lohnen kann unnötige Null Werte zu vermeiden. In diesem Beispiel gibt es dazu eine simple Methode. Wir können nämlich das Schema  $\mathcal{S}_{\text{Personen}}$  wie folgt in zwei Schemata aufteilen:

$$\begin{aligned}\mathcal{S}_{\text{AllePersonen}} &:= (\text{PersId}, \text{Name}, \text{GebDatum}) \\ \mathcal{S}_{\text{Brillenträger}} &:= (\text{PersId}, \text{Brille}),\end{aligned}$$

wobei *PersId* in  $\mathcal{S}_{\text{Brillenträger}}$  ein Fremdschlüssel auf  $\mathcal{S}_{\text{AllePersonen}}$  ist. Die obige Relation wird somit aufgeteilt in:

AllePersonen			Brillenträger	
PersId	Name	GebDatum	PersId	Brille
1	Eva	19710429	3	-3.5
2	Tom	19720404		
3	Eva	19680101		
4	Ann	19841214		
5	Bob	20140203		

Wir verwenden also *AllePersonen* als Basisklasse und haben diese zu einer einzigen abgeleiteten Klasse, nämlich *Brillenträger*, spezialisiert. Mit diesem DB-Schema lassen sich die vielen Null Werte vermeiden. Abb. 3.16 zeigt die graphische Darstellung dieses DB-Schemas.

Die Aufteilung der Tabelle *Personen* in die zwei Tabellen *AllePersonen* und *Brillenträger* vermeidet nicht nur die vielen Null Werte. Sie löst auch das Problem der zwei Bedeutungen von Null. In der Tabelle *Personen* können wir nicht unterscheiden ob,

- 1. Eva keine Brille hat oder
- 2. die Korrektur von Evas Brille unbekannt ist.

In beiden Fällen lautet der Eintrag in Personen

( 1, Eva, 19710429, Null ).

Wenn wir Brillenträger von AllePersonen ableiten, so können wir folgende Fälle unterscheiden:

1. Eva hat keine Brille. Dann gibt es in der Tabelle Brillenträger keinen Eintrag mit PersId 1.
2. Die Korrektur von Evas Brille ist unbekannt. Dann gibt es in Brillenträger einen Eintrag

( 1, Null ).

---

### 3.4 Modellierung komplexer Systeme

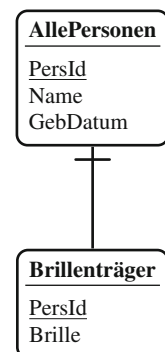
In diesem Abschnitt werden wir verschiedene Beispiele für Schemata von komplexen Datenbanken studieren. Wir beschreiben jeweils zuerst die Anforderungen an die Datenbank und geben dann die graphische Beschreibung eines entsprechenden Schemas an. Um das Schema zu entwickeln, werden wir üblicherweise Annahmen treffen müssen über Sachverhalte, die nur unvollständig in den Anforderungen spezifiziert sind. Diese zusätzlichen Annahmen werden wir ebenfalls auflisten.

*Beispiel 3.3 (Hochschul-Datenbank).*

**Anforderungen.** In einer Hochschul-Datenbank sind Daten über folgende Einzelheiten abzulegen:

1. Dozierende (Name, Fachbereich, Raum),
2. Assistenten (Name, Raum),

**Abb. 3.16** Vermeidung von Null Werten



3. Vorlesungen (Nummer, Titel, Hörsaal),
4. Übungen zu Vorlesungen (Nummer, Hörsaal),
5. Studierende (Matrikel-Nummer, Name, Studienfach),
6. Hilfsassistenten (Matrikel-Nummer, Name, Anstellungsgrad).

