Datenbanksysteme 1

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Mihhail Aizatulin (avatar@hot.ee)
und Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

Hinweise

- Literaturempfehlung:
  - A.Heuer, G. Saake – Datenbanken; Thompson, 2003.
  - J. Biskup – Informationssysteme; Vieweg, 1995. Mit mathematischen Grundlagen; das Buch ist inzwischen schwer zu bekommen (auch nicht in der Uni-Bibliothek erhältlich).
  - A.Kemper, A.Eikler – Datenbanksysteme; Oldenbourg, 2003. Vollständig, mit Einblick in die Technologie der Datenbanken; ziemlich fehlerfrei.
  - B. Thalheim – Entity-Relationship-Modeling; Springer, 2000. Mit Discount erhältlich.

  Das zweite und das dritte Buch im elektronischen Format sowie weitere Literaturangaben findet man auf der Website zur Vorlesung [Tha04].
- Es ist ein Praktikum vorgesehen – ein Projekt zur Verwaltung von E-Mails.
- Anschlussvorlesungen:
  - Datenbanksysteme 2 (Modellierung)
  - Datenbanksysteme 3 (Programmierung und Technologie)
  - Datenbanktheorie

  Parallel läuft eine vier-semestrige Vorlesungsreihe:
  - Verteilte Datenbanken
  - Intelligente Informationssysteme
  - Web-Informationssysteme (im nächsten Wintersemester)
  - Datenbank-Anwendungen (in verschiedenen Facetten)

1 Grundlagen und Ziele

Informationssysteme sind heute:

- *Datenbanksysteme:* Datenbanken mit (generischen) Funktionen für Retrieval und Modifikation
- *Information-Retrieval-Systeme:* Hypertext-Systeme wie Google (ausgereifte, unscharfe Suchfunktionen)
- *Entscheidungsunterstützungssysteme:* Anfragen zu Hypothesen (darauf wird in der Vorlesung nicht eingegangen!)
- *Management-Informationssysteme:* schließen Funktionen zum „Entdecken“ von Zusammenhängen mit ein (Data Mining)
- *Dokumenteninformationssysteme*
- *Wissensbanken*

Definition: Ein *Datenbanksystem* besteht aus einem *Datenbank-Management-System (DBMS)* und einer Menge von Datenbanken. Eine *Datenbank* ist eine große Menge (meist einheitlich bezüglich eines Schema) strukturierter Daten.

An ein Datenbanksystem werden folgende **Anforderungen** gestellt:

- Systeme müssen mit *großen Datenmengen* umgehen können
- Daten müssen *dauerhaft (persistent)* sein
- Daten müssen *verläßlich (dependable)* abgelegt sein: Daten müssen unabhängig vom Systemzustand bestehen bleiben
- *verschiedene Benutzer (shared)* mit unterschiedlichen Sichtweisen
- *Sicherheit* der Daten, d.h. es können Rechte vergeben werden, und Aktionen müssen später nachweisbar sein
- *Effizienz*: nur *schnelle Algorithmen*, keine mit schlechterer Laufzeit als $O(n \log n)$

Bestandteile eines modernen DBMS:

- auf der untersten Ebene: Betriebssystem, Data Dictionary, Logbuch, Support Systeme etc

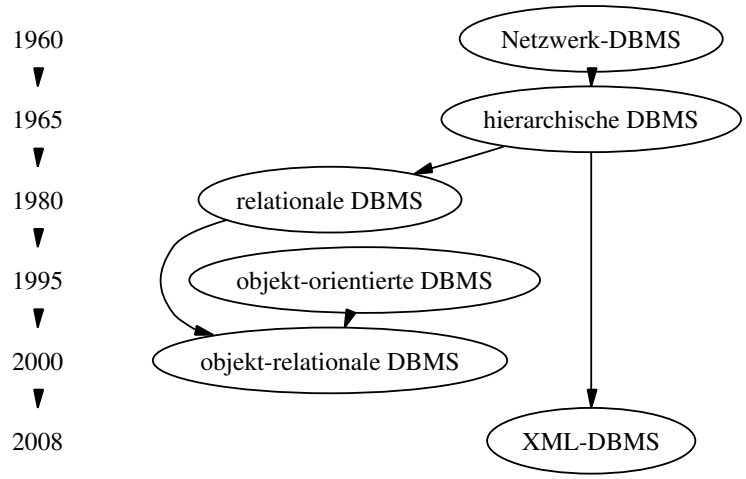


Abbildung 1: zur **Geschichte** von Datenbanken

- Buffer-Management, da sich die Übertragungsgeschwindigkeit der Speichermedien sich mit der Zeit sehr langsam verbessert
- Synchronisation paralleler Zugriffe
- Compiler, damit das System Hardware-unabhängig ist
- Kommunikationssystem für die Kommunikation mit dem Benutzer

Verbreitete DBMS[1] sind vor allem Oracle, DB2 (von IBM), der MSSQL-Server, Sybase und Informix (jetzt von IBM gekauft). Wir werden uns meist für Oracle, DB2 und Sybase interessieren, zu allen dreien gibt es auch Linux-Varianten (mit unterschiedlichen Implementationsbeschränkungen, z.B. bei Oracle).

Es gibt zudem eine Vielzahl von Datenbank-Systemen für PCs, z.B. MySQL – diese haben allerdings viel weniger Funktionalität als die oben genannten Systeme.

Beispiele für eingesetzte Sprachen:

- *Structured Query Language (SQL)*, verschiedene Standards (1980, 1989, 1992, 1999, 2003), hier wird der 1992er-Standard behandelt
- *Query-by-Example (QBE)*

[1] Die „Halbwertzeit“ schätzt man bei Software auf 2,5 Jahre, bei Datenbanken sind es ca. 10 Jahre, bei Datenbankanwendungen sogar 20 Jahre!

Man unterscheidet zwischen verschiedenen **Arten von Sprachen** [Tha04]:

- *Datenbankdefinitionssprache (DDL):* Definition des konzeptionellen und der externen Schemata – daraus wird Speicherbelegung abgeleitet (Storage Definition Language)
- *Datenmanipulationssprache (DML):*
  - *Retrieval (Selektion)* für die Anfrage
  - *Insert* für das Einfügen von (noch nicht vorhandenen) Daten
  - *Delete* für das Streichen von (vorhandenen) Daten
  - *Update* für die Modifikation von (lokalisierbaren) Daten

  Sprache ist i.a. nicht prozedural, sie ist mengenorientiert und deklarativ.
- *Datensicht(erzeug-)Sprache (VDL)*
- *Datenkontrollsprache (DCL):* zur Steuerung der Datenbank (`GRANT, REVOKE, COMMIT, ROLLBACK, ASSIGN`)
- *Wirtssprache (Host Language)* des unterlegten Programmiersystemes

Die Sprachen sollen sauber definierbar sein!

2 Modellierung und Sichtweisen

Ziel: Bei Modellierung wollen wir von Details der realen Welt abstrahieren und verwenden daher *Abstraktionen* es gibt folgende *Abstraktionsarten*:

- *Komponentenabstraktion:* Wir beschreiben, wie die Komponenten interagieren. Dazu verwenden wir drei Methoden:
  - Gruppierung – Klassifikation
  - Spezifikation – Generalisierung
  - Aggregation – Konstruktion
- *Lokalisationsabstraktion:* Man betrachtet nur ein Teil der Welt – unterschiedliche Sichtweisen, Parametrisierung.
- *Implementationsabstraktion:* Hiding/Kapselung

Unterscheide folgende **Gesichtspunkte**:

- *statische Gesichtspunkte*
  - *Struktur:* Wir versuchen, die Basisstruktur zu bestimmen und dann zu bestimmen, was die ableitbaren Strukturen sind.
  - *Statische Semantik:* Existenzpostulate, Aussonderungspostulate, Verallgemeinerungspostulate
- *dynamische Gesichtspunkte*
  - Operationen (Änderungen, Retrieval)
  - dynamische Semantik
- *Agenten:* Rollen und Rechte

Unterscheidung [Wik04]: Die *ANSI-SPARC-Architektur*, auch *Drei-Ebenen-Architektur*, beschreibt die grundlegende Definition eines Datenbanksystems:

- die *externe Ebene*, die dem Anwender eine individuelle Benutzersicht bereitstellt;
- die *konzeptionelle Ebene*, in der beschrieben wird, welche Daten in der Datenbank gespeichert sind sowie deren Beziehungen untereinander;
- die *interne Ebene*, die die physische Sicht der Datenbank im Computer darstellt. In ihr wird beschrieben, wie die Daten in der Datenbank gespeichert werden.

3 Relationale Datenbanken

3.1 Relationales Modell (Spezifikationssprache)

Ziel: Wir versuchen, ein Sprache zu finden, die eine möglichst einfache Struktur hat und einfach zu implementieren ist.

Unsere **Vorstellung:** Die Anwendungswelt lässt sich (über Basis-Datentypen) durch *Relationen* darstellen. Wir benutzen dabei *Tabellen* als Repräsentationsform[2]. Interpretation ist dabei, daß in den Spalten Attribute stehen und in den Zeilen die Objekte.

3.1.1 Attribute, Schemata, Datenbanken

Definition: Ein *Basisdatentyp* $DT = (\text{domain}(DT), \text{op}(DT), \text{pred}(DT))$ verfügt über einen *Wertebereich* $\text{domain}(DT)$, eine Menge von *Operationen* $\text{op}(DT)$ und eine Menge von *Prädikaten* $\text{pred}(DT)$. Bei den Prädikaten setzen wir voraus, dass die Gleichheit „=“ dazugehört. Die Menge der Datentypen bezeichnen wir mit $\mathcal{BT}$.

Weitere spezifische Eigenschaften sind:

- Präzision/Genauigkeit
- Granularität – wie genau sollen die realen Objekte abgebildet werden? Wollen wir z.B. den vollen Namen einer Person nehmen oder nur den Vornamen?
- Ordnungsrelationen
- Klassifikation
- Speicherrepräsentation
- Präsentationsformate
- Standard-Werte
- Rundungsregeln
- zugeordnete Maßeinheiten mit ggf. Umwandlungsregeln
- Aggregationsoperationen, z.B. Durchschnittsbildung

[2]Tabelle und Relation sind aber nicht das gleiche: Tabellen sind Multimengen (d.h. die gleichen Elemente können mehrmals vorkommen), zudem besitzen sie eine Darstellungsordnung! Relation dagegen ist eine Menge ohne Ordnung.

Definition: Ein *Relationenschema* $\underline{R}$ ist von der Form (R, K, Σ) mit

- einer Menge von *Attributen* $R \subseteq U$, wobei U der *universelle Namensraum* für Eigenschaften der Dinge der Anwendungswelt ist – den Attributen wird ein *Wertebereich (Domäne)* zugeordnet mittels $\mathrm{dom}_R \colon U \circ\!\!\rightarrow \mathcal{BT}$,
- einer Menge von *Schlüsseln* zur Hauptidentifikation $K \subseteq R$,
- einer Menge von *Integritätsbedingungen (FD)* Σ (dazu später mehr).

Definitionen:

- Ein *Tupel* t über $\underline{R}$ ist eine Funktion

$$t \colon R \to \bigcup_{A \in R} \mathrm{dom}_R(A) \;\text{ mit }\; t(A) \in \mathrm{dom}_R(A).$$

- Eine *Klasse (Instanz)* $\underline{R}^C$ von $\underline{R}$ ist eine Menge von Tupeln über $\underline{R}$. Integritätsbedingungen spezifizieren zugelassene Klassen.
- Ein *relationales Datenbankschema* ist eine Folge $\underline{DB} = (\underline{R}_1, \ldots, \underline{R}_n, \Phi)$ von Relationenschemata mit einer weiteren Menge Φ von Integritätsbgedingungen.
- Eine *Datenbank* $\underline{DB}^C = (\underline{R}_1^C, \ldots, \underline{R}_n^C)$ zum Datenbank-Schema $\underline{DB}$ ist eine Folge von Klassen, die Φ genügen.

Darstellung: Wir verwenden Hypergraphen als Repräsentationsform von Datenbank-Schemata. Ein Knoten entspricht einem Attribut, eine Kante einem Relationenschema. Die Kanten können mehrere Knoten enthalten, und zwei Kanten können sich in mehreren Knoten schneiden.

Dynamische Veränderung von Datenbanken modellieren wir als eine Folge von Datenbanken: Ein *dynamisches Datenbankschema* ist von der Form ${}^{dyn}\underline{DB} = (\underline{DB}, {}^{dyn}\Phi)$, dann muß eine Folge $\left\langle \underline{DB}_0^C, \underline{DB}_1^C, \ldots \right\rangle$ von Datenbanken der Integritätsbedingung ${}^{dyn}\Phi$ genügen.

3.1.2 Integritätsbedingungen

Klassifizierung der Integritätsbedingungen:

- *Statische:* Damit spezifiziert man, welche Zustände von Relationen und Klassen zugelassen sind.
  - *Gleichungspostulate:* Wir definieren, wann bestimmte Daten gleiche Werte besitzen müssen. Wenn z.B. zwei Matrikelnummern gleich sind, sollen auch die Namen, Hauptfach usw. gleich sein. Oder: „Kein Raum ist im gleichen Semester zur gleichen Zeit doppelt belegt."
  - *Existenzpostulate:* z.B. „Eine Vorlesung, die man belegt hat, muss auch existent sein."
  - *Anzahlbeschränkungen:* z.B. „Jeder Professor liest mindestens 8 SWS Vorlesungen."
- *Dynamische:* Man spezifiziert, welche Folgen zugelassen sind:
  - *Transitionsbedingungen*, die auf dem Zustand und dem Folgezustand definiert sind,
  - *temporale Formeln*, z.B. „Ein Student, der immatrikuliert ist, wird irgendwann exmatrikuliert."

  Bei den dynamischen Integritätsbeziehungen sollte man möglichst sparsam sein, damit die Effizienz nicht leidet.

3.1.3 Schlüssel

Zu **Identifizierung** von Objekten: Einzelne Elemente von Mengen (Klassen) sind identifizierbar über ihre *Werte* (d.h. alle ihre Komponenten). Sie sind z.T. aber auch schon identifizierbar über *einige* ihrer Komponenten, diese Identifikationskomponenten nennt man *Schlüssel*. Von Interesse sind *minimale Schlüssel*; ein ausgezeichneter minimaler Schlüssel ist der *Primärschlüssel*.

Definition: $X \subseteq R$ heißt *Schlüssel* von $\underline{R}^C$, falls alle Objekte von $\underline{R}^C$ verschiedene X-Werte haben. X heißt *minimaler Schlüssel*, falls X ein Schlüssel ist und X nicht bezüglich der Schlüsseleigenschaft reduzierbar ist, d.h. aus X kein Element entfernt werden kann, ohne daß X die Schlüsseleigenschaft verliert.

Beispiel: Gegeben seien folgende Relationen Student und Vorlesung:

Personal-nummer	Matrikel-nummer	Haupt-fach	Neben-fach	Betreuer
...	...	...	...	...

Kurs	Raum	Zeit	Semester	Lesender
...	...	...	...	...

Minimale Schlüssel wären dann:

- Student: {Matrikelnummer, Nebenfach}, {Personalnummer, Nebenfach}
- Vorlesung: {Kurs, Semester, Zeit}, {Raum, Semester, Zeit}, {Lesender, Semester, Zeit}

Lemma: Sind X und Y minimale Schlüssel von $\underline{R}$, so gilt: $X \not\subseteq Y$ und $Y \not\subseteq X$.

