Kapitel 3

Logische Datenmodelle

In Abhängigkeit von dem zu verwendenden Datenbanksystem wählt man zur computergerechten Umsetzung des Entity-Relationship-Modells das hierarchische, das netzwerkorientierte, das relationale oder das objektorientierte Datenmodell.

3.1 Das Hierarchische Datenmodell

Datenbanksysteme, die auf dem hierarchischen Datenmodell basieren, haben (nach heutigen Standards) nur eine eingeschränkte Modellierfähigkeit und verlangen vom Anwender Kenntnisse der interne Ebene. Trotzdem sind sie noch sehr verbreitet (z.B. IMS von IBM), da sie sich aus Dateiverwaltungssystemen für die konventionelle Datenverarbeitung entwickelt haben. Die zugrunde liegende Speicherstruktur waren Magnetbänder, welche nur sequentiellen Zugriff erlaubten.

Im Hierarchischen Datenmodell können nur baumartige Beziehungen modelliert werden. Eine Hierarchie besteht aus einem Wurzel-Entity-Typ, dem beliebig viele Entity-Typen unmittelbar untergeordnet sind; jedem dieser Entity-Typen können wiederum Entity-Typen untergeordnet sein usw. Alle Entity-Typen eines Baumes sind verschieden.

Abbildung 3.1 zeigt ein hierarchisches Schema sowie eine mögliche Ausprägung anhand der bereits bekannten Universitätswelt. Der konstruierte Baum ordnet jedem Studenten alle Vorlesungen zu, die er besucht, sowie alle Professoren, bei denen er geprüft wird. In dem gezeigten Baum ließen sich weitere Teilbäume unterhalb der *Vorlesung* einhängen, z.B. die Räumlichkeiten, in denen Vorlesungen stattfinden. Obacht: es wird keine Beziehung zwischen den Vorlesungen und Dozenten hergestellt! Die Dozenten sind den Studenten ausschließlich in ihrer Eigenschaft als Prüfer zugeordnet.

Grundsätzlich sind einer Vater-Ausprägung (z.B. `Erika Mustermann`) für jeden ihrer Sohn-Typen jeweils mehrere Sohnausprägungen zugeordnet (z.B. könnte der Sohn-Typ `Vorlesung` 5 konkrete Vorlesungen enthalten). Dadurch entsprechen dem Baum auf Typ-Ebene mehrere Bäume auf Entity-Ebene. Diese Entities sind in Preorder-Reihenfolge zu erreichen, d.h. vom Vater zunächst seine Söhne und Enkel und dann dessen Brüder. Dieser Baumdurchlauf ist die einzige Operation auf einer Hierarchie; jedes Datum kann daher nur über den Einstiegspunkt Wurzel und von dort durch Überlesen nichtrelevanter Datensätze gemäß der Preorder-Reihenfolge erreicht werden.

Die typische Operation besteht aus dem Traversieren der Hierarchie unter Berücksichtigung der jeweiligen Vaterschaft, d. h. der Befehl `GNP VORLESUNG` (gesprochen: GET NEXT VORLESUNG

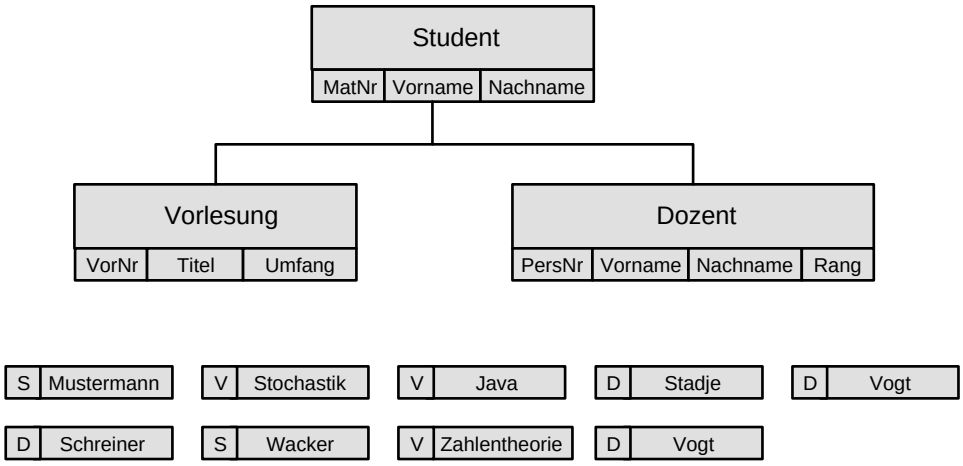

Abbildung 3.1: Hierarchisches Schema und eine Ausprägung.
Der Recordtyp sei erkennbar an den Zeichen S (Studenten), V (Vorlesungen) und D (Dozenten)

WITHIN PARENT) durchläuft sequentiell ab der aktuellen Position die Preorder-Sequenz solange vorwärts, bis ein dem aktuellen Vater zugeordneter Datensatz vom Typ *Vorlesung* gefunden wird.