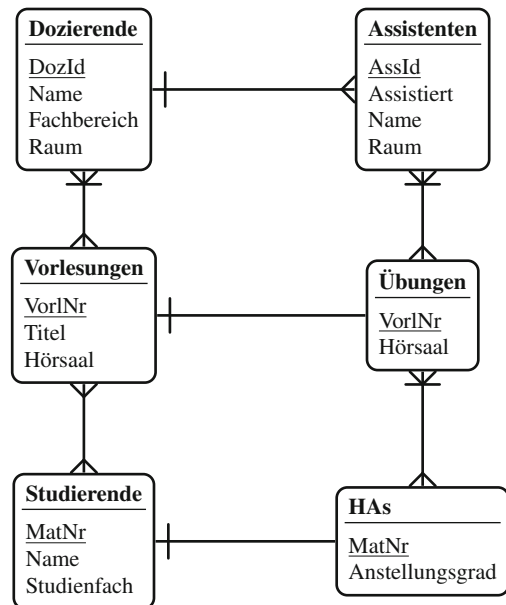
Ferner ist zu beachten: Professoren haben Assistenten und halten Vorlesungen; Assistenten betreuen Übungen, welche wiederum nur in Verbindung mit einer Vorlesung stattfinden. Einer Übung können mehrere Hilfsassistenten zugeordnet werden (zur Korrektur von Übungsserien); ein Hilfsassistent ist jedoch insbesondere ein Studierender und hört als solcher Vorlesungen.

**DB-Schema.** Das DB-Schema für die Hochschul-Datenbank ist in Abb. 3.17 dargestellt.

**Überlegungen zum Schema.** Zusätzlich zu den Angaben in der Spezifikation haben wir folgende Annahmen getroffen:

1. DozId (in Dozierende) und AssId (in Assistenten) sind zusätzliche Attribute welche jeweils als Primärschlüssel dienen.
2. Ein Assistent ist bei *genau* einem Dozierenden angestellt. Um dies zu modellieren, ist in der Tabelle Assistenten das Attribut Assistiert ein Fremdschlüssel auf Dozierende.
3. Zu einer Vorlesung gibt es nur eine Übung und Übungen gibt es nicht ohne entsprechende Vorlesung.

**Abb. 3.17** Hochschul-Datenbank



4. HAS (Hilfsassistenten) ist eine Spezialisierung von Studierende. Entsprechend wird das Attribut Name von Studierende geerbt und muss nicht extra in HAS erfasst werden.
5. Ein Hilfsassistent kann mehrere (muss jedoch mindestens eine) Übungen betreuen.
6. Ein Assistent kann mehrere (ev. auch keine) Übungen betreuen. Jede Übung hat mindestens einen Assistenten.
7. Das Schema garantiert nicht, dass ein Assistent, welcher die Übungen zu einer Vorlesung betreut, auch bei einem Dozierenden dieser Vorlesung angestellt ist.

Wir haben in Abb. 3.17 die abgekürzte Notation für m:n-Beziehungen verwendet. Das vollständige Diagramm (mit allen Tabellen, welche im DB-Schema vorkommen werden) ist in Abb. 3.18 dargestellt.