Folgerung: Für die Menge der minimalen Schlüssel für relationale Schemata mit n Attributen gilt

$$|K| \leq \binom{n}{\lfloor \frac{n}{2} \rfloor}$$

Leider gibt es **große Systeme minimaler Schlüssel**, die $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ Elemente besitzen! Sei $R = (A_1, \ldots, A_n)$ eine Attributmenge, wo alle Attribute den gleichen Wertebereich haben. Sei

$$K = \left\{ X \subseteq R \mid |X| = \left\lfloor \frac{n}{2} \right\rfloor = k \right\}$$

Wir schreiben $K = \{X_1, \ldots, X_m\}$ mit $m = \binom{n}{\lfloor \frac{n}{2} \rfloor}$. Seien $a_1, \ldots, a_m, b_1, \ldots, b_m$ paarweise verschiedene Attributwerte. Wir definieren

$$\underline{R}^C = \bigcup_{i=1,\ldots,m} \{(a_i, \ldots, a_i)\} \cup \left(\bigcup_{A \in X} \{ t \mid t(A) = b_i \wedge t[X \setminus A] = a_i \} \right)$$

Anschaulich: falls $X_1 = \{A_1, \ldots, A_k\}$ und $X_2 = \{A_1, \ldots, A_{k-1}, A_{k+1}\}$, so

würde die Relation $\underline{R}^C$ unter anderem folgende Tupel enthalten:

A_1	A_2	A_3	A_4	$\ldots$	A_k	A_{k+1}	$\ldots$	A_n
a_1	a_1	a_1	$\ldots$	a_1	a_1	a_1	$\ldots$	a_1
b_1	a_1	a_1	$\ldots$	a_1	a_1	a_1	$\ldots$	a_1
a_1	b_1	a_1	$\ldots$	a_1	a_1	a_1	$\ldots$	a_1
				$\vdots$				
a_1	a_1	a_1	$\ldots$	b_1	a_1	a_1	$\ldots$	a_1
a_1	a_1	a_1	$\ldots$	a_1	b_1	a_1	$\ldots$	a_1
a_2	a_2	a_2	$\ldots$	a_2	a_2	a_2	$\ldots$	a_2
b_2	a_2	a_2	$\ldots$	a_2	a_2	a_2	$\ldots$	a_2
a_2	b_2	a_2	$\ldots$	a_2	a_2	a_2	$\ldots$	a_2
				$\vdots$				
a_2	a_2	a_2	$\ldots$	b_2	a_2	a_2	$\ldots$	a_2
a_2	a_2	a_2	$\ldots$	a_2	a_2	b_2	$\ldots$	a_2

Für die so definierte Relation $\underline{R}^C$ gilt: K ist die Menge der minimalen Schlüssel von $\underline{R}^C$.

3.1.4 Funktionale Abhängigkeit

Definition: Seien $X, Y \subseteq R$. Dann besteht eine *funktionale Abhängigkeit*, in Zeichen $X \to Y$, falls die X-Werte die Y-Werte in Objekten determinieren.

Zudem schreibt man $\underline{R}^C \models X \to Y$, falls für alle $t, t' \in \underline{R}^C$ gilt, daß aus $t[X] = t'[X]$ auch $t[Y] = t'[Y]$ folgt[3].

Beispiel: Betrachte die folgende Relation Vorlesungsbesuch:

Kurs	Raum	Zeit	Semester	Lesender	Student	Note
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Dann bestehen folgende funktionale Abhängigkeiten:

- {Kurs, Student} $\to$ {Note}
- {Kurs, Semester, Zeit, Student} $\to$ {Lesender, Note, Raum}

Zudem ist {Kurs, Semester, Zeit, Student} ein minimaler Schlüssel.

Folgerung: Für $X, Y \subseteq R$ mit $X \cup Y = R$ gilt: Ist $X \to Y$, so ist X ein Schlüssel in R.

[3]Notation: $t[X] \hat{=} t|_X$

Wir geben jetzt ein **deduktives System** (Kalkül) für funktionale Abhängigkeiten an. Dazu gelte als Axiom $X \cup Y \rightarrow Y$. Als Regeln verwenden wir:

$$\frac{X \rightarrow Y}{X \cup Z \cup W \rightarrow Y \cup Z} \quad \text{sowie} \quad \frac{X \rightarrow Y, Y \rightarrow Z}{X \rightarrow Z}$$

Aus diesen Regeln kann man weitere Regeln herleiten, beispielsweise:

$$\frac{X \rightarrow Y, X \rightarrow Z}{X \rightarrow Y \cup Z} \quad \text{oder} \quad \frac{X \rightarrow Y \cup Z}{X \rightarrow Y}$$

Beweis der linken abgeleiteten Regel:

$$\frac{\dfrac{X \rightarrow Y}{X \rightarrow Y \cup X} \quad \dfrac{X \rightarrow Z}{X \cup Y \rightarrow Z \cup Y}}{X \rightarrow Y \cup Z}$$

Definitionen:

- Wir definieren eine *Ableitungsbeziehung:* Aus einer Menge Σ von funktionalen Abhängigkeiten $A \rightarrow B$ lassen sich durch das obige Kalkül funktionale Abhängigkeiten ableiten, in Zeichen: $\Sigma \vdash X \rightarrow Y$.
- Wir definieren folgende *Folgerungsbeziehung:* Aus einer Menge Σ von funktionalen Abhängigkeiten folgt $X \rightarrow Y$, falls in jeder Klasse, in der Σ gilt, auch $X \rightarrow Y$ gilt, in Zeichen: $\Sigma \models X \rightarrow Y$.

Satz: Sei Σ eine Menge von funktionalen Abhängigkeiten, $X, Y \subseteq R$ Mengen von Attributen R. Dann gilt:

$$\Sigma \models X \rightarrow Y \quad \Longleftrightarrow \quad \Sigma \vdash X \rightarrow Y$$

Damit ist das deduktive System vollständig und korrekt.

Beweis:

„$\Leftarrow$“ Die Regeln sind offensichtlich korrekt.

„$\Rightarrow$“ Durch Widerspruch. Angenommen, es existieren X und Y mit $\Sigma \nvdash X \rightarrow Y$, aber $\Sigma \models X \rightarrow Y$. Wir definieren nun die Menge aller aus X ableitbaren Attribute (Hülle) als

$$X^+ = \{ A \in R \mid \Sigma \vdash X \rightarrow \{A\} \}$$

Dann ist $Y \not\subseteq X^+$, d.h. es existiert $A \in Y \setminus X^+$, und es existieren Tupel t, t', so daß gilt:

	X^+	A	Rest
t	a	a	a
t'	a	b	b

Dies ist ein Widerspruch zu $\Sigma \models X \to Y$.

3.1.5 Mehrwertige Abhängigkeit

Definition: *Mehrwertige Abhängigkeiten*, in Zeichen $X \twoheadrightarrow Y$, werden durch folgende äquivalente Formulierungen definiert:

- Die X-Werte determinieren die *Menge* der Y-Werte.
- Für alle $t_1, t_2 \in \underline{R}^C$ mit $t_1[X] = t_2[X]$ existiert auch ein $t' \in \underline{R}^C$ mit

  $$t'[X \cup Y] = t_1[X \cup Y] \quad \text{und} \quad t'[X \cup (R \setminus Y)] = t_2[X \cup (R \setminus Y)].$$

  D.h. stimmen zwei Tupel über X überein, dann existiert ein drittes Tupel, das Werte vom ersten über $X \cup Y$ und vom zweiten über $R \setminus (X \cup Y)$ übernimmt.
- Die Komponenten von R lassen sich strukturieren in X, Y, Z, wobei Y und Z unabhängig voneinander sind bei Berücksichtigung von X.
- Zu jedem X-Wert kann zu jedem Z-Wert eine identische Menge Y-Werte existieren [Lin04].

Veranschaulichung mit $Z = R \setminus (X \cup Y)$:

R^C	X	$X \cap Y$	$Y \setminus X$	Z
t_1	a	a	a	a
t_2	a	a	b	b
t'	a	a	a	b

Für ein **Beispiel** siehe [Kel04].

Im **Kalkül** gelten folgende Regeln:

$$\frac{X \to Y}{X \twoheadrightarrow Y} \qquad \frac{X \twoheadrightarrow Y}{X \twoheadrightarrow Y \setminus X} \qquad \frac{X \twoheadrightarrow Y}{X \twoheadrightarrow (R \setminus Y)}$$

3.1.6 Inklusionsabhängigkeit

Definition: Eine *Inklusionsabhängigkeit* definiert die Gleichheit der Attribute auf X und Y, d.h. es gilt $\underline{R}[X] \subseteq \underline{S}[Y]$ in $\underline{R}^C$ bzw. $\underline{S}^C$, falls für alle $t \in \underline{R}^C$ auch $t[X] \in \underline{S}^C[Y]$ gilt.

Beispiel mit oben angegebenen Relationen:

- Vorlesungsbesuch[Student] $\subseteq$ Student[SNum]
- Vorlesungsbesuch[Kurs, Raum, Zeit, Semester, Lesender] $\subseteq$ Vorlesung[Kurs, Raum, Zeit, Semester, Lesender]

3.2 Relationale Algebra

Ziel: Wir wollen die Algebra der Operationen über einer Datenbank entwickeln. Dabei sollen die folgenden Beschränkungen gelten:

- Die Algebra muss *generisch*, d.h. automatisch generierbar über der Darstellung der Datenbank sein. Wir wollen, sobald wir das Schema wissen, die ganze Algebra daraus folgern können.
- Da der Computer endlich ist, betrachten wir nur *endliche* Strukturen, obwohl Basisdatentypen potentiell unendlich sein können.
- Wir betrachten vorläufig nur *Mengen* (SQL benutzt auch Multimengen).

Wir **erweitern** die Basisdatentypen um den Wert `NULL` und schreiben

$$\mathrm{DT}^{\texttt{NULL}} = (\mathrm{domain}(\mathrm{DT}) \cup \{\texttt{NULL}\}, \mathrm{op}^{\texttt{NULL}}(\mathrm{DT}), \mathrm{pred}^{\texttt{NULL}}(\mathrm{DT}))$$

Es soll dabei gelten $\texttt{NULL} \neq a$ für alle $a \in \mathrm{domain}(DT)$. Der Wert `NULL` ist semantisch überladen: die Berechnungen sind ggf. sinnlos. Bei Einführung von Operationen wird deswegen abgesprochen, ob sie mit `NULL` operieren können und was ihr Ergebnis in diesem Falle ist.

3.2.1 Typerhaltende Operationen

Gegeben sei ein Typ $\underline{R}$. Die Operanden und das Resultat von typerhaltenden Operationen sind Klassen über $\underline{R}$. Wir unterscheiden folgende Operationen:

Mengenoperationen: Für zwei Klassen $\underline{R}^{C_1}, \underline{R}^{C_2}$ betrachten wir

- die Vereinigung $\underline{R}^{C_1} \cup \underline{R}^{C_2}$ (mit Duplikateliminierung)

- den Durchschnitt $\underline{R}^{C_1} \cap \underline{R}^{C_2}$
- die Differenz $\underline{R}^{C_1} \setminus \underline{R}^{C_2}$ (alle Elemente aus der ersten Klasse, die nicht in der zweiten enthalten sind).
- das Komplement $\mathrm{dom}(R) \setminus \underline{R}^{C_1}$ – es wird allerdings nicht benutzt, weil es unsicher ist: Wenn man eine unendliche Wertemenge hat, bekommt man eine unendliche Menge als Ergebnis.

Selektion: Wir wollen alle Elemente der Klasse auswählen, die eine gegebene Bedingung erfüllen.

Definition: Seien $A \in R$ und $\Theta \in \{<, \leq, =, \geq, >, \neq\}$. Für $a \in \mathrm{domain}(\mathrm{dom}(A))$ und $B \in R$ definieren wir

$$\begin{aligned} \sigma_{A\,\Theta\,a}(\underline{R}^C) &:= \left\{ t \in \underline{R}^C \;\middle|\; t(A)\ \Theta\ a \right\} \\ \sigma_{A\,\Theta\,B}(\underline{R}^C) &:= \left\{ t \in \underline{R}^C \;\middle|\; t(A)\ \Theta\ t(B) \right\} \end{aligned}$$

Induktiv definieren wir weitere Operationen. Für Bedingungen α und β sei

$$\begin{aligned} \sigma_{\alpha\wedge\beta}(\underline{R}^C) &:= \sigma_\alpha(\underline{R}^C) \cap \sigma_\beta(\underline{R}^C) \\ \sigma_{\alpha\vee\beta}(\underline{R}^C) &:= \sigma_\alpha(\underline{R}^C) \cup \sigma_\beta(\underline{R}^C) \\ \sigma_{\neg\alpha}(\underline{R}^C) &:= \underline{R}^C \setminus \sigma_\alpha(\underline{R}^C) \end{aligned}$$

3.2.2 Typmodifizierende Operationen

Definition: *Umbenennung von Attributen*: Für $A \in R$ und $B \notin R$ sei $\varrho_{A\to B}(\underline{R}^C)$ die Klasse, in der das Attribut A in B „umbenannt“ wurde. Es soll dabei gelten $\mathrm{dom}_{\varrho_{A\to B}(R)}(B) = \mathrm{dom}(A)$.

3.2.3 Typgenerierende Operationen

Seien $\underline{R}$ und $\underline{S}$ Relationenschemata und $\underline{R}^C, \underline{S}^C$ entsprechende Klassen.

Definition: Sei $X \subseteq R$. Dann definieren wir die *Projektion* als

$$\pi_X(\underline{R}^C) = \underline{R}^C[X] := \left\{ t \mid \exists t' \in \underline{R}^C : t =_X t' \right\}$$

Unter $t =_X t'$ verstehen wir dabei: $\forall A \in X : t(A) = t'(A)$.

Definition: Sei der *natürliche Verbund* definiert durch

$$\underline{R}^C \bowtie \underline{S}^C := \left\{ t \mid \exists t' \in \underline{R}^C,\ t'' \in \underline{S}^C : t =_R t' \wedge t =_S t'' \right\}$$

Beispiel: Gegeben seien folgende Relationen Student und Vorlesungsbesuch:

SNum	Hauptfach	Nebenfach
007	Info	Psychologie
007	Info	Physik

Kurs	Raum	SNum
DB1	II	007
DB5	I	007
Compiler	III	4711

Das Ergebnis von Student $\bowtie$ Vorlesungsbesuch ist dann

SNum	Hauptfach	Nebenfach	Kurs	Raum
007	Info	Psychologie	DB1	II
007	Info	Physik	DB1	II
007	Info	Psychologie	DB5	I
007	Info	Physik	DB5	I

Das Ergebnis ist also definiert über den gemeinsamen Attributen SNum von Student und Vorlesungsbesuch. Für das Beispiel ergibt sich eine Relation mit fünf Attributen, da SNum nur einmal geführt ist.

Bemerkung: Man braucht nur solche t' und t'' zu verbinden, für die $t' =_{R \cap S} t''$ gilt (man sagt dazu: t' und t'' sind *verbundkompatibel*). Alle anderen sind nicht relevant und werden zu dem natürlichen Verbund nicht dazugenommen.

Eigenschaften des natürlichen Verbunds:

- Für $R \cap S = \emptyset$ ist $\underline{R}^C \bowtie \underline{S}^C = \underline{R}^C \times \underline{S}^C$.
- Für $R = S$ ist $\underline{R}^C \bowtie \underline{S}^C = \underline{R}^C \cap \underline{S}^C$
- Kommutativität: $\underline{R}^C \bowtie \underline{S}^C = \underline{S}^C \bowtie \underline{R}^C$
- Assoziativität: $\underline{R}^C \bowtie (\underline{S}^C \bowtie \underline{T}^C) = (\underline{R}^C \bowtie \underline{S}^C) \bowtie \underline{T}^C$
- Seien $X_1, \ldots, X_n \subseteq R$ mit $\bigcup_{i=1}^n X_i = R$. Dann gilt: $\underline{R}^C \subseteq \bowtie_{i=1}^n \pi_{X_i}(\underline{R}^C)$.

Als Beispiel dazu betrachte folgende Relationen:

PNum	SNum
0815	007
0815	007
0915	4711

SNum	Hauptfach
007	Info
007	Info
4711	Info

Hauptfach	Nebenfach
Info	Psychologie
Info	Physik
Info	BWL

Nebenfach	Betreuer
Psychologie	H
Physik	E
BWL	G

Der natürliche Verbund dieser Relationen ist

PNum	SNum	Hauptfach	Nebenfach	Betreuer
0815	007	Info	Psychologie	H
0815	007	Info	Physik	E
0815	007	Info	BWL	G
0915	4711	Info	Psychologie	H
0915	4711	Info	Physik	E
0915	4711	Info	BWL	G

Es ging nur negative Information verloren (d.h. die Information darüber, welcher Student welches Nebenfach *nicht* hat). Die positive Information blieb in der Tabelle enthalten.

Definition: Der *äußere Verbund* wird definiert als

$$\begin{aligned}&\{t \in \mathrm{dom}(R \cup S) \mid \exists t' \in \underline{R}^C : t =_R t' \wedge (\ \ (\exists t'' \in \underline{S}^C : t =_S t'') \\ &\qquad\qquad \vee((\nexists t'' \in \underline{S}^C : t' =_{R\cap S} t'') \wedge t[S \setminus R] = \texttt{NULL}))\} \\ \cup&\{t \in \mathrm{dom}(R \cup S) \mid \exists t'' \in \underline{S}^C : t =_S t'' \wedge (\ \ (\exists t' \in \underline{R}^C : t =_R t') \\ &\qquad\qquad \vee((\nexists t' \in \underline{R}^C : t' =_{R\cap S} t'') \wedge t[R \setminus S] = \texttt{NULL}))\}\end{aligned}$$

Die erste Menge heißt der *linke äußere Verbund* und die zweite der *rechte äußere Verbund*.