Um zu vermeiden, daß alle Angaben zu den Dozenten mehrfach gespeichert werden, kann eine eigene Hierarchie für die Dozenten angelegt und in diese dann aus dem Studentenbaum heraus verwiesen werden.

Es folgt ein umfangreicheres Beispiel, entnommen dem Buch von C.J. Date. Abbildung 3.2 zeigt das hierarchische Schema, Abbildung 3.3 eine Ausprägung.

Die Query *Welche Studenten besuchen den Kurs M23 am 13.08.1973?* wird unter Verwendung der DML (Data Manipulation Language) derart formuliert, daß zunächst nach Einstieg über die Wurzel der Zugriff auf den gesuchten Kurs stattfindet und ihn als momentanen Vater fixiert. Von dort werden dann genau solche Records vom Typ *Student* durchlaufen, welche den soeben fixierten Kursus als Vater haben:

```
GU COURSE   (COURSE#='M23')
   OFFERING (DATE='730813');
if gefunden then
begin
  GNP STUDENT;
  while gefunden do
  begin
    write (STUDENT.NAME);
    GNP STUDENT
  end
end;
```

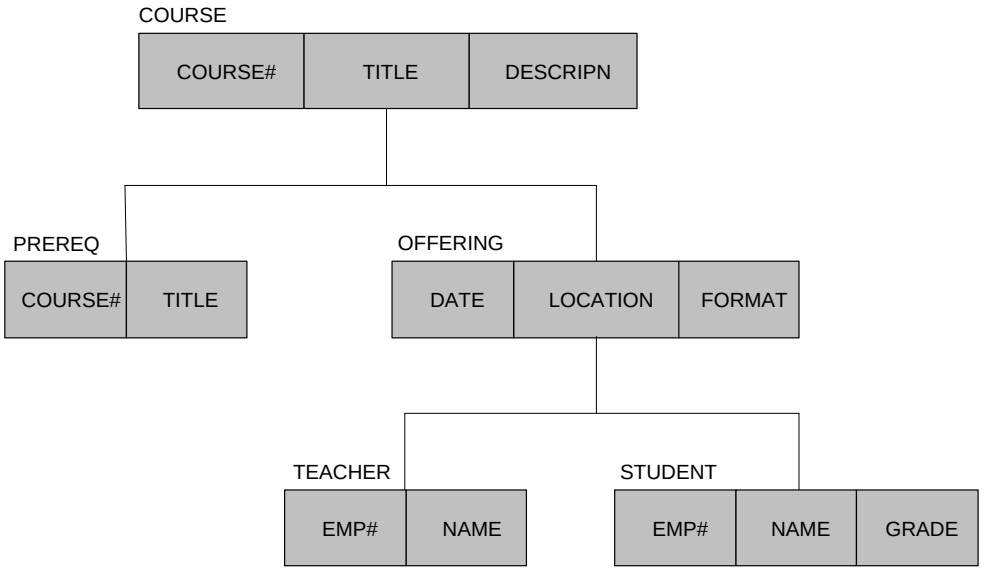

Abbildung 3.2: Beispiel für ein hierarchisches Schema

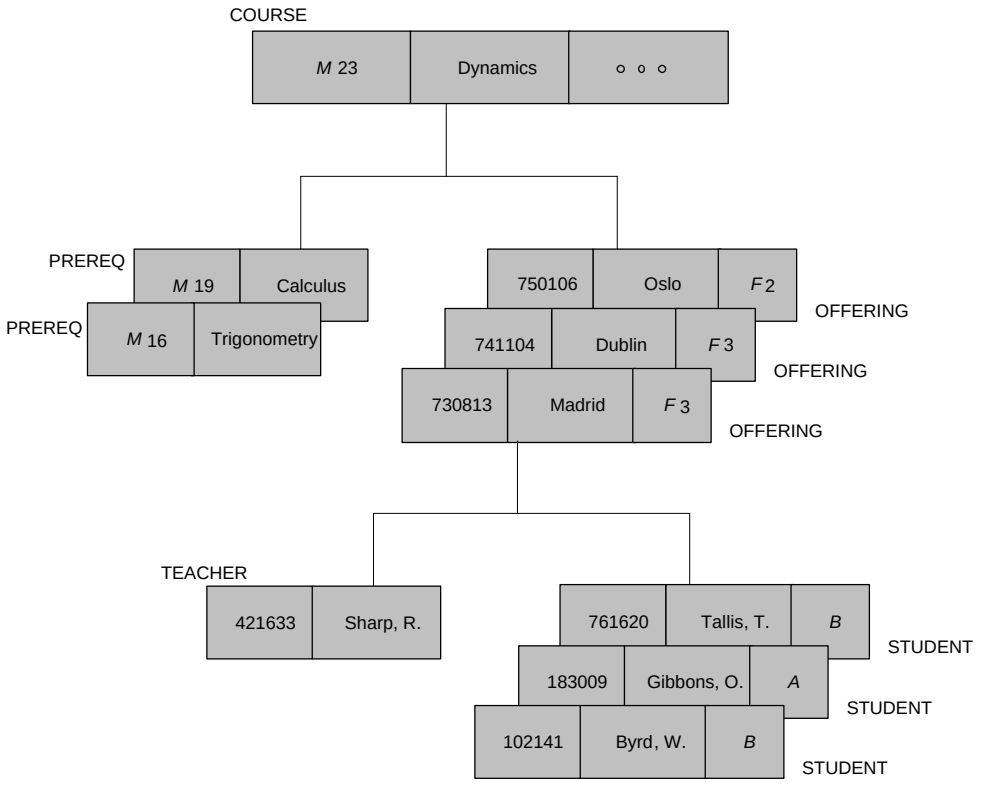