*Beispiel 3.4 (Warenhauskette).*

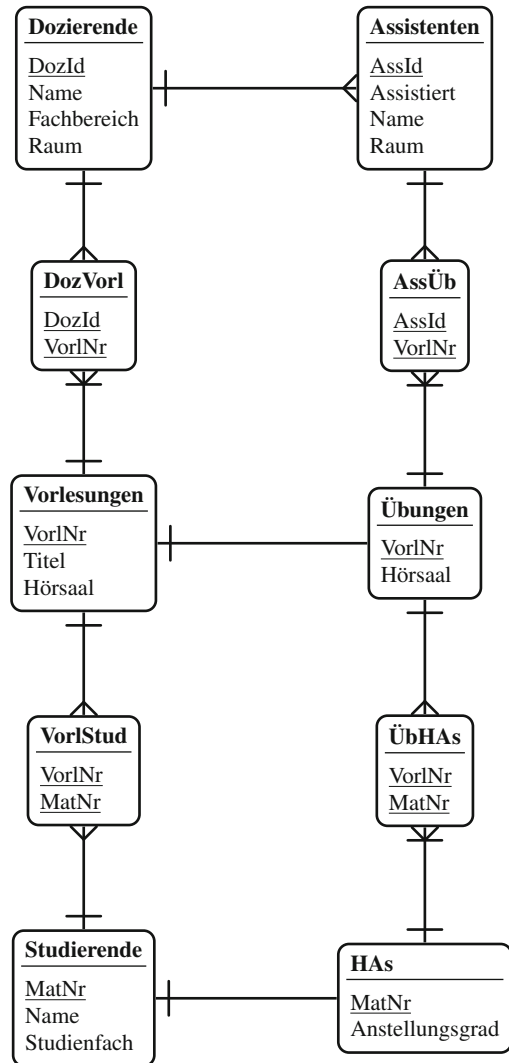
**Anforderungen.** Eine Datenbank für die Lagerverwaltung einer Warenhauskette soll Auskunft geben über das Sortiment jeder Filiale. Ausserdem soll für jeden Lieferanten ersichtlich sein, welchen Artikel er zu welchem Preis liefern kann. Für jeden Artikel sollen die Bezeichnung, für jeden verkauften Artikel der Verkaufspreis und das Verkaufsdatum, und für jeden gelieferten Artikel das Lieferdatum gespeichert werden.

**DB-Schema.** Das DB-Schema für die Warenhauskette ist in Abb. 3.19 dargestellt.

### Überlegungen zum Schema.

1. Wir haben allen Tabellen Id-Attribute hinzugefügt, welche als Primärschlüssel dienen.
2. Ein Artikel ist ein bestimmter Gegenstand, der an Lager sein kann oder bereits verkauft wurde. Falls z. B. die Warenhauskette Computer verkauft, so sind die einzelnen Computer Artikel. Ein bestimmtes Computermodell ist dann ein Artikeltyp.
3. Ein Artikel wird in der Artikel-Tabelle eingetragen, wenn er ins Lager kommt. Wird er dann verkauft, so wird er auch noch in die Tabelle Verkäufe eingetragen (bleibt aber in Artikel).
4. Preis ist ein Attribut von TypLiefer, das angibt, welcher Lieferant diesen Artikeltyp gegenwärtig zu welchem Preis liefert.
5. Das Attribut Verkaufspreis in Artikeltyp gibt an, was der aktuelle Verkaufspreis für Artikel dieses Typs ist. So ist gewährleistet, dass alle gleichen Artikel auch gleich viel kosten. Bei Preisanpassungen (Aktionen) ist nur ein Update nötig und es muss nicht der Preis bei allen Artikeln einzeln angepasst werden.
6. Das Attribut Ankaufspreis in Artikel gibt an, zu welchem Preis dieser Artikel tatsächlich eingekauft wurde. Der aktuelle Preis, der in der Tabelle TypLiefer abgespeichert ist, kann davon abweichen.
7. Das Attribut Verkaufspreis in Verkäufe gibt an, zu welchem Preis dieser Artikel tatsächlich verkauft wurde. Der aktuelle Wert von Verkaufspreis in Artikeltyp kann davon abweichen.

**Abb. 3.18** Vollständiges  
Diagramm der  
Hochschul-Datenbank

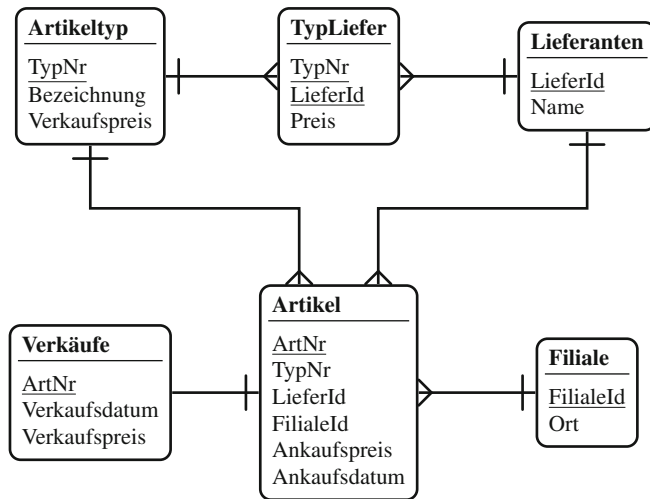


**Variante.** Im DB-Schema für die Warenhauskette haben wir zur Vermeidung von Null Werten das Muster aus dem Beispiel der Brillenträger (siehe Abb. 3.16) angewendet. Das heisst, wir haben die Tabellen Artikel und Verkäufe. So gibt es keine Null Werte in den Attributen Verkaufsdatum und Verkaufspreis.