Weitere Operationen sind unter anderem

- Hier fehlt noch eine Definition für $\underline{R}^C \ {}_X\bowtie_Y \ \underline{S}^C$!
- Das *kartesische Produkt* $\underline{R}^C \times \underline{S}^C$. Im Unterschied zum natürlichen und äußeren Verbund werden hier die gleichen Attribute von $\underline{R}^C$ und $\underline{S}^C$ nicht zusammengefasst (z.B. Student $\times$ Vorlesungsbesuch hat sechs Attribute). Stattdessen hängt man an den Namen des Attributen den Namen der Tabelle vorne an (z.B. $a \to R.a$), um die Namenskollision zu vermeiden.

- *Theta-Verbund*: Für eine Bedingung α ist

$$\underline{R}^C \Theta_\alpha \underline{S}^C = \sigma_\alpha(\underline{R}^C \times \underline{S}^C)$$

- Division $\underline{R}^C / \underline{S}^C$. Sie enthält alle Tupel von $\underline{R}^C$, für die „nichts“ in $\underline{S}^C$ existiert: Ein Tupel t ist in $\underline{R}^C / \underline{S}^C$ enthalten, falls für jedes Tupel t_s aus $\underline{S}^C$ ein Tupel t_r aus $\underline{R}^C$ gibt, so gaß die beiden folgenden Bedingungen erfüllt sind:

$$t_r =_S t_s \text{ und } t_r[R \setminus S] = t$$

3.2.4 Abfragen mit der relationalen Algebra

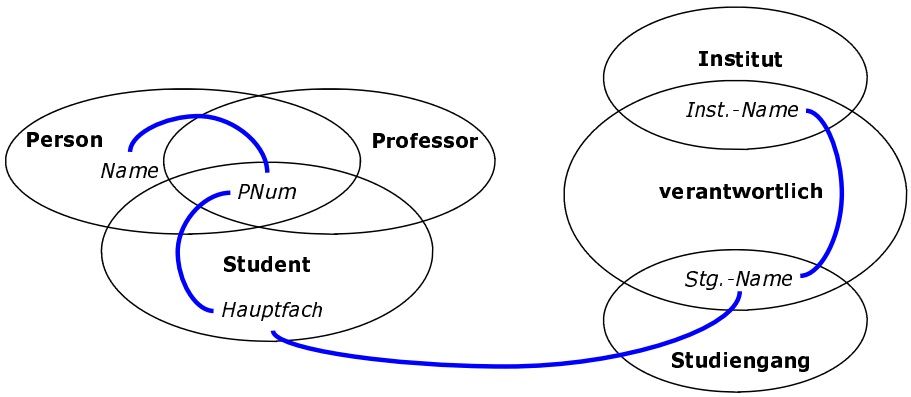
Nach der Definition der Algebra, z.B. mit den Operatoren π_x, σ_α, $\bowtie$, $\varrho_{A \to B}$, $\cup$, $\cap$, können nun **Ausdrücke** definiert werden.

Definition: Ausdrücke in der Relationalen Algebra sind definiert durch:

1. Jedes Relationenschema ist ein (elementarer) Ausdruck.
2. Für Ausdrücke e_1, e_2 sind auch $\pi_x(e_1)$, $\sigma_\alpha(e_1)$, $e_1 \bowtie e_2$, $\varrho_{A \to B}(e_1)$, $e_1 \cup e_2$ Ausdrücke.

Die Ausdrücke der Algebra können wiederum **visualisiert** werden.

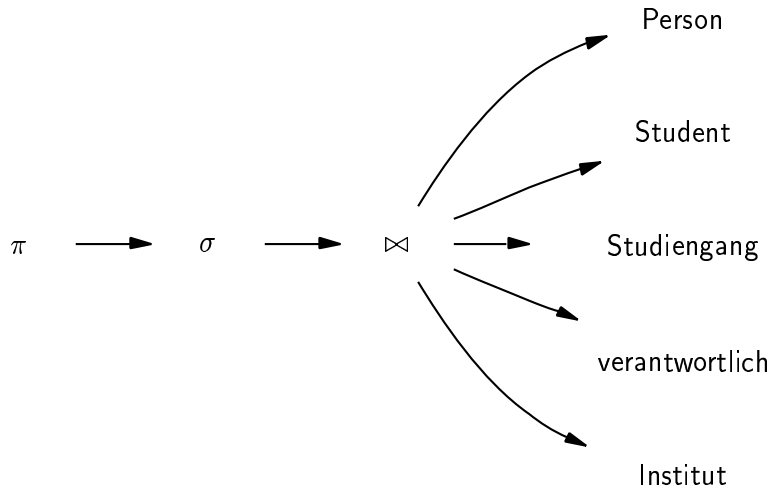
Beispiel: Seien folgende Relationen und die Anfrage „An welchem Institut studiert Alf?“ gegeben:



Dann läßt sich die Anfrage formal darstellen als:

$$\pi_{\mathsf{InstName}}(\sigma_{\mathsf{Name}=\text{„Alf“}}(\mathsf{Person} \bowtie \mathsf{Student}\, \Theta_{\mathsf{Hauptfach}=\mathsf{StgName}}\, \mathsf{Studiengang} \bowtie \mathsf{verantwortlich} \bowtie \mathsf{Institut}))$$

Dies läßt sich auch als Baum darstellen:



Gewünscht ist aber eine einfacherere Sprache zur Formulierung solcher Anfragen. Ein pragmatisches Herangehen zur Formulierung von Anfragen sähe folgendermaßen aus [Tha04]:

1. *Ergänzung* der Anfrageäußerung zu einer „Vollfrage“ (genau formulierte Anfrage) durch
   - Disambiguierung von Fragesätzen,
   - Ergänzung der Ellipsen zu vollständigen Sätzen,
   - Klärung, inwieweit eine Closed-World-Assumption oder eine partiell offene Datenwelt in der Datenbank unterlegt wird (Behandlung von Nullwerten) und
   - Schärfung der Formulierung von Aggregationsfunktionen;
2. *Reformulierung* der Anfrage in eine existentiell geprägte Form, damit sie mit den Mitteln der Technologie aufgelöst werden kann, wobei
   - nicht alle Generalisierungen aufgelöst werden müssen, sondern über `ALL` und `ANY` abgebildet werden können, und
   - ggf. auch besser überschaubare Boolesche Bedingungen erzeugt werden, indem z.B. die Negation möglichst weit zu den atomaren Formeln gezogen wird.
3. *Abbildung* der Anfragebegriffe auf das *Datenbank-Schema*, dabei ggf.
   - Spezifika der Schema-Definition beachten (Nullwerte und Default-Werte, die eine Anfrageberechnung verändern können),
   - Abkürzungen benutzen, falls möglich, und

- Schrittfolge zur Berechnung der Resultate durch eine Prozedur bereitstellen.

4. Abbildung der Resultatskonzepte auf *Antwortformen*.

Definition: Seien S, S' Schemata. Eine *Anfrage* der relationalen Algebra ist eine Abbildung q mit $q(\underline{S}^C) = \underline{S'}^C$.

Dabei gelten noch einige **Einschränkungen** für die Anfrage-Funktionen:

1. Die Abbildung sollte *typisiert* sein,
2. sie muß (bis auf in der Anfrage verwendete Konstanten) *isomorphietreu* sein,
3. sie muß *(effektiv?) berechenbar* sein, und
4. sie sollte *universiumstreu* sein (d.h. falls $\underline{S}^C$ über Domain $D \subseteq D'$ definiert ist, dann ist die Anfrage über Domain D und D' gleich).

3.2.5 Sichten

Relationale Sichten werden definiert durch ein Relationenschema V der Sicht und einer Anfrage über einem relationalen Datenbankschema [Tha04].

3.3 Relationaler Tupelkalkül (TRC)

Motivation [KE97]: Ausdrücke in der Relationenalgebra spezifizieren, wie das Ergebnis der Anfrage zu berechnen ist. Diese prozedurale Berechnungsvorschrift ergibt sich aus den Algebraoperatoren. Besonders deutlich wird das an der Operatorbaum-Darstellung, wo man sich veranschaulichen kann, daß die Zwischenergebnis-Tupel von unteren zu weiter oben angeordneten Operatoren weitergeleitet werden.

Demgegenüber ist der *Relationenkalkül* stärker deklarativ orientiert, d.h. es werden die qualifizierenden Ergebnistupel beschrieben, ohne daß eine Herleitungsvorschrift angegeben wird.

Formal [KE97]: Ein Audruck des Tupelkalküls besitzt die Form $\{t \mid \varphi(t)\}$, wobei t eine Tupelvariable ist und φ eine Formel über dem Datenbankschema. Oft spezifizieren wir die Attribute von t, die uns interessieren und schreiben $t_1.A_1, t_2.A_2, \ldots, t_n.A_n$.

Induktiver Aufbau der Formeln:

Definition: Die *TRC-Terme* sind:

- Konstanten (Zahlen oder Zeichenketten)
- Tupelvariablen
- Attributterme $t.A$ mit Tupelvariabler t und Attribut A

Definition:

1. Für $\Theta \in \{\leq, \geq, <, >, =, \neq\}$ sind folgendes atomare *TRC-Formeln*:
   - $R(s)$ für relationales Schema R und Tupelvariable s
   - $t_1 \,\Theta\, t_2$ für Terme t_1, t_2
2. Für Formeln α, β sind $\alpha \wedge \beta, \alpha \vee \beta, \neg\alpha$ Formeln.
3. Für eine Formel α und Tupelvariable t sind $\forall t\alpha$, $\exists t\alpha$ Formeln (o.B.d.A. t frei in α)

Bemerkung: Es sind Formeln verboten, wo ein Attribut nur negiert vorkommt – sonst ist das Ergebnis eine unendliche Menge.

Beobachtung: Beide Sprachen besitzen gleiche Ausdrucksstärke! Es gibt aber Einschränkungen dieser Sprachen, z.B. sind Hüllen nicht ausdrückbar, da bislang keine Rekursion möglich ist!

Zwei zusätzliche **Operationengruppen**:

- **Partitionierung** in Mengen anhand einer Separationsbedingung, z.B. Partitionierung aller Personen nach dem ersten Buchstaben des Namens;
- **Aggregatbildung** über die Gruppe: Wertermittlung für Minimum, Maximum, Summe, Anzahl, Durchschnitt – falls der Wertebereich diese Operation zuläßt

Beispiele: Betrachte folgendes Schema:

- Angestellter(Filiale, Nummer, Name, Gehalt, Abteilung, Geburtsjahr, Einstellungsdatum)
- Abteilung(Nummer, Name, Filiale, Stock, Leiter)
- Filiale(Nummer, Stadt, Land)
- Lieferant(Nummer, Name, Stadt, Land)
- Artikel(Nummer, Name, Abteilung, Preis, Bestand, Lieferant)

- Verkauf(Nummer, Datum, Abteilung, Artikel, Anzahl, Angestellter, Betrag)

Zu diesem Schema betrachte folgende Kalkül-Ausdrücke:

- Name aller Angestellten mit Gehalt von weniger als 400 Euro

```
{t.Name | Angestellter(t) ∧ t.Gehalt < 400 }
```

- Namen und Preise aller Artikel, die von einem Lieferanten aus Schleswig-Holstein geliefert werden

```
{t.Name, t.Preis | Artikel(t)
                 ∧ ∃ l (Lieferant(l)
                      ∧ t.Lieferant = l.Nummer
                      ∧ l.Land = 'SH')}
```

- Namen und Bestände aller Filialen in Berlin

```
{t.Name, t.Bestand | Artikel(t)
                   ∧ ∃ f (Filiale(f)
                        ∧ f.Stadt = 'Berlin'
                        ∧ ∃ v ∃ a (Verkauf(v)
                            ∧ Abteilung(a)
                            ∧ v.Abteilung = a.Nummer
                            ∧ a.Filiale = f.Nummer
                            ∧ v.Artikel = t.Nummer))}
```

Kalkül-Ausdrücke lassen sich auch durch Ausfüllen eines Formulars angeben:

Artikel	Name	Bestand	Nummer
t	✓	✓	

Filiale	Nummer	Stadt
f		'Berlin'

Verkauf	Artikel	Abteilung
v		t.Nummer

Abteilung	Nummer	Filiale
a	v.Abteilung	f.Nummer

- Alle Artikel, die in einer Abteilung verkauft wurden, deren Leiter „Helmut K. Raffke“ ist, die er aber nicht selbst verkauft hat

$$
\begin{aligned}
\{t \mid\ & \mathsf{Artikel}(t) \wedge \exists\ v\ \exists\ a\ \exists\ e\ \exists\ e2 \\
& (\mathsf{Verkauf}(v) \\
& \wedge\ \mathsf{Abteilung}(a) \\
& \wedge\ \mathsf{Angestellter}(e) \\
& \wedge\ \mathsf{Angestellter}(e2) \\
& \wedge\ t.\mathsf{Nummer} = v.\mathsf{Artikel} \\
& \wedge\ v.\mathsf{Abteilung} = a.\mathsf{Nummer} \\
& \wedge\ a.\mathsf{Leiter} = e.\mathsf{Nummer} \\
& \wedge\ e.\mathsf{Name} = \texttt{'Helmut}\,\sqcup\texttt{K.Raffke'} \\
& \wedge\ v.\mathsf{Angestellter} = e2.\mathsf{Nummer} \\
& \wedge\ e2.\mathsf{Nummer} \neq e.\mathsf{Nummer})\ \}
\end{aligned}
$$

3.4 Standard Query Language (SQL)

Die *Standard Query Language (SQL)* ist eine Sprache zur Verwaltung von Datenbanken, die von IBM Anfang der 70er Jahre vorgeschlagen und Anfang der 80er auf den Markt gebracht wurde. Seit 1987 gibt es einen ISO-Standard für SQL, der allerdings von keiner Implementierung vollständig unterstützt wird. Eine ausführlichere Beschreibung von SQL findet man in [Kle04].

3.4.1 Anfragen

Die prinzipielle Form von **Anfragen** ist die `SELECT`-**Anweisung**. Seien A_1, $\ldots$, A_n, B_1, $\ldots$, B_n Attribute, $\underline{R}_1^c, \ldots, \underline{R}_m^c$ Relationen (Tabellen) und α eine logische Bedingung. Dann sieht die `SELECT`-Anweisung (vereinfacht) wie folgt aus:

```
SELECT A_1 [AS B_1], ..., A_n [AS B_n]  // Zielliste
FROM R_1^c, ..., R_m^c                  // Argumentliste
WHERE α                                 // Bedingung
```

Die Semantik dieser Anweisung ist

$$\varrho_{A_1 \to B_1, \ldots, A_n \to B_n}(\pi_{A_1,\ldots,A_n}(\sigma_\alpha(\underline{R}_1^c \times \ldots \times \underline{R}_m^c))),$$

wobei die Operationen so neu definiert sind, dass Duplikate aus dem Ergebnis nicht entfernt werden (wie es z.B. bei π sonst der Fall wäre). Um das Löschen von Duplikaten zu erzwingen, kann man `SELECT DISTINCT` verwenden.

Falls die Namen der Attribute nicht eindeutig sind (z.B. falls das gleiche Attribut in mehreren Tabellen vorkommt), so schreibt man diese Namen als Tabellenname.Attributname.

Beispiel: Seien folgende Tabellen gegeben:

Customer(Cno, Name, City)
Order(Cno, Depname, Ino)

Eine mögliche SELECT-Anweisung wäre dann

```
SELECT Name AS CustomerName, City, Ino
FROM Customer, Order
WHERE City = 'Kiel' AND Customer.Cno = Order.Cno
```

Bei Anfragen mit mehreren Tabellen wird eine eindeutige Zuordnung der Vorkommen von Attributen durch die Verwendung von Tabellennamen oder Variablen erreicht, zusammen mit der Punktnotation von Attributen wie oben. Variablen sind dabei notwendig, falls mehrere Kopien einer Tabelle benötigt werden. Falls beispielsweise die Tabelle Motherhood(Mother, Child) gegeben ist, so kann folgende Anweisung mit Variaben x und y verwendet werden:

```
SELECT x.Mother AS Grandmother,
       y.Child AS Grandchild
FROM Motherhood x, Motherhood y
WHERE x.Child = y.Mother;
```

Bemerkungen:

- Man kann * in einer Zielliste verwenden, um alle Attribute auszuwählen. Mit * sollte man nicht programmieren: wenn man die Relation verändert, ändert sich auch der Rückgabewert von `SELECT *`.
- In einer Zielliste sind neben den Attributnamen auch andere Ausdrücke erlaubt, z.B.

```
SELECT 2 + 2 AS '2 + 2'
```

3.4.2 Bedingungen

In den Bedingungen wird **dreiwertige Logik** verwendet. Neben `TRUE` und `FALSE` benutzt man den Wert `UNKNOWN`, der das Ergebnis aller Vergleiche mit `NULL` ist. Die logischen Operationen sind wie folgt definiert:

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

Beim Ausführen einer Anfrage, werden nur die Tupel zum Ergebnis hinzugefügt, bei den die Bedingung zu `TRUE` ausgewertet wird. Man soll sehr vorsichtig

sein, da viele Tautologien in der Booleschen Logik keine Tautologien in der dreiwertigen Logik sind.