Abbildung 3.3: Beispiel für eine Ausprägung des hierarchischen Schemas

3.2 Das Netzwerk-Datenmodell

Im Netzwerk-Datenmodell können nur binäre many-one- (bzw. one-many)-Beziehungen dargestellt werden. Ein E-R-Diagramm mit dieser Einschränkung heißt *Netzwerk*. Zur Formulierung der many-one-Beziehungen gibt es sogenannte *Set-Typen*, die zwei Entity-Typen in Beziehung setzen. Ein Entity-Typ übernimmt mittels eines Set-Typs die Rolle des *owner* bzgl. eines weiteren Entity-Typs, genannt *member*.

Im Netzwerk werden die Beziehungen als gerichtete Kanten gezeichnet vom Rechteck für *member* zum Rechteck für *owner* (funktionale Abhängigkeit). In einer Ausprägung führt ein gerichteter Ring von einer *owner*-Ausprägung über alle seine *member*-Ausprägungen (Abbildung 3.4).

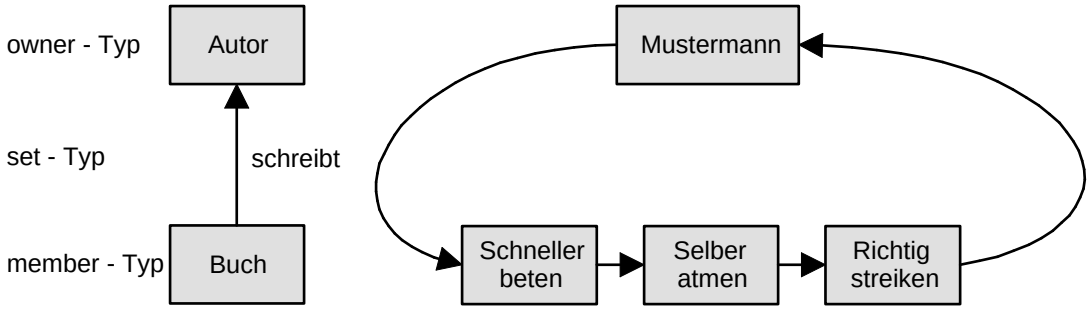

Abbildung 3.4: Netzwerkschema und eine Ausprägung

Bei nicht binären Beziehungen oder nicht many-one-Beziehungen hilft man sich durch Einführung von künstlichen *Kett-Records*. Abbildung 3.5 zeigt ein entsprechendes Netzwerkschema und eine Ausprägung, bei der zwei Studenten jeweils zwei Vorlesungen hören.