Gemäss Spezifikation soll unsere Datenbank zur Lagerverwaltung der Warenhauskette dienen. Jedoch ist es in unserem DB-Schema sehr aufwändig folgende Abfrage zu bearbeiten:

Welches ist der aktuelle Lagerbestand eines gegebenen Artikeltyps? (3.1)



**Abb. 3.19** Warenhauskette

Dazu müssen wir nämlich diejenigen Artikel des gegebenen Typs finden, welche *nicht* in der Verkäufe-Tabelle eingetragen sind.

Wir können diese Abfrage vereinfachen, indem wir die beiden Tabellen wieder zusammenfügen. Damit wird die Tabelle Verkäufe nicht mehr benötigt und sowohl Verkaufspreis als auch Verkaufsdatum sind Attribute von Artikel. Diese Attribute haben den Wert Null, falls der Artikel noch nicht verkauft wurde. Um (3.1) zu beantworten, können wir nun alle Artikel des gegebenen Typs suchen, deren Verkaufsdatum den Wert Null hat.

Entsprechend können wir die Abfrage

Finde alle verkauften Artikel eines gegebenen Typs (3.2)

beantworten, indem wir alle Artikel des gegebenen Typs suchen, deren Verkaufsdatum nicht Null ist.

Abhängig von den verwendeten Abfragen und der konkreten Verteilung der Daten kann es sein, dass diese Tests auf Null (bzw. auf nicht Null) nicht effizient durchgeführt werden können.<sup>6</sup> Unter Umständen kann es sich lohnen, der Tabelle Artikel ein neues Attribut Status hinzuzufügen, welches zwei Werte annehmen kann: *an Lager* und *verkauft*. Nun können wir (3.2) ganz einfach beantworten, indem wir auf den Status *verkauft* testen. In der endgültigen Version braucht es dann die Tabelle Verkäufe nicht mehr und die Tabelle Artikel hat die Attribute ArtNr, TypNr,

<sup>6</sup>Damit ist gemeint, dass die aufgebauten Indizes diese Abfragen nicht beschleunigen können, siehe Abschn. 7.1.

LieferId, FilialeId, Ankaufspreis, Ankaufsdatum, Verkaufspreis, Verkaufsdatum und Status.

---

## Weiterführende Literatur<sup>7</sup>

1. Chen, P.P.S.: The entity-relationship model: toward a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976). <https://doi.org/10.1145/320434.320440>
2. Martin, J., Odell, J.J.: Object-oriented Analysis and Design. Prentice-Hall, Inc., Upper Saddle River (1992)
3. Simsion, G.: Data Modeling: Theory and Practice. Technics Publications. ISBN: 978-0977140015 (2007)

---

<sup>7</sup>Bereits Mitte der 70er- Jahre entwickelte Chen [1] mit den Entity-Relationship Diagrammen (ER-Diagramme) eine einflussreiche konzeptionelle Modellierungssprache zur Unterstützung des Datenbank-Entwurfs. Wir verwenden in unserem Buch jedoch eine andere, einfachere Diagramm-Sprache, nämlich die sogenannten Krähenfuss-Diagramme (auch Martin Notation genannt) [2], welche in der Praxis häufiger eingesetzt wird [3]. Der Vorteil der Krähenfuss Notation gegenüber ER-Diagrammen ist die kleinere semantische Distanz, das heisst, die Modellierungssprache ist näher bei der Implementierungssprache.