Beispiel: Bei der Belegung $B = \texttt{NULL}, A = 1, C = 2$ gilt:

$$(A = B \ \wedge \ B = C) \rightsquigarrow \texttt{UNKNOWN}$$
$$(A = B \ \wedge \ A = C) \rightsquigarrow \texttt{FALSE}$$

Weiteres zu Bedingungen:

- Man kann den Wert eines Booleschen Ausdrucks α mit dem Prädikat `IS [NOT]` α testen.
- Das Prädikat `IS [NOT] NULL` erlaubt zu überprüfen, ob der Wert eines Ausdrucks `NULL` ist.
- Beim Vergleich von Zeilen kann man Muster (Patterns) benutzen mit Hilfe der Operatoren `LIKE` und `SIMILAR`.

Zudem können Mengen in speziellen Prädikaten benutzt werden. Sei dazu M eine (Multi-)Menge, nun sind Anfragen wie $x \in M$, $x \notin M$, $M = \emptyset$, $M \neq \emptyset$ möglich. Dazu benutzt man Teilanfragen: Das Ergebnis einer `SELECT`-Anfrage kann in der `WHERE`-Klausel einer anderen Anfrage verwendet werden.

Beispiel: Die Teilanfrage gibt ein einziges, einstelliges Tupel zurück:

```
SELECT Mother AS GrandmaOfCharles
FROM Motherhood
WHERE Child = (SELECT Mother
               FROM Motherhood
               WHERE Child = 'Charles');
```

Beachte dazu:

- Teilanfragen müssen geklammert werden.
- Falls die Teilanfrage mehr als ein Tupel liefert, ergibt sich hier ein Fehler.
- Es gibt eine Bereichsregel (scoping rule): Ein Vorkommen eines Attributes bezieht sich auf das gemäß der Schachtelung nächste Vorkommen eines Tabellennamens mit diesem Attribut, das Teil einer `FROM`-Klausel ist und dem – falls angegeben – seine Variable zugeordnet ist.

Weitere Prädikate sind etwa `[NOT] IN`, `EXISTS`, `ANY`, `ALL`.

Beachte:

- Das Ergebnis von `EXISTS` ist wahr, falls das Ergebnis der Teilanfrage nicht leer ist, sonst falsch. `UNKNOWN` ist nicht möglich!
- Die Quantoren `ANY` und `ALL` werden durch Disjunktion bzw. Konjunktion ausgewertet (endlicher Wertebereich!).
- Zur Behandlung von `UNKNOWN`-Werten und leeren Abfragen siehe auch [Kle04].

Beispiele:

```
SELECT Child
FROM Motherhood
WHERE Mother IN (SELECT Child
                 FROM Motherhood
                 WHERE Mother = 'Elizabeth');

SELECT *
FROM Item
WHERE Price >= ALL (SELECT Price FROM Item);

SELECT *
FROM Item
WHERE Price >  ANY (SELECT Price FROM Item);
```

Beachte auch folgendes Beispiel: Es sei eine Tabelle NULLWERTE mit den Werten `10` und `NULL` gegeben. Dann ergibt folgendende Anfrage nur `10` als Antwort, da das `IN`-Konstrukt durch `= ANY` übersetzt wird und `NULL= NULL` nur `UNKNOWN` ergibt:

```
SELECT Wert
FROM Nullwerte
WHERE Wert IN (SELECT Wert FROM Nullwerte)
```

Aus **Hamlet** kennen wir „*To be or not to be!*“. Aber in SQL gilt: „*Maybe and maybe may be not maybe!*“ Sei eine Relation R gegeben mit $R.A = 1$ und $R.B =$ `NULL`. Betrachte dann die folgende Abfrage, die `1` liefert (!):

```
SELECT x.A
FROM R x
WHERE NOT EXISTS (SELECT * FROM R y WHERE y.B =  1)
  AND NOT EXISTS (SELECT * FROM R y WHERE y.B <> 1)
```

3.4.3 Mengenoperationen

Vorhandene Mengenoperationen sind Vereinigung (`UNION`), Schnitt (`INTERSECT`) und Differenz (`EXCEPT` oder `MINUS`). Dabei müssen die Argumente dieser Operatoren *`UNION`-kompatibel* sein (d.h. die Attribute müssen entsprechend übereinstimmen).

Semantik: Im Unterschied zu `SELECT`-Operationen, die die Multimengen-Semantik (*bag*) verwenden, wird bei den Mengenoperationen als Default die Mengen-Semantik (*set*) benutzt. **Begründung**: Durchschnit und Löschen bei Bag-Semantik ist teuer. Allerdings kann Bag-Semantik durch Verwendung von `ALL` nach `UNION` etc. erzwungen werden.

Achtung: Falls man Mengenoperationen mit Multimengen-Semantik benutzt, gelten viele vertrauten Regeln nicht mehr, u.a. gilt zum Beispiel folgende Gleichheit auf Mengen, nicht aber auf Multimengen:

$$(R \cap S) \setminus T = R \cap (S \setminus T)$$

Die **Nullwerte** werden bei Mengenoperationen anders behandelt als in Booleschen Ausdrücken. Laut dem Standard werden zwei `NULL`-Werte nicht als *distinct* bezeichnet. Es gilt deswegen

```
R INTERSECT S ≠ SELECT x.*
                FROM R x, S y
                WHERE x.A = y.A AND
                      x.B = y.B ...
```

3.4.4 Aggregation

In einer `SELECT`-Klausel können die Funktionen `SUM`, `AVG`, `MIN`, `MAX` und `COUNT` auf Wertausdrücke angewandt werden. Bei Booleschen Wertausdrücken sind auch `EVERY`, `ANY` (`SOME`) erlaubt. Man kann `DISTINCT` verwenden, um Duplikate vor der Anwendung der Aggregation zu entfernen.

Beispiel: `COUNT` liefert die Anzahl der Tupel im Ergebnis.

```
SELECT COUNT(DISTINCT City)
FROM Customer
```

Problem: Die Syntax von SQL erlaubt nicht, mehrere Funktionenaufrufe hintereinanderzuschalten.

Die `GROUP BY`-Klausel wird verwendet, um die Ergebnisse von `SELECT` gemäß den Werten einer angegebenen Liste von Attributen zu gruppieren. Dann werden die Aggregationen pro Gruppe durchgeführt.

Eine `HAVING`-Klausel benutzt man, um Auswahlbedingungen für Gruppen entsprechend der `WHERE`-Klausel für `FROM` zu stellen.

Beispiele: Betrachte die folgende Tabelle Item:

Ino	Description	Price
...	doll	84.99
	doll	61.99
...	skirt	40.50
	skirt	59.50
...	skirt	53.00

Die Anfrage

```
SELECT Description, SUM(Price)
FROM Item
GROUP BY Description
```

liefert das Ergebnis

Description	Price
doll	146.98
skirt	153.00

Die Anfrage

```
SELECT Description, SUM(Price)
FROM Item
GROUP BY Description
HAVING COUNT(*) > 2
```

liefert das Ergebnis

Description	Price
skirt	153.00

3.4.5 Join-Operationen

SQL unterstützt folgende Join-Operationen:[4]

- *Old-Style Join.* Geschieht mit Angabe der Bedingungen in der `WHERE`-Klausel. Falls keine Bedingung angegeben wird, wird das Kreuzprodukt gebildet:

[4]Man beachte: Die Join-Operationen sind in SQL nicht kommutativ und links-assoziativ.

```
SELECT *
FROM X, Y
WHERE X.a = Y.a
```

- *Inner Join* – verschiedene Formen davon, darunter
  - *Natural Join*: Entspricht der $\bowtie$-Operation, nur ohne Mengenbildung

    ```
    SELECT * FROM X NATURAL JOIN Y
    ```

  - *Conditional Join*: Entspricht der Kreuzproduktbildung mit anschließender Selektion.

    ```
    SELECT * FROM X JOIN Y ON X.a = Y.a
    ```

- *Outer Join*. Es gibt *left* bzw *right outer join* und *full outer join*. Auch hier ist die Angabe von Bedingungen mit `ON` möglich

  ```
  SELECT * FROM X LEFT OUTER JOIN Y ON X.a = Y.a
  ```

- *Cross Join*. Entspricht `SELECT * FROM X, Y`

  ```
  SELECT * FROM X CROSS JOIN Y
  ```

- *Union Join*. Entspricht `SELECT * FROM X OUTER JOIN Y ON 1 = 2`

  ```
  SELECT * FROM X UNION JOIN Y
  ```

Beim `CROSS JOIN` werden die Spalten im Unterschied zu relationaler Algebra nicht unbenannt. Das Ergebnis einer Anfrage kann also durchaus eine Tabelle sein, die zwei Spalten mit gleichem Namen besitzt.

Die `ORDER BY`-Klausel kann verwendet werden, um die Ergebnistabelle zu ordnen.

3.4.6 Sichten (Views)

In SQL ist **Sicht** ein Ausdruck, der eine Tabelle beschreibt, ohne sie zu erzeugen. Die Syntax ist

`CREATE VIEW` *Name* `[(` *Liste von Spaltennamen* `)]` `AS` *query*

Beispiel:

```
CREATE VIEW ItemsOnSale AS
SELECT Ino, Price
FROM Item
WHERE EXISTS (SELECT *
              FROM Order
              WHERE Order.Ino = Item.Ino)
```

Die Sicht-Tabelle ItemsOnSale kann nun in den Anfragen, wie ein Basis-Datentyp verwendet werden:

```
SELECT Ino FROM ItemsOnSale WHERE Price > 50
```

Auswertung: Ersetze den Sichtenbezeichner durch die Anfrage der Sichtendefinition (mit geeigneten Anpassungen). Der Einsetzungsprozess muss immer zu einer syntaktisch korrekten SQL-Anfrage führen können.

Bemerkung: Es gibt auch *materialisierte Sichten*, wo die Daten aus der Datenbank extrahiert und getrennt gespeichert werden.

Änderungsoperationen sind auf Sichten problematisch, wenn die erforderlichen Änderungen in den Basistabellen nicht eindeutig bestimmt sind. Aus diesem Grund unterscheidet der SQL-Standard die Tabellen danach, ob sie *updatable* sind - dazu mehr in [Kle04].

Beispiel: Betrachte die folgenden Tabellen R und S

A	B
1	2
3	2

B	C
2	1
2	4

Das Ergebnis von $R \bowtie S$ ist

A	B	C
1	2	1
3	2	1
1	2	4
3	2	4

Aus dieser Tabelle ist es nicht möglich, den ersten Tupel zu entfernen ohne Veränderungen der Tabellen R und S.

Mit `WITH CHECK OPTION` kann veranlasst werden, dass über eine Sicht vorgenommene **Änderungen** die **Bedingung** in der `WHERE`-Klausel der Sicht **erfüllen**. Weiter gibt es die Formen `WITH LOCAL CHECK OPTION` und `WITH CASCADED CHECK OPTION`, in Abhängigkeit davon, ob weitere Sichten, die über der gegebenen Sicht definiert sind, auch ihre Check Options übernehmen sollen. Standard ist hier `CASCADED`.

Beispiel:

```
CREATE VIEW CHEAP_ITEMS AS
SELECT *
FROM ITEM
WHERE Price < 10.00
WITH LOCAL CHECK OPTION
```

3.4.7 Definition eines Datenbankschemas

Neue Tabellen werden erzeugt mit Hilfe von

`CREATE TABLE` *Name* (*Liste der Elemente*)

Die Elementliste kann enthalten:

- Attribute und ihre Typen, evtl. mit Modifikatoren, z.B. `NOT NULL`.
- Schlüssel: `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY`. Der Unterschied zwischen `PRIMARY KEY` und `UNIQUE` besteht darin, dass im ersten Fall keine Nullwerte erlaubt sind, aber im zweiten. Falls der Schlüssel aus nur einem Attribut besteht, so kann man ihn als Modifikator gleich nach dem Attributtyp deklarieren. Fremdschlüssel sind Schlüssel, die eine andere Tabelle referenzieren – mehr zur Syntax in [Kle04]
- Integritätsbedingungen: `CHECK (`*search condition*`)`

Mit Hilfe von `DROP TABLE` *Name* kann man eine Tabelle **löschen**.

Beispiele:

- Tabelle mit einem primären Schlüssel:

```
CREATE TABLE Item(
      Ino CHAR(5) PRIMARY KEY,
      Descr VARCHAR(30),
      Price DECIMAL(5, 2) NOT NULL);
DROP TABLE Item;
```

- Tabelle mit einem Fremdschlüssel:

```
CREATE TABLE Unterbringung(
      Unterkunft CHAR(10),
      Kategorie INTEGER,
      PRIMARY KEY(Unterkunft, Kategorie),
      FOREIGN KEY (Unterkunft)
            REFERENCES Hotel(Hotelid));
```

Hier ist vorausgesetzt, dass eine Tabelle Hotel(Hotelid, ...) existiert.

- Tabelle mit einer Integritätsbedingung:

```
CREATE TABLE CheapItem(
    ...
    Price DECIMAL(5, 2),
    CHECK(Price < 100.0));
```

3.4.8 Weitere Operationen

- Das **Einfügen** der Daten erfolgt mit Hilfe der `INSERT`-Anweisungen:

  `INSERT INTO` *table* `[(`$A_1, A_2, \ldots$`)]`
  `VALUES` $(exp_1, exp_2, \ldots)$

  Beispiel:

```
INSERT INTO Buch (Autor, Titel)
VALUES ('Biskup', 'Informationssysteme')
```

- Das **Löschen** der Daten erfolgt mit Hilfe der `DELETE`-Anweisungen:

  `DELETE FROM` *table* `[WHERE` *predicates*`]`

  Beispiel:

```
DELETE FROM Order WHERE Depname = 'Toys'
```

- Zum **Ändern** der Daten verwende die `UPDATE`-Anweisung:

  `UPDATE` *table*
  `SET` $A_1 = exp_1, A_2 = exp_2, \ldots$
  `[WHERE` *predicates*`]`

3.4.9 Rekursion

Problem: Prädikatenlogik erlaubt es nicht, die transitive Hülle einer zweistelligen Relation zu beschreiben. Als Beispiel, betrachte die folgende Tabelle, die die zweistellige Relation Motherhood beschreibt:

Mother	Child
QueenMum	Elizabeth
Elizabeth	Charles
Elizabeth	Anne
Anne	Zara

Es ist zwar möglich, mit Hilfe von SQL zu beantworten, wer die Mutter, die Großmutter usw. von Zara ist, allerdings ist es mit bisherigen Mitteln unmöglich, eine Anfrage zu schreiben, die in einer *beliebigen* Tabelle des Typs Motherhood *alle* Vorfahren ausgibt! Als Lösung bietet SQL **rekursive Definitionen** an mit folgendem Syntax:

```
WITH RECURSIVE Tabellenname(Attributliste)
((initial subquery)
 UNION ALL
 (recursive subquery))
```

Als Beispiel betrachte die folgende Anfrage, die alle Vorfahren von Zara in einer beliebigen Tabelle des Typs `Motherhood` liefert:

```
WITH RECURSIVE Ancestors(Mother)
((SELECT Mother
  FROM Motherhood
  WHERE Child = 'Zara')
 UNION ALL
 (SELECT x.Mother
  FROM Motherhood x, Ancestors y
  WHERE x.Child = y.Mother));

SELECT Mother
FROM Ancestors;
```

Bei der Auswertung wird zuerst mit Hilfe von *initial subquery* die Tabelle Ancestors gebildet und dann mit mehrfacher Anwendung von *recursive subquery* so lange erweitert, bis das Ergebnis der Query bei einem Schritt leer ist. Es wird also die Vereinigung folgender Anfragenergebnisse gebildet:

- $\mathsf{Hülle}_1$:

  $$\mathsf{Motherhood}[\mathsf{Child} = \text{'Zara'}][\mathsf{Mother}][\mathsf{Mother} \to \mathsf{Child}] \rightsquigarrow \{(\mathsf{Anne})\}$$

- $\mathsf{Hülle}_2$:

  $$(\mathsf{Hülle}_1 \bowtie \mathsf{Motherhood})[\mathsf{Mother}][\mathsf{Mother} \to \mathsf{Child}] \rightsquigarrow \{(\mathsf{Elisabeth})\}$$

- ...