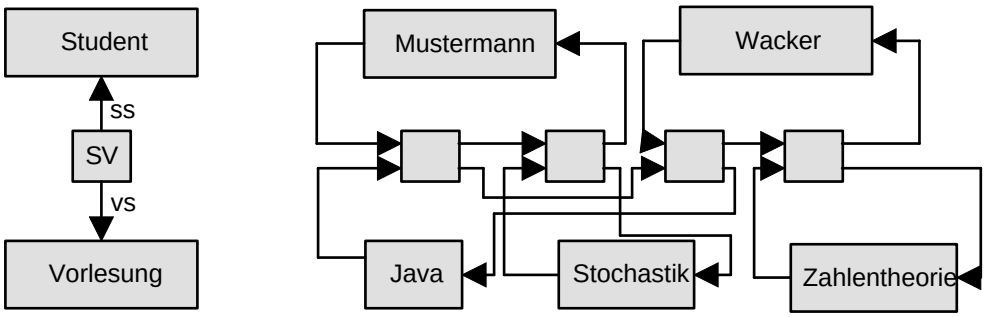

Abbildung 3.5: Netzwerkschema mit Kett-Record und eine Ausprägung

Die typische Operation auf einem Netzwerk besteht in der Navigation durch die verzeigerten Entities. Mit den Befehlen

```
FIND NEXT Student
FIND NEXT sv WITHIN ss
FIND OWNER WITHIN vs
```

lassen sich für einen konkreten Studenten alle seine Kett-Records vom Typ `sv` durchlaufen und dann jeweils der *owner* bzgl. des Sets `vs` ermitteln.

```
STUDENT.NAME := 'Mustermann';
FIND ANY STUDENT USING NAME;
if gefunden then
begin
  FIND FIRST SV WITHIN SS;
  while gefunden do
  begin
    FIND OWNER WITHIN VS;
    GET VORLESUNG;
    WRITE(VORLESUNG.TITEL);
    FIND NEXT VORLESUNG WITHIN VS;
  end
end;
```

3.3 Das Relationale Datenmodell

Seien $D_1, D_2, \ldots, D_k$ Wertebereiche. $R \subseteq D_1 \times D_2 \times \ldots \times D_k$ heißt Relation. Wir stellen uns eine Relation als Tabelle vor, in der jede Zeile einem Tupel entspricht und jede Spalte einem bestimmten Wertebereich. Die Folge der Spaltenidentifizierungen heißt *Relationenschema*. Eine Menge von Relationenschemata heißt *relationales Datenbankschema*, die aktuellen Werte der einzelnen Relationen ergeben eine Ausprägung der relationalen Datenbank.

- pro Entity-Typ gibt es ein Relationenschema mit Spalten benannt nach den Attributen.
- pro Relationshiptyp gibt es ein Relationenschema mit Spalten für die Schlüssel der beteiligten Entity-Typen und ggf. weitere Spalten.

Abbildung 3.6 zeigt ein Schema zum Vorlesungsbetrieb und eine Ausprägung. Hierbei wurden die über ihre Tabellengrenzen hinausgehenden und zueinander passenden Attribute jeweils gleich genannt, um zu verdeutlichen, daß sie Daten mit derselben Bedeutung speichern.

Student

MatNr	Vorname	Nachname
653467	Erika	Mustermann
875462	Willi	Wacker
432788	Peter	Pan

Hoert

MatNr	VorNr
653467	6.718
875462	6.718
432788	6.718
875462	6.108

Vorlesung

VorNr	Titel	Umfang
6.718	Java	4
6.174	Stochastik	2
6.108	Zahlentheorie	4

Abbildung 3.6: Relationales Schema und eine Ausprägung

Die typischen Operationen auf einer relationaler Datenbank lauten:

- **Selektion:**
  Suche alle Tupel einer Relation mit gewissen Attributeigenschaften
- **Projektion:**
  filtere gewisse Spalten heraus
- **Verbund:**
  Finde Tupel in mehreren Relationen, die bzgl. gewisser Spalten übereinstimmen.

Beispiel-Query: Welche Studenten hören die Vorlesung Zahlentheorie ?

```
SELECT Student.Nachname from Student, Hoert, Vorlesung
WHERE  Student.MatNr   = Hoert.MatNr
AND    Hoert.VorNr     = Vorlesung.VorNr
AND    Vorlesung.Titel = "Zahlentheorie"
```

3.4 Das Objektorientierte Modell

Eine Klasse repräsentiert einen Entity-Typ zusammen mit darauf erlaubten Operationen. Attribute müssen nicht atomar sein, sondern bestehen ggf. aus Tupeln, Listen und Mengen. Die Struktur einer Klasse kann an eine Unterklasse vererbt werden. Binäre Beziehungen können durch mengenwertige Attribute modelliert werden.

Die Definition des Entity-Typen *Person* mit seiner Spezialisierung *Student* incl. der Beziehung *hoert* sieht im objektorientierten Datenbanksystem O_2 wie folgt aus:

```
class Person
  type tuple (name        : String,
              geb_datum   : Date,
              kinder      : list(Person))
end;

class Student inherit Person
  type tuple (mat_nr      : Integer,
              hoert       : set (Vorlesung))
end;

class Vorlesung
  type tuple (titel       : String,
              gehoert_von : set (Student))
end;
```