Als ein weiteres Beispiel betrachte die folgende Tabelle, die Flugverbindungen darstellt:

Flug	Abflughafen	Zielflughafen	Kosten
LH377	FRA	JFK	700
BA170	LHR	JFK	400
LH122	HAM	FRA	100
BA730	FRA	LHR	150

Mit einer rekursiven Definition kann eine Tabelle Reisen berechnet werden, die alle möglichen Ziele samt Routen und Gesamtkosten, die von Hamburg (HAM) erreichbar sind, enthält (für SQL-Definition siehe [Kle04]):

Zielflughafen	Route	Gesamtkosten
FRA	„FRA“	100
JFK	„FRA, JFK“	800
LHR	„FRA, LHR“	250
JFK	„FRA, LHR, JFK“	650

3.4.10 Bemerkungen

Don Chamberlin [Cha98]:

> *Faced with these unpalatable options, the designers of SQL fell back to the principle "trust the user". Providing support for explicit nulls and threevalued logic in the database system gives users the tools to represent missing data with minimal cost and without requiring defensive coding in every application. At the same time, supporting the NOT NULL constraint at the column level allows users to avoid the anomalies associated with the null values where this is considered important. Since none of the options for representing missing information is flawless, and since users are paying the bills, it seems appropriate that users should be able to choose the appropriate approach they find least troublesome.*

Belnap und Steele [BS76]:

> *The meaning of a question addressed to a query system is not to be identified with how the system processes the query (and is not to be identified with a program at any level), but rather is to be identified with the range of answers that the question permits. That is, for a query system and a user to agree on the meaning of a question is for there to be agreement as to what counts as an answer to the question, regardless of how, or if, any answer is produced.*

4 Datenbank-Modellierung

Problem: Das relationale Modell hat Grenzen bezüglich der Natürlichkeit, u.a. bei komplexen Eigenschaften (z.B. Adreßdaten, Namen von Personen) die nicht mit den atomaren Typen dargestellt werden können, und bei der Semantikdarstellung über Integritätsbedingungen (insbesondere referentielle Integrität).

Gesucht ist eine adäquate, abstrakte Sprache – bei DBMS hat es in diesem Bereich eine Entwicklung gegeben hin zu objektrelationalen DBMS (mit komplexen Datentypen und komplexen Operationen). Anforderungen:

- *Abstraktionsinvarianz* (Analyse einer Anwendung, konzeptionelle Darstellung, Umsetzbarkeit in logische Sprachen)
- *Integrierte Darstellung* von Strukturierung und Funktionalität

4.1 Das (erweiterte) Entity-Relationship-Modell

Die Sprache besteht in diesem Fall aus einer abstrakten Notation, die eine Visualisierung bietet – die Sprache wird induktiv aufgebaut mit Modellierungsannahmen.

Definition: *Attribute* seien folgendermaßen induktiv definiert:

1. Alle $A \in U$ (universeller Namensraum) sind atomare Attribute mit $\mathrm{dom}(A) = DT \in \mathcal{LT}$.
2. Es seien $A_1, \ldots, A_n$ Attribute, B ein Name. Dann seien auch folgendes Attribute:
   - $B\{A_1\}$ (Mengenkonstruktor) mit $\mathrm{dom}(B\{A_1\}) = 2^{\mathrm{dom}(A_1)}_{\mathrm{endl.}}$
   - $B\,\langle A_1\rangle$ (Listenkonstruktor) mit $\mathrm{dom}(B\,\langle A_1\rangle) = \mathrm{list}_{\mathrm{endl.}}(\mathrm{dom}\,A_1)$
   - $B(A_1, \ldots, A_n)$ (Tupelkonstruktor) mit $\mathrm{dom}(B(A_1, \ldots, A_n)) = \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$
   - Weitere Konstruktoren können z.B. Multimenge $\{\{\cdot\}\}$, optionale Komponenten $[\cdot]$, Vereinigung $(\cdot \cup \cdot)$ etc. sein.

Gleichzeitig mit Operationen auf Attributen sind auch Auswahl-Funktionen sowie Prädikate $=$, $\neq$ etc. zu definieren.

Beispiele:

- Adressen mit optionalen Informationen:

$$\mathsf{Adresse}(\mathsf{PLZ}, \mathsf{Ort}, \mathsf{Straße}, \mathsf{Hausnummer}[, \mathsf{Etage}[, \mathsf{Richtung}]])$$

- Beschreibung von Namen:

$$\mathsf{Name}(\mathsf{Vornamen}\ \langle \mathsf{Vorname} \rangle, \mathsf{Familienname}, [\mathsf{Geburtsname},] \\ \mathsf{Titel}(\mathsf{AkademischeTitel}\{\mathsf{AkademischerTitel}\} \cup \mathsf{Familientitel}))$$

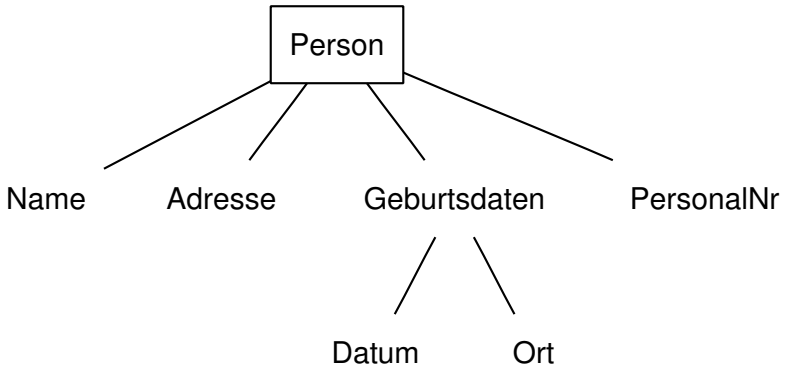
Wir definieren nun formal:

Definition: Ein Entity-Typ sei E = (Menge von Attributen, Identifikation, Integritätsbedingungen).

Beispiel:

$$\mathsf{Person} = (\{\mathsf{Name}, \mathsf{Adresse}, \mathsf{Geburtsdaten}(\mathsf{Datum}, \mathsf{Ort}), \mathsf{PersonalNr}\}, \\ \{\mathsf{Name}, \mathsf{Geburtsdatum}\}, \emptyset)$$

Die entsprechende **graphische Notation**:

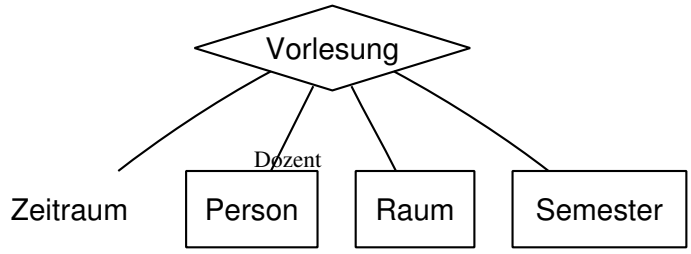


Definition: Ein Relationship-Typ sei R = ([Rolle:]Entity-Typ, ..., [Rolle:]Entity-Typ, {Attribute}, Integritätsbedingungen)

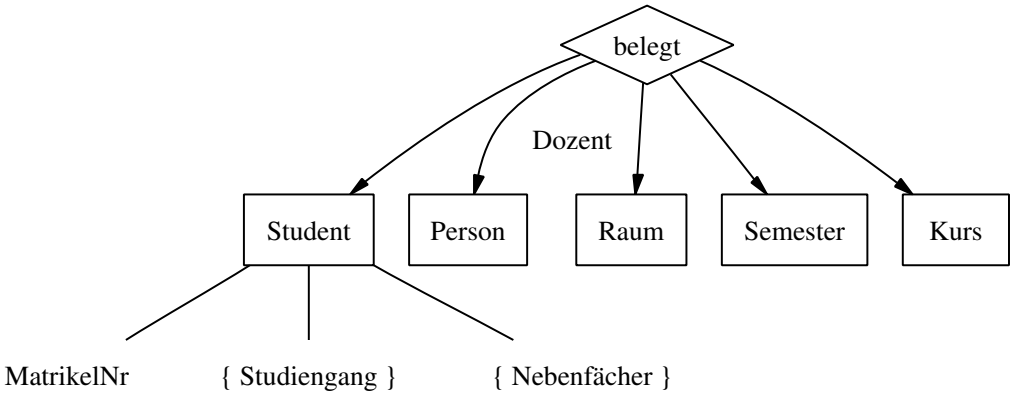
Beispiel:

$$\mathsf{Vorlesung} = (\mathsf{Dozent}\text{: }\mathsf{Person}, \mathsf{Raum}, \mathsf{Semester}, \{\ \mathsf{Zeitraum}\ \}, \emptyset).$$

Auch hier die **graphische Notation**:



Größeres **Beispiel:**



Nebenbedingung wäre zum Beispiel

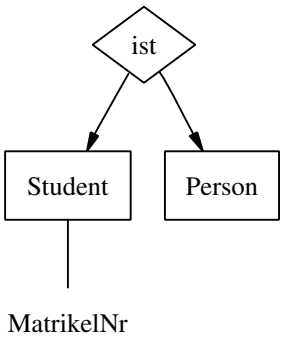
belegt[D, R, S, K] $\subseteq$Vorlesung[D, R, S, K]

Definition: Ein Relationship-Typ i-ter Ordnung sei $R = ($[Rolle:]Typ, ..., [Rolle:]Typ, {Attribute}, Integritätsbedingungen), wobei die Typen jeweils höchstens Ordnung $i-1$ haben und Entity-Typen Ordnung 0 haben.

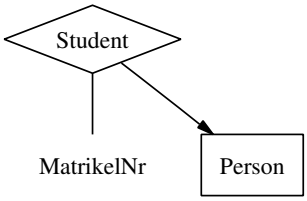
Definitionen:

- Als *Entity-Klasse* E^C bezeichnet man zu einem Entity-Typ $E = (\{A_1, \ldots, A_n\}, ID, JC)$ eine Mengen von Objekten über $\mathrm{dom}(E) \widehat{=} \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$.
- Eine *Relationship-Klasse* R^C für $R = (R_1, \ldots, R_n, \{A_1, \ldots, A_n\}, JC)$ ist eine endliche Menge über $\mathrm{dom}(R) \widehat{=} \mathrm{dom}(ID(R_1)) \times \ldots \times \mathrm{dom}(ID(R_n)) \times \mathrm{dom}(\{A_1, \ldots, A_n\})$.

Sonderfall: Betrachte folgendes Schema mit zwei Entity-Typen und der Einschränkung, daß jeder Student genau einer Person zugeordnet ist, aber Personen keine Studenten sein müssen:



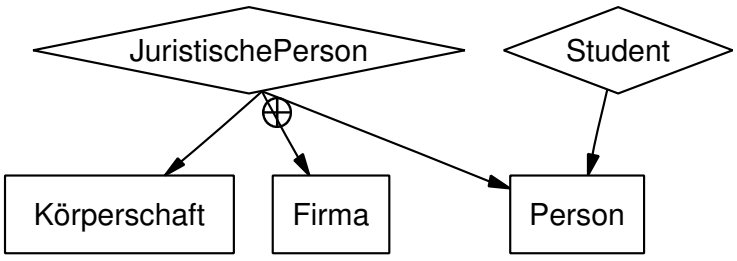
Eine andere Möglichkeit der Modellierung wäre, einen Entity-Typen und einen Relationship-Typen zu verwenden:



Weiterer Fall: Falls beispielsweise eine juristische Person definiert werden soll als entweder eine natürliche Person oder eine Körperschaft oder eine Firma, so verwende folgende Definition, um diese Form der Generalisierung zu trennen von einer Spezialisierung über Relationship-Typen (ist-ein-. . .-Relationen bzw. unäre relationale Typen):

Definition: Ein Cluster-Typ sei $C = (\oplus(R_1, \ldots, R_n), \{A_1, \ldots, A_n\}, IC)$.

Auch hier die **graphische** Notation:



Definition: Ein *Entity-Relationship-Schema* besteht aus einer strukturell abgeschlossenen Menge von Entity-, Relationship-, Attribut- und Cluster-Typen.

Dabei sind die Objektwerte strikt typenbezogen – oder (XML-artig) entfaltet über dem Schema (hoch redundant!).

Wir machen folgende **Annahmen** an unsere Spezifikationssprache:

1. Entity-, Relationship- und Cluster-Typen haben haben mindestens einen Attributtyp.
2. Attribute sind ihrem Typen zugeordnet (d.h. eine lokale Definition von Attributen ist möglich, es wird nicht eine global gültige Zuordnung von Attribut zu Typ verlangt).
3. Datentypen sind Attribut-Typen zugeordnet.
4. Typstruktur ist strikt hierarchisch – hierdurch können wir eine generische Definition der Operationen (Gleichheit etc.) verwenden!
5. Rigide Identifikation.

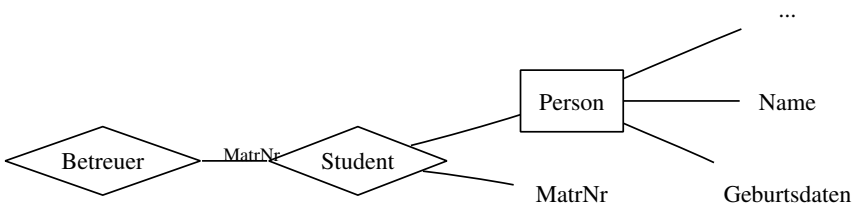
4.2 Integritätsbedingungen

Statische Integritätsbedingungen müssen in jedem Datenbank-Zustand gelten, **dynamische** Integritätsbedingungen werden später behandelt (dort aber nur Transitionsbedingungen, d.h. wenn in in einer Datenbank α gilt, dann soll in nächsten Zustand β gelten). Auch die statischen Integritätsbedingungen sind weit umfangreicher als der hier gezeigte Auszug.

Anliegen der **Modellierung** ist neben der Separation von Gesichtspunkten und der Separation von Spezialisierungen auch die Pflege der Datenzusammenhänge und numerische Einschränkungen.

Betrachte folgende Klassen von Integritätsbedingungen:

- *Schlüsselabhängigkeiten*, insbesondere minimale Schlüssel – beachte auch Vererbungsmechanismen für Typen, die den gegebenen Typen verwenden, Beispiel:



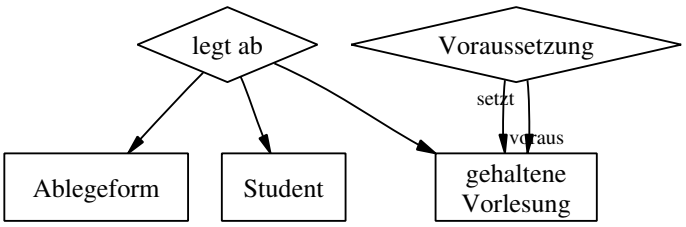
- *Funktionale Abhängigkeiten* (nunmehr für Komponenten eines Typs) am Beispiel Volesung:

$$\begin{aligned} \{\mathsf{Raum},\ \mathsf{Semester},\ \mathsf{Zeit}\} &\rightarrow \{\mathsf{Kurs},\ \mathsf{Dozent}\} \\ \{\mathsf{Dozent},\ \mathsf{Semester},\ \mathsf{Zeit}\} &\rightarrow \{\mathsf{Raum}\} \end{aligned}$$

- *Kardinalitätsbeschränkungen*: Gegeben sei ein Typ R und Teilkomponenten R'. Dann gilt $\mathrm{card}(R, R') = (m, n)$ in einer Klasse R^C, falls für alle $o' \in R'^C$ gilt:

$$m \leq \left|\left\{ o \in R^C \;\middle|\; o =_{R'} o' \right\}\right| \leq n$$

Betrachte dazu das Beispiel:



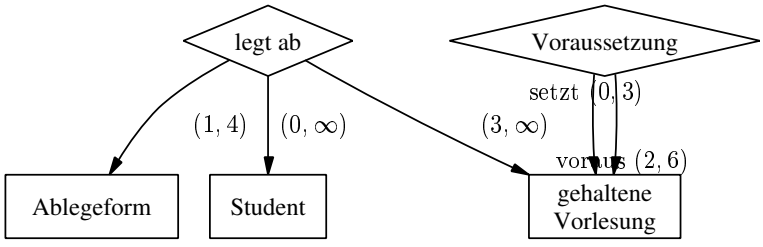
Man könnte jetzt folgende Beschränkungen einführen:

$$\begin{aligned} \mathrm{card}(\mathrm{Voraussetzung}, \mathrm{setzt}) &= (0, 3) \\ \mathrm{card}(\mathrm{Voraussetzung}, \mathrm{voraus}) &= (2, 6) \end{aligned}$$

Dabei sind überlappende Kardinalitätsintervalle erforderlich (d.h. etwas wie $\mathrm{card}(\mathrm{Voraussetzung}, \mathrm{voraus}) = (4, \ldots)$ wäre nicht möglich)!

$$\begin{aligned} \mathrm{card}(\mathrm{legt\ ab}, \mathrm{Student}) &= (0, \infty) \\ \mathrm{card}(\mathrm{legt\ ab}, \mathrm{Ablegeform}) &= (1, 4) \\ \mathrm{card}(\mathrm{legt\ ab}, \mathrm{gehaltene\ Vorlesung}) &= (3, \infty) \end{aligned}$$

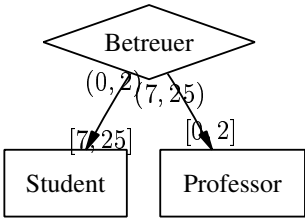
In der graphischen Notation:



Weitere Einschränkung:

$$\text{card}(\text{legt ab}, \text{Student}, \text{gehaltene Vorlesung}) = (0, 2)$$

Die hier verwendete Semantik bezeichnet man als *Partizipationssemantik*, eine andere Semantik ist die *Lookup-Semantik* (z.B. UML): Es gilt $\text{card}^*(R, R_1, R_2) = [m, n]$ für Komponenten R_1, R_2 in R^C, falls jedes Objekt in R_1^C mit höchstens n und mindestens m Objekten von R_2^C über R_C assoziiert ist:

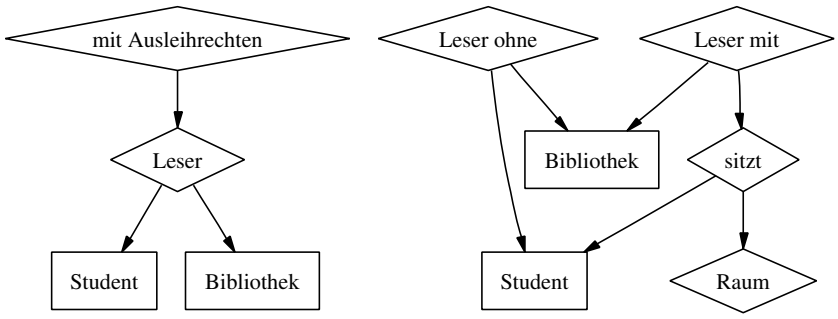


Lookup- und Partizipationssemantik sind nicht direkt übertragbar für n-stellige Typen ($n > 2$).

Es gelten folgende Beziehungen:

- $\text{card}(R, R') = (0, 1)$ ist äquivalent mit R' ist Schlüssel in R
- $\text{card}(R[R''], R') = (0, 1)$ ist äquivalent mit funktionaler Abhängigkeit $R' \to R''$
- $\text{card}(R, R') = (1, \infty)$ ist äquivalent zur Inklusionsabhängigkeit $R' \subseteq R[R']$

- *Exklusionsabhängigkeiten* und *Wertebereichsbeschränkungen*: im relationalen Modell kann für Attribute $R^C \subseteq M \subsetneq \text{dom}(R)$ gelten (mit M als Einschränkung), z.B. Matrikelnummer oft *Jahreszahl*$\times$*num[4]*. Hier in der Universität können nur Personen mit festem Zimmer Bücher ausleihen. Zwei Formen der Modellierung:



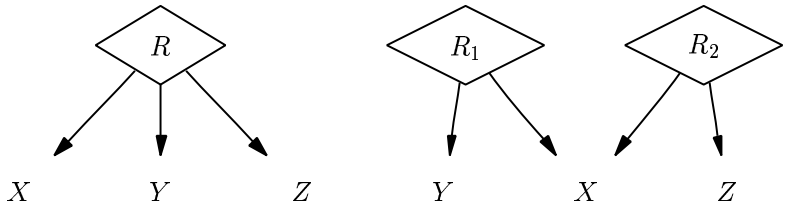
- *Merhwertige Abhängigkeiten*: Es gilt $X \twoheadrightarrow Y|Z$ (für Komponentenmengen X, Y, Z von R) nach klassischer Definition, wenn für alle $o, o' \in R^C$ gilt:

  $$o[X] = o'[X] \Rightarrow \exists\, o'' \in R^C((o[X \cup Y] = o''[X \cup Y]) \wedge (o'[X \cup Z] = o''[X \cup Z]))$$

  Andere Äquivalenz: $R^C \models X \twoheadrightarrow Y|Z$ genau dann, wenn

  $$R^C[X \cup Y \cup Z] = R^C[X \cup Y] \bowtie R^C[X \cup Z]$$

  Insbesondere für Relationship-Typen (o.B.d.A. enthält $X \cup Y \cup Z$ alle Komponenten von R und sind paarweise disjunkt): Dann läßt sich dies folgendermaßen umformen:



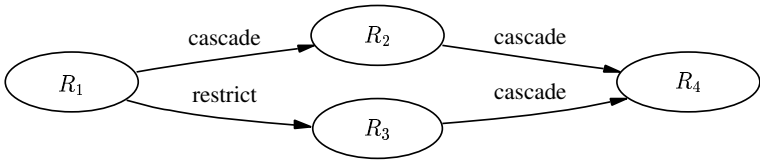
4.2.1 Rahmen zur Definition von Abhängigkeiten

Wichtig ist neben der Definition der Abhängigkeit auch eine Strategie zum Auffinden einer **Ungültigkeit einer Abhängigkeit**. Sei beispielsweise LeserMit[Student] $\subseteq$ sitzt[Student] – was passiert nun beim Einfügen, beim Ändern oder beim Löschen von Daten? Möglichkeiten:

- *cascade:* Modifikation wird weitergegeben
- *restrict:* Modifikation nicht erlaubt
- *no action:* Verhält sich meistens genauso wie *restrict*
- *setNull/setDefault:* Werte werden auf `NULL` oder den Default-Wert gesetzt

Zudem ist eine Priorisierung (im Falle der Verletzung weiterer Abhängigkeiten) notwendig sowie evtl. Sonderbedingungen für spezielle Lebensphasen.

Weiterhin treten Probleme bei Mengen von Integritätsbedingungen auf – betrachte folgende Relationen bei einem *delete*:



Soll nun beim Löschen in R_1 auch ein Eintrag in R_4 gelöscht werden oder ist dies nicht erlaubt? Um dies zu lösen, muß eine Standard-Strategie für alle Typen vorgegeben werden, oder die Auflösung des Strategiegraphen muß mitprotokolliert werden und ggf. wieder rückgängig gemacht werden.

4.3 Übertragung von ER-Konstrukten in relationale

Problem bei der Umsetzung der ER-Konstrukte (Strukturen und Integritätsbedingungen) ist die begrenzte Ausdrucksstärke des relationalen Modells: Es hat eine sehr einfache Struktur, damit größere Beschreibungskomplexität und nur rudimentäre Menge von Integritätsbedingungen.

4.3.1 Übertragung von Strukturen

Wir führen die Übersetzung in mehreren Schritten durch (wobei ein Schritt ggf. Obligationen für weitere Schritte beinhaltet):

1. Zerlegung strukturierter *Attribute* in atomare (Überführung in die 1. Normalform).[5] Zur Übersetzung von Konstruktoren gibt es folgende Möglichkeiten:

   - Mengen-Konstruktor
     (a) Beibehalten
     (b) Abbildung auf eine Beschränkte Menge, z.B. $B\{\mathsf{AkadTitel}\} \rightsquigarrow \{\mathsf{AkadTitel1}, \mathsf{AkadTitel2}, \mathsf{AkadTitel3}\}$ mit Präferenzregel zum Befüllen.
     (c) Abbildung auf Konkatenationstypen, etwa

     $$\text{STRING}[x_1] \text{ delimiter } \text{STRING}[x_2] \ldots$$

     mit zugehöriger Verarbeitungsinformation.
     (d) Einführung eines neuen Schemas.

     Dazu kommt die Obligation für Übersetzung der Integritätsbedingungen.
   - Listenkonstruktor: a), b), c), d)

[5] Neuerdings sind oft auch *user-defined data types* erlaubt.

- Tupelkonstruktor: a), b)
- Multimengenkonstruktor, Arraykonstruktor... analog.

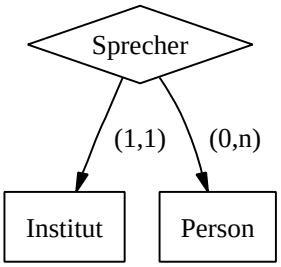
2. Übersetzung von *Entity-Typen* mit DBMS-gerechten Attributen: Direkte Übertragung in relationales Schema. Zur Übertragung der Semantik (mit ggf. Restrukturierung) siehe nächstes Kapitel.

3. *Relationship-Typen*: Sei ein Typ $R = (R_1, \ldots, R_n, \{A_1, \ldots, A_m\}, I)$ gegeben, bestehend aus Relationship-Typen R_i, Attributen A_i und Integritätsbedingungen I. Wir nehmen an, dass alle Komponenten schon übersetzt sind. Wir konstruieren eine Tabelle, die die Identifikatoren der Komponenten (z.B. Primärschlüssel) beinhaltet. Wir machen also die Abbildung in folgendens relationales Schema:

$$R \rightsquigarrow (\text{key}(R_1) \cup \ldots \cup \text{key}(R_n) \cup \{A_1, \ldots, A_m\}, \text{key}(R), \emptyset)$$

Dazu kommt die Obligation für Übersetzung der Integritätsbedingungen. Die Tabelle kann ggf. erweitert werden, z.B. um Rollen der Komponenten in der Relationship.

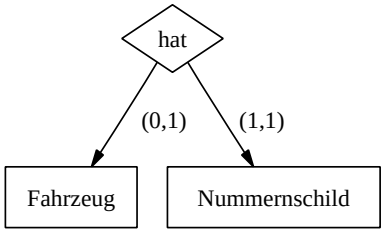
In einigen Fällen lässt sich eine vereinfachte Übersetzung durchführen. Im Falle eines binären Typs mit $(1,1),(0,n)$-Kardinalität ist die Einbettung in die $(1,1)$-Komponente möglich. Beispiel:



lässt sich übersetzen als

$$\mathsf{Institut} = (\{\mathsf{Name},\ \mathsf{Tel},\ \mathsf{Adresse},\ \mathsf{Sprecher}\}, \mathsf{Name}, \emptyset)$$

Mit Obligation: $\mathsf{Institut}[\mathsf{Sprecher}] \subseteq \mathsf{Person}[\mathsf{Key}]$. Im Falle eines Typs mit $(1,1),(0,1)$-Kardinalität ist das Zusammenführen der Relationen für beide Komponenten möglich. Beispiel:

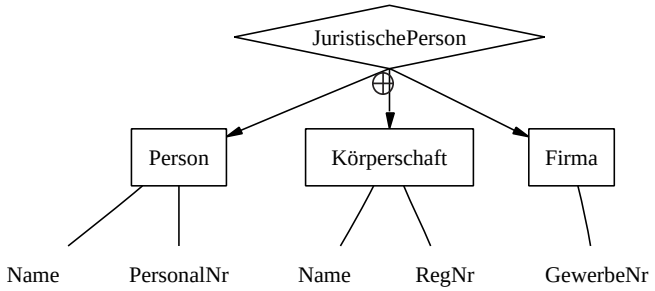


lässt sich darstellen als

$$\mathsf{Fahrzeug} = (\{\mathsf{ID},\ \mathsf{Modell},\ \mathsf{NummernschildNummer}\}, \mathsf{ID}, \emptyset)$$

Hier ist also kein Verweis auf eine andere Relation mehr, sondern die Nummernschild-Information wird gleich in Fahrzeug mitgeführt. n-stellige Relationship-Typen mit $(1,1),\ldots$-Kardinalität lassen sich analog behandeln.

4. *Cluster-Typen* werden dargestellt mit Hilfe einer zusätzlichen Relation mit Attributen {ID, KomponentenTyp} und Obligationen für Integritätsbedingungen. Beispiel:



Die neue Relation wäre JuristischePerson mit {ID, KomponentenTyp}, wo als ID jeweils PersonalNr, RegNr oder GewerbeNr der Komponente stehen kann und wo KomponentenTyp die Werte Person, Körperschaft oder Firma annehmen kann.

Wichtig sind **Regeln zum Umgang mit Vererbungshierarchien**, z.B. Student, Professor, die vom Typ Person abgeleitet sind. Hier gibt es folgende Modelle:

- *event-separation:* Jeder Typ führt den vollständigen Satz von Attributen mit. Man macht die Aufteilung: Student, Professor, AnderePerson
- *event-nonseparation:* Attribute verbleiben im wesentlichen beim Supertyp.

- *weak-universal:* Person mit Attribut für Typ.
- *union:* Vollständige Attributvererbung.

4.3.2 Übertragung von Integritätsbedingungen

Zur Übertragung von Integritätsbedingungen bestehen folgende Möglichkeiten:

- *Restrukturierung des Schemas.*
- *Deklarative:* Schlüssel, Wertebereichsabhängigkeiten, referentielle Integritätsbedingungen (schlüsselbasierte Inklusionsabhängigkeiten)
- *Prozedurale:* Konstrukte der DBMS, wie Trigger oder stored procedures.
- *Abbildung in Wirtssprache.* Problem: die Lebensdauer der Datenbanksysteme ist oft größer als die der Programmiersprachen. Außerdem will man unabhängig von Programmierumgebungen bleiben.
- *Zusätzliche integritätssichernde Maßnahmen:* Wohldefinierte Schnittstellen, Ausnahmebehandlung, Transaktionen. Bei den Schnittstellen Problem der Disziplinierung: Der Programmierer (Benutzer) will sich nicht an die Regeln der Schnittstelle halten, sobald er diese umgehen kann.

Bei der Integritätssicherung muss man beachten:

- *Gleichheit von Werten:* Wann sind Null-Werte und Default-Werte gleich? Möglichkeiten:
  - Derartige Tupel nicht betrachten (default)
  - `FULL MATCH`: Separation in volldefinierte und andere Tupel: der Tupel (`NULL`, ..., `NULL`) ist gleich nur sich seblst und nichts anderem.
  - `PARTIAL MATCH`: Betrachtung Attributweise.
- *Zeitpunkt der Überprüfung:* Zu jeder Integritätsbedingung kann mittels `INITIALLY IMMEDIATE` und `INITIALLY DEFERRED` festgelegt werden, ob sie direkt nach Ausführung einer SQL-Anweisung, oder nach Ausführung einer Transaktion überprüft werden soll, z.B.

```
CREATE TABLE T1 (...)
CONSTRAINT FOREIGN KEY (a) REFERENCES T2
INITIALLY DEFERRED;
```

- *DBMS-Intern*: Kontrollzeitpunkt nach/vor Anweisung, Kontrollbereich tupelweise/anweisungsweise.

Zu **deklarativen Spezifikationen** gehören:

- Spezifikationen, die mit einer konkreten Tabelle assoziiert sind: `PRIMARY KEY`, `UNIQUE`, `CHECK(`*Bedingung*`)`, wobei als Bedingung eine einwertige SQL-Abfrage erwartet wird.
- Kontrollanweisungen außerhalb einer Tabellendefinition:

  `CREATE ASSERTION` *name* `CHECK` (*condition*)

Zu **prozeduralen Spezifikationen** gehören:

- *Trigger:* Routinen, die beim Auftreten bestimmter Ereignise ausgeführt werden. Beispiel (Oracle):

```
CREATE/REPLACE TRIGGER TriglnstitutFakult
AFTER INSERT ON Institut
FOR EACH ROW
    WHEN (new.Fakultaet NOT IN
          (SELECT Nummer FROM Fakultaet))
    BEGIN INSERT INTO Fakultaet(Nummer)
          VALUES (:new.Fakultaet)
    END;
```

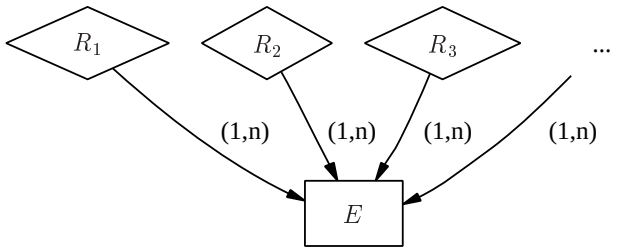
  Analog in Sybase, DB2, MS SQL mit folgenden Einschränkungen:

  - In Sybase 1 Trigger pro Ereignis.
  - In DB2 und Oracle 1 Ereignis pro Trigger.
  - In Informix n Ereignisse pro n Trigger.

  Probleme beim Benutzen von Triggern:

  - delete-insert- bzw. insert-delete-Abhängigkeiten. Betrachte folgendes Beispiel: Relationen C_1, C_2, C_3 mit Integritätsbedingungen $C_1 \subseteq C_2, C_1 \subseteq C_3$ und $C_2 \cap C_3 = \emptyset$. Was passiert beim Hinzufügen eines Tupels zu C_1? Das Ergebnis ist unterschiedlich in Abhängigkeit von der Reihenfolge, in der die Trigger gefeuert werden.
  - Effektbeibehaltung – will man, dass wenn ein Element hinzugefügt wird, dieses sofort gelöscht wird?
  - Lokalität versus Globalität – ein Trigger wird lokal gefeuert und global wirksam.

- *Stored Procedures:* Eine Möglichkeit mehrere zusammenhängende Anweisungen gleichzeitig atomar auszuführen. Besonders nützlich, wenn die Anweisungen bei Ausführung einzeln eine Integritätsbedingung verletzen würden. Beispiel: ein Student muss immatrikuliert und für ein Fach eingeschrieben sein. Man kann einen neuen Studenten nicht in die Immatrikulationstabelle eintragen, weil er nicht in seiner Fachtabelle drin steht und umgekehrt. Allgemeiner: bei einer Sternarchitektur



  wird eine stored procedure verwendet, die mehrere `INSERT` gleichzeitig macht:

$$\texttt{INSERT}\ E \rightsquigarrow \texttt{INSERT}\ R_1,\ \texttt{INSERT}\ R_2,\ \texttt{INSERT}\ R_3,\ \ldots$$

Siehe auch 4.2.1, wo die Erzwingungsmodi für die Integritätsbedingungen beschrieben sind.

4.4 Normalisierung

Bei der Normalisierung geht es um Optimierung von Relationen (sowohl Strukturierung als auch Funktionalität). Betrachte als Beispiel die Relation Projekt mit Attributen

{ProjNr, ProjBez, ProjKonto, ProjMitarbSVNr, ProjMitarbName, AbtProjMitarb, LeiterProjMitarb, AdresseAbt, Stunden, Zeitraum, Betrag, MitarbKonto}

Der Primärschlüssel sei {ProjNr, ProjMitarbSVNr, Zeitraum, Stunden} und die funktionalen Abhängigkeiten seien

- {ProjNr} → {ProjBez, ProjKonto}
- {ProjNr, ProjMitarbSVNr, Zeitraum, Stunden} → {Betrag}
- {ProjMitarbSVNr} → {MitarbKonto, ProjMitarbName, AbtProjMitarb}
- {AbtProjMitarb} → {LeiterProjMitarb}

- ggf. $\{\mathsf{AbtProjMitarb}\} \rightarrow \{\mathsf{AdresseAbt}\}$ – hier kann ein Problem auftreten, da z.B. die Informatik sich in der Olshausentsraße befindet, aber auch in HRS3.

Probleme zur Lösung:

- *Redundanz* – 12 mal pro Jahr wird die gleiche Information mitgeführt (falls Gehälter monatlich abgerechnet werden).
- *Insert-Anomalie* – Informationsblockierung – man kann nichts zum Projekt eintragen, wenn noch keine Mitarbeiter da sind.
- *Delete-Anomalie* – Informationsverluste – wenn der letzte Eintrag zur Informatik verschwindet, verschwindet die ganze Informatik!
- *Update-Anomalie* – Verhaltensveränderung – wenn ein Mitarbeiter heiratet und den Familiennamen ändert, wie viele Tupel muss man verändern?
- *Performanz*

Lösung: Zerlegung des Schemas in sinnvolle Einheiten (Normalisierung). Vorteile: redundanzarm, keine Anomalien, Schemastabilität. Nachteile: Pflege des Zusammenhangs, Modifikation ggf. komplexer. [6]

Die Dekomposition $(R_1, \ldots, R_n)$ von $R = (X, \Sigma)$ mit $R_i = (X_i, \Sigma_i)$ soll folgende Forderungen erfüllen:

- *Informationsverlustfreiheit:* $R^C = \bowtie_{i=1}^{n} R^C[X_i]$ (wobei „$\subseteq$“ stets gilt)
- *Semantikverlustfreiheit:* $(\bigcup_{i=1}^{n} \Sigma_i) \cup \Sigma^* \models \Sigma$, wobei Σ^* besagt, dass $R^C[X_i \cap X_j] = R_i^C[X_j]$ für alle i, j gilt. Dies erlaubt also eine lokale Überprüfung funktionaler Abhängigkeiten.

Definition: Eine Komponente heißt *Nichtschlüsselkomponente*, falls sie kein Element in einem minimalen Schlüssel ist. Ein ER-Typ R ist in der *3. Normalform (3NF)*, falls für alle nichttrivialen funktionalen Abhängigkeiten $Z \rightarrow \{A\}$ (also mit $A \notin Z$) mit einer Nichtschlüsselkomponente A gilt: Z ist ein Schlüssel des Typs R.

Definition: Ein ER-Typ R ist in der *Boyce-Codd-Normalform (BCNF)*, falls für alle nichttrivialen funktionalen Abhängigkeiten $Z \rightarrow \{A\}$ gilt: Z ist ein Schlüssel des Typs R.

[6] Aus diesem Grund macht man oft einige Normalisierungsschritte rückgängig (Denormalisierung). So wird z.B. an Effizienz gewonnen, da für Abfragen mit denormalisierten Relationen weniger Joins nötig sind.

3NF ist also etwas schwächer: sie sagt nichts über die Abhängigkeiten innerhalb von Schlüsseln aus. 3NF ist stets erreichbar ohne Verlust von funktionalen Abhängigkeiten. Ein weiterer Vergleich:

	Vorteile	**Nachteile**
3NF	Informationsverlustfrei	NP-Nachprüfung
BCNF	P-Nachprüfung, Informationsverlustfrei	Semantikverlust

Beispiel: Für die Relation Projekt wäre eine Dekomposition in 3NF wie folgt (hier nur Attributmengen):

{ProjNr, ProjBez, ProjKonto}
{ProjNr, ProjMitarbSVNr, Zeitraum, Stunden, Betrag}
{ProjMitarbSVNr, ProjMitarbName, MitarbKonto, AbtProjMit}
{AbtProjMit, LeiterProjMitarb, AdresseAbt}

Algorithmen zur Konstruktion einer 3NF: Input: Eine Menge R von Attributen und eine Menge Σ von funktionalen Abhängigkeiten. Output: Eine Dekomposition von R in 3NF.[7]

- **Analysealgorithmus:** Durchprobieren aller möglichen Dekompositionen.

  1. Für $X \to Y \in \Sigma$ Dekomposition in $X \cup Y$ und $R \setminus Y \cup X$.
  2. Gibt es weitere Abhängigkeiten in Dekomposition?

     Ja: Schritt 1 über dekomponierten Typ

     Nein: Prüfung auf 3NF. Ist keine Normalform vorhanden, so mache einen Schritt rückgängig und versuche Schritt 1 mit einer anderen Abhängigkeit.

  Problem: ggf. exponentiell. Beschleunigungstrick: Blattabhängigkeiten zuerst betrachten. Blattabhängigkeiten sind solche, wo alle Attribute der rechten Seite in keiner anderen Abhängigkeit links stehen.

- **Synthesealgorithmus:** Reduktion der Abhängigkeitsmenge.

  1. Herstellung einer *kanonischen Überdeckung* Σ_C für Σ, d.h.
     - $\Sigma_C^+ = \Sigma^+$

[7] Die Tabelle oben und die Algorithme sind mit Fehlern — wird korrigiert. Solange siehe A.3

- jede Abhängigkeit in Σ_C besitzt einelementige rechte Seite
- Σ_C ist minimal: es existiert keine Abhängigkeit $X \to Y \in \Sigma_C$ mit $\Sigma_C \setminus \{X \to Y\} \models \Sigma_C$
- Σ_C ist linksminimal: es existiert keine Abhängigkeit $X \to Y \in \Sigma_C$ und Komponente $A \in X$ mit

$$(\Sigma_C \setminus \{X \to Y\}) \cup \{X \setminus \{A\} \to Y\} \models \Sigma_C$$

Berechnung kanonischer Überdeckung:

- Linksreduktion
- Rechtsreduktion
- Entfernung trivialer Abhängigkeiten, insbesondere $X \to \emptyset$
- Zusammenführung nach linken Seiten:

$$X \to Y_1, \ldots, X \to Y_n \quad \rightsquigarrow \quad X \to Y_1 \cup \ldots \cup Y_n$$

2. Einführung eines Typs für jede funktionale Abhängigkeit in berechneter Überdeckung.
3. Falls kein minimaler Schlüssel in einem Typ vollständig enthalten ist, dann wird ein Typ mit einem minimalen Schlüssel hinzugefügt.
4. Falls die Komponenten eines Typs enthalten sind in den Komponenten eines anderen Typs, dann wird der erste Typ entfernt.

Noch einmal zum **Problem** der Boyce-Codd-Normalform: Eine verlustfreie und abhängigkeitserhaltende Dekomposition ist nicht immer gewinnbar. Ursachen sind neben inhärent zyklischen Abhängigkeiten auch Modellierungsfehler. Betrachte dazu folgende Abhängigkeiten als Beispiel:

$$\{\mathsf{Ort},\ \mathsf{Straße}\} \to \{\mathsf{PLZ}\} \ \text{ und } \ \{\mathsf{PLZ}\} \to \{\mathsf{Ort}\}$$

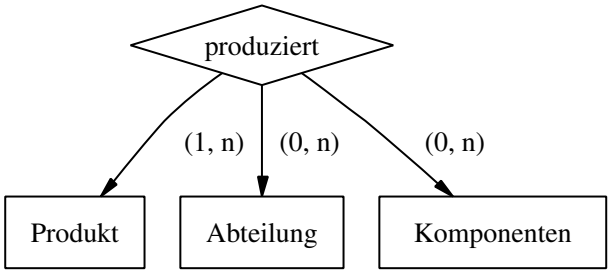
Hier sind im Prinzip verschiedene Dinge gemeint – der „linke“ Ort ist ein Ort im geographischen Sinne; und die PLZ determiniert nicht den Ort, sondern die Zustellung. Gemeint wäre daher eigentlich:

$$\{\mathsf{geographischer\ Ort},\ \mathsf{Straße}\} \to \{\mathsf{PLZ}\}$$

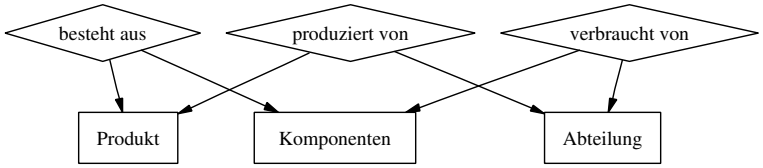
$$\text{und } \ \{\mathsf{PLZ}\} \leftrightarrow \{\mathsf{Zustellbezirk}\}$$

Dieses Problem kann also durch die Bearbeitung der Attributstruktur gelöst werden.

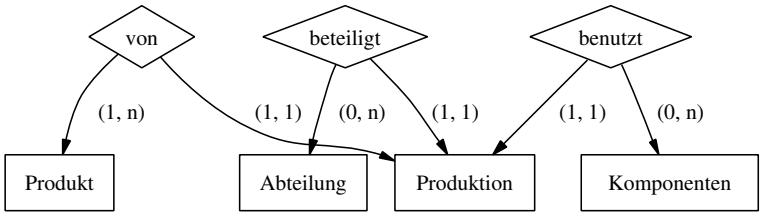
Weitere **Bemerkung**: In der Literatur wird oft behauptet, zweistellige Relationship-Typen seien ausreichend für die Modellierung von Beziehungen. Betrachte dazu folgendes Beispiel:



Der erster Versuch, dies mit zweistelligen Relationship-Typen darzustellen, ist nicht informationserhaltend und damit falsch:



Der zweite Versuch ist informationserhaltend, aber ineffizient: Es bedarf der Pflege der Integritätsbedingung, falls Produktion nicht von, beteiligt, benutzt bei Übersetzung inkludierte:

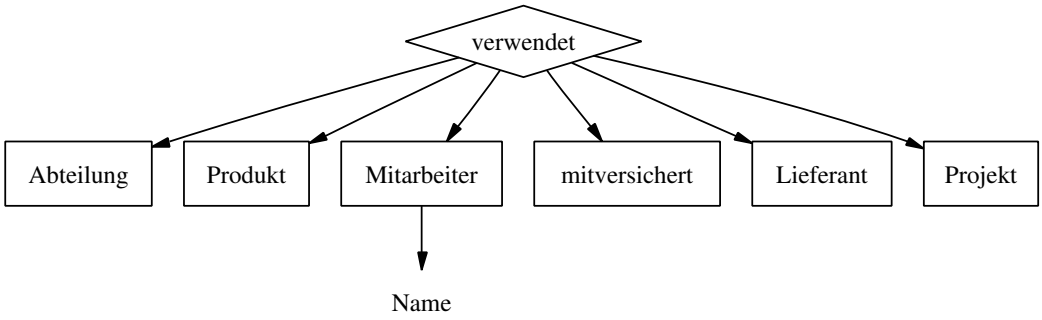


4.4.1 Mehrwertige Abhängigkeiten

Betrachte folgende Abhängigkeiten:

$$\begin{aligned}
\{\text{Name}\} &\twoheadrightarrow \{\text{Abteilung, Mitversichert}\}|\{\text{ Projekt, Produkt, Lieferant }\} \\
\{\text{Name}\} &\twoheadrightarrow \{\text{Mitversichert}\}|\{\text{ Abteilung, Projekt, Produkt, Lieferant }\} \\
\{\text{Projekt}\} &\twoheadrightarrow \{\text{Name, Abteilung, Mitversichert}\}|\{\text{ Produkt, Lieferant }\} \\
\{\text{Produkt}\} &\twoheadrightarrow \{\text{Name, Abteilung, Mitversichert, Projekt}\}|\{\text{ Lieferant }\}
\end{aligned}$$

Als Entity-Relationship-Modell:



Die Abhängigkeit $\{\mathsf{Produkt}\} \twoheadrightarrow \{\mathsf{Lieferant}\}$ ließe sich zum Beispiel umschreiben als „Wenn zu einem Produkt mehrere Lieferanten existieren, so sind diese unabhängig von der Abteilung oder dem Projekt etc.“.
Bemerkungen für mehrwertige Abhängigkeiten für einen Typen (R, Σ) und $X, Y \subseteq R$:

- Es reicht aus, $X \twoheadrightarrow Y \setminus X$ zu betrachten.
- Falls in R^C die Abhängigkeit $X \twoheadrightarrow Y$ gilt, dann gilt auch $X \twoheadrightarrow R \setminus Y$.
- Wir können $X \twoheadrightarrow Y$ und $X \twoheadrightarrow Z$ gemeinsam betrachten und als $X \twoheadrightarrow Y|Z$ schreiben.
- $X \twoheadrightarrow Y$ gilt in R^C, falls $R^C = R^C[X \cup Y] \bowtie R^C[X \cup (R \setminus Y)]$ gilt.

Wie bereits oben werden wir mehrere **äquivalente Definitionen** betrachten:

- *Formal:* $X \twoheadrightarrow Y$ gilt in R^C, falls für alle Objekte $t_1, t_2 \in R^C$ mit der Bedingung $t_1[X] = t_2[X]$ ein Objekt $t \in R_C$ existiert mit $t_1[X \cup Y] = t[X \cup Y]$ und $t_2[X \cup (R \setminus Y)] = t[X \cup (R \setminus Y)]$.
- *Verbal:* Die Y-Werte sind von den Z-Werten bezüglich der X-Werte unabhängig.
- Die verbale Definition noch einmal formal: $\sigma_{X=x}(R^C)[Y] = (\sigma_{X=x,Z=z}(R^C)[Y])$ für $X \cap Y = Y \cap Z = X \cap Z = \emptyset$
- Mit $X \cap Y = Y \cap Z = X \cap Z = \emptyset$ gilt für alle $x \in R^C[X]$:

$$\sigma_{X=x}(R^C) = \{x\} \times (\sigma_{X=x}(R^C)[Y]) \times (\sigma_{X=x}(R^C)[Z])$$

- Existenz einer *internen Strukturierung:* $X \twoheadrightarrow Y|Z$ bedeutet:

X	Y	Z
$A_1 \cdots A_n$	$B_1 \cdots B_k$	$C_1 \cdots C_m$

- *Trennung von Gesichtspunkten:* Im Beispiel kann man eine Trennung in drei Gesichtspunkte vornehmen: {Projekt}, {Produkt, Lieferant} und {Name, Abteilung, Mitversichert}.

Für Datenbanken mit mehrwertigen Abhängigkeiten existiert nun eine *vierte Normalform* (R, Σ):

Definition: Ein ER-Typ (R, Σ) ist in der *vierten Normalform (4NF)*, falls für nichttriviale $X \twoheadrightarrow Y \in \Sigma^+$ gilt, daß X Schlüssel von (R, Σ) ist.

Diese Normalform läßt sich über den Dekompositionsalgorithmus erreichen, die Komplexität für diese Normalisierung wäre aber zu hoch. Betrachte aber ein Deduktionssystem mit Axiom $X \cup Y \to Y$ und Regeln

$$\frac{X \to Y}{X \cup Z \cup W \to Y \cup Z} \quad \text{und} \quad \frac{X \to Y, Y \to Z}{X \to Z} \quad \text{sowie} \quad \frac{X \to Y}{X \twoheadrightarrow Y}$$

Dazu kommen folgende Regeln: kommt noch
Verallgemeinerung: Die mehrwertigen Abhängigkeiten lassen sich zu *hierarchischen Abhängigkeiten* ausweiten:

Definition: Es gilt die *hierarchische Abhängigkeit* $X \twoheadrightarrow Y_1|Y_2|\ldots|Y_k$ in R^C genau dann, falls alle $X \twoheadrightarrow Y_i$ in R^C gelten.

Damit ist ein **Zwischenziel** für die Normalisierung gegeben: Die *feinste hierarchische Abhängigkeit* – Abhängigkeitsbasis von Σ bezüglich X und $R \setminus X^+$.

Separation des obigen Beispiels nach {Name}:

Separation des obigen Beispiels nach {Projekt}:

Bemerkung: Es sind ggf. weitere, damit unterschiedliche Sichtweisen auf die Anwendung sichtbar; diese stehen eventuell in Konkurrenz zueinander – diese Konkurrenzen können zum Teil aufgelöst werden Auflösung durch *Schemaverfeinerung*, durch *Verschärfung der Abhängigkeiten* oder durch *Priorisierung* einer einzelnen Sichtweise.

4.5 Verhaltensmodellierung

Bisher wurden statische Aspekte des Informationssystems modelliert; doch Daten besitzen einen Lebenszyklus – sie werden benutzt und verändert, d.h. es werden Operationen auf der Datenbank ausgeführt. Daher besteht die *Spezifikation* eines Informationssystems nicht nur aus der *Struktur* (Entity-Relationship-Modell), sondern auch aus dem *Verhalten*.

Ziel ist es also, Operationen auf den Daten in der Datenbank zu modellieren. Deren Funktionalität ergänzt die Struktur und wird durch diese bestimmt.

Probleme dabei:

- *Komplexität*
- *software-orientierter Entwurf*
- *aspektorientierte Funktionalität:* Funktionalität, die nicht mit der Struktur direkt zusammenhängt bzw. nicht in die Struktur abgebildet werden kann, z.B. Navigation
- *Vermischung der Abstraktionsebenen:* von Anforderungn bis zur Programmierung
- *Wiederverwendbarkeit, Verständlichkeit, Fehlertoleranz*

Idee ist also, neben der Struktur auch das Verhalten zu Modellieren. Ansatz dazu: Informationssysteme sind reaktive Systeme, d.h. der „Benutzer“ (im weitesten Sinne) löst ein Ereignis aus und das System reagiert auf dieses Ereignis. Welche Ereignisse spielen eine Rolle?

Ausgangspunkt: Benutzeranforderungen, Motive, Ideen, Aufgaben:

- Welche Benutzer erledigen welche Aufgaben?
- Wie wird das System gewartet und gepflegt?
- ...

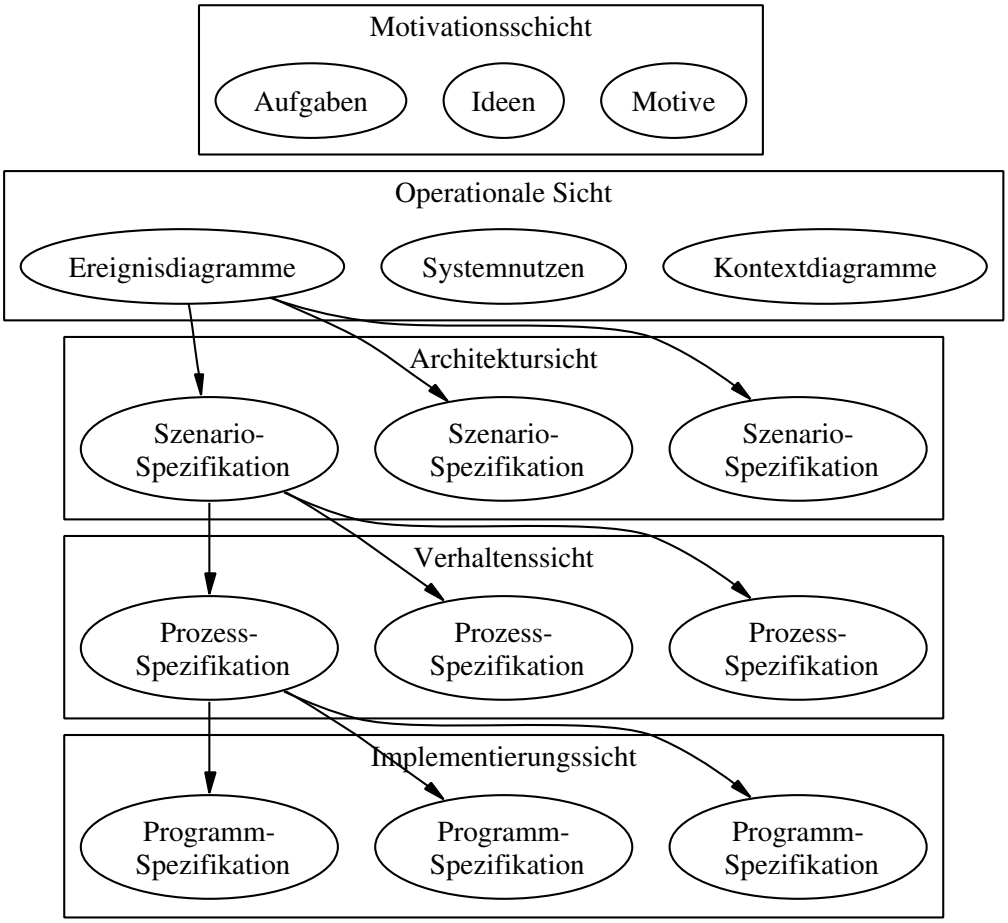


Abbildung 2: **Sichten** auf ein System

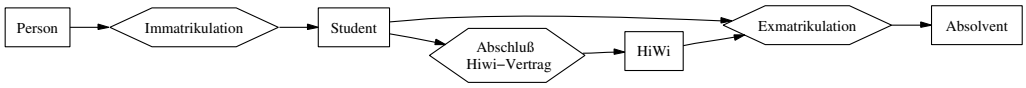
Dabei können Ereignisse nicht beliebig auftreten, es gelten dynamische Integritätsbedingungen. Außerdem ist das System in eine beschränkte Umgebung eingebettet, es existieren also Operationsbegrenzungen wie z.B. Kapazität, Zeit oder Sicherheit.

Wir definieren nun verschiedene **Sichten** auf das System: Die *operationale Sicht* auf das System besteht aus dem *Systemnutzen*, aus *Kontextdiagrammen* und aus *Ereignisdiagrammen*. Letztere werden in der *Architektursicht* aufgesplittet in einzelne *Szenarien*. Diese werden in der *Verhaltenssicht* in einzelne *Prozesse* zerlegt, die in der Implementierungssicht in Programme umgesetzt werden.
Für all diese Ebenen werden **Sprachen** benötigt, d.h. zur Darstellung von

Ereignissen und deren Beziehungen, zur Darstellung von Handlungsabläufen, Prozessen und schließlich Programmen. Dabei gibt es im wesentlichen zwei Ansätze, die *ereignis-* und die *zustandsorientierten Richtungen.*

4.5.1 Ereignisorientierte Spezifikation

Eine Möglichkeit ist die Darstellung basierend auf Petrinetzen, dabei wird das Verhalten als Menge von Ereignissen und Sichten (Input- oder Output-Generatoren) modelliert. Als kurzes **Beispiel** den „Lebenszyklus“ eines Studenten:



Bei der *Ereignisgesteuerten Prozesskette (EPK)* wird der Ablauf durch einen Graphen beschrieben, dabei sind die Knoten Funktionen (Kästen), Ereignisse (Sechsecke) und Verknüpfungsoperationen wie `AND` oder `OR` (Kreise). Für ein Beispiel siehe Graphik.

Probleme dabei sind u.a. das Aufblähen der Spezifikation (durch das Wechselspiel Ereignis, Funktion, Ereignis, Funktion, ...), die fehlende Abstraktion und fehlende Hierarchie, keine zeitliche Schichtung und keine Integration der Organisationsstruktur – dies ließe sich durch Erweiterungen ergänzen, etwa indem *Akteure* bzw. *Rollen* mit in das Diagramm aufgenommen werden; ähnlich für die Modellierung von Datenflüssen.

Somit läßt sich die ereigsnisgesteuerte Prozesskette eher zur Anforderungsanalyse benutzen als für wirkliche Spezifikation.

4.5.2 Zustandsorientierte Spezifikation

Idee: Die zustandsorientierten Spezifikationen verwenden *Statecharts* in verschiedenen Varianten als Verallgemeinerung endlicher Automaten, um Handlungsabläufe und Prozesse zu beschreiben und somit das Verhalten zu visualisieren. Die Diagramme sind induktiv Aufgebaut, bieten eine rigide Semantik und Interoperabilität, und sind auf breiter Ebene akzeptiert.

Aufbau: Zustände des Automaten sind Zustände des Modells, Transitionen sind beschriftet mit auslösenden Ereignissen (Bedingungen) und/oder mit auszuführenden Aktionen. Zudem werden Start- und Endzustände angegeben (ausgefüllter Kreis bzw. ausgefüllter Kreis mit doppeltem Rand).

Mittels *hierarchischer Statecharts* können Teile zusammengefaßt werden, d.h. Zustände können hierarchisch gruppiert werden. Ein Objektzustand ist aktiv, wenn einer seiner Unterzustände aktiv ist.

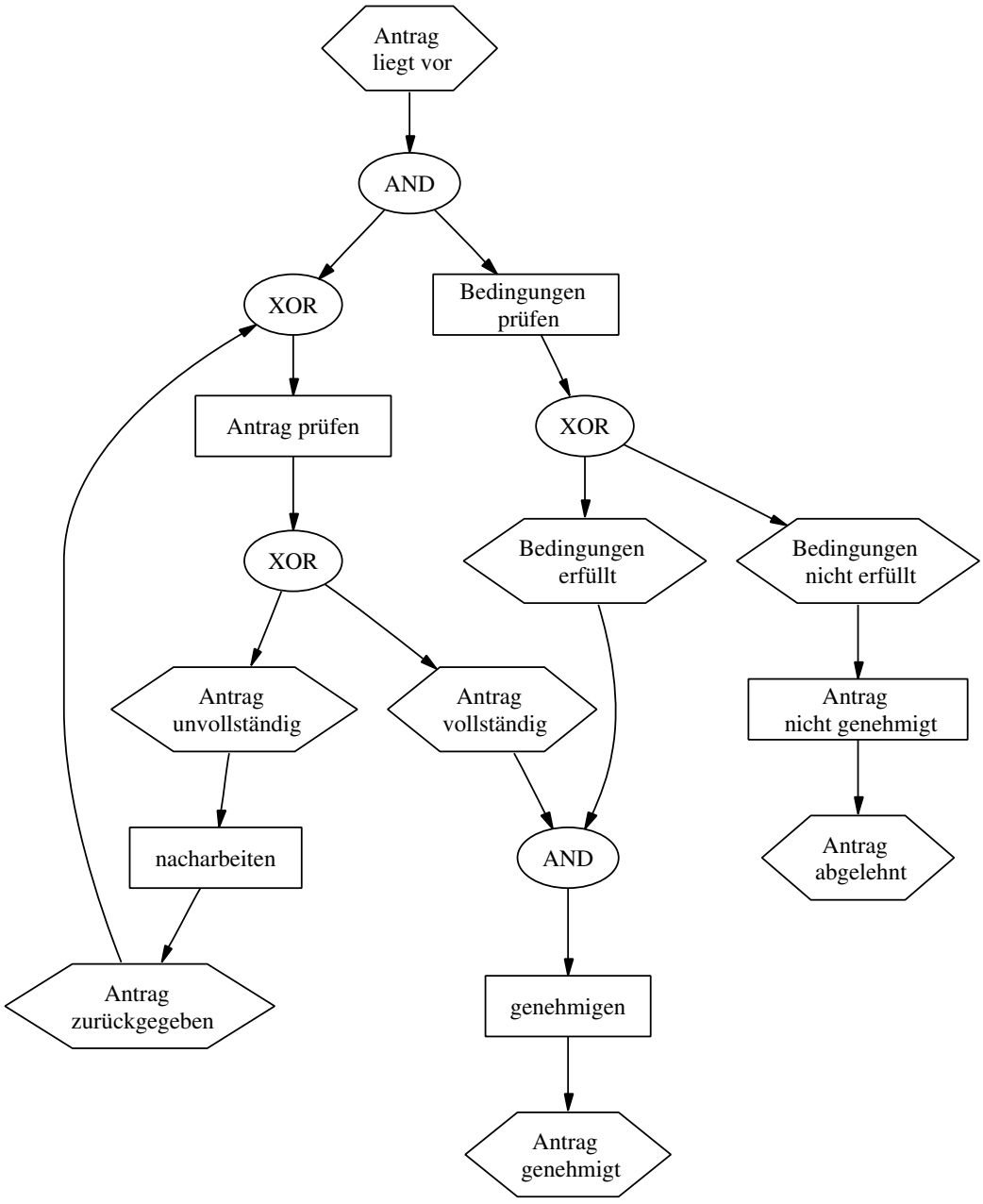


Abbildung 3: Beispiel für die **ereignisgesteuerte Prozesskette**

Erweiterungen gibt es u.a. zur Steuerung der Ablaufkontrolle: Verzweigung, Zusammenführung, Parallelausführung, Synchronisation, Zuordnung von Akteuren.

4.5.3 Abstrakte Zustandsmaschinen

Bisher kennengelernt:

- Spezifikation des Systemeverhaltens (EPK, Statecharts).
  - Wir konzentrieren uns auf die Abläufe im System.
  - Zustandsbasiert oder ereignisbasiert.
  - Kommunikationsschnittstelle
- Spezifikation der DB-Anwendung aus Nutzersicht (SiteLang).
  - Abläufe in der Mensch-Maschine-Kollaboration.
  - Akteure (Abstraktion einer Gruppe von Benutzern). Besitzen: ein Portfolio (Beschreibung der Eigenschaften, der Aufgaben, Rechte, Pflichten) und ein Profil (Beschreibung der Haupteigenschaften der Gruppe der Nutzern)
  - Kollaboration (Kommunikation, Kooperation, Koordination)

Es gibt folgende Ansätze zur Verhaltensmodellierung:

- *CRC-Karten* (Class Responsibility Collaboration) wurden 1975 eingeführt, um die Spezifikation der Aufgabe zu ermöglichen. *Class:* Spezifikation der Typen für Schnittstellen; *Responsibility:* abstrakte Beschreibung der Abläufe; *Collaboration:* Beziehungen zwischen den Objekten – alles auf einer Karteikarte darstellbar.
- *Abstrakte Zustandsmaschinen (ASM).* Anforderungen:
  - sowohl eine allgemeine Spezifikation, als auch eine implementationsnahe Spezifikation
  - Verhaltensbeschreibung
  - Verfeinerungen zum konkreten Maschinenmodell
  - eine Möglichkeit, parallele Abläufe darzustellen

Wir definieren hier die abstrakten Zustandsmaschinen. Die Zustände von ASMs sind Algebren. **Formal**:

Definitionen:

- Die *Signatur* Σ einer ASM besteht aus Funktionennamen[8] $f_1, \ldots, f_n, \ldots$ mit Arität n_i für f_i mit ggf. nullstelligen Funktionen (Konstanten). Jede Signatur soll zumindest die Konstanten `UNDEF`, `FALSE`, `TRUE` enthalten.
- Ein *Zustand* $\mathcal{A}$ zu einer Signatur Σ besteht aus:
  - Einer nichtleeren Menge (*Universum*) W
  - Einer Funktion $f_i^{\mathcal{A}} \colon W^{n_i} \to W$ zu jedem $f_i \in \Sigma$. Im Folgenden lassen wir das Zeichen $\mathcal{A}$ bei Funktionen oft weg. Die Funktionen mit Werten `UNDEF` interpretieren wir als partielle Funktionen. Prädikate bzw. Relationen stellen wir durch Funktionen der Art $f \colon W^n \to \{\texttt{UNDEF}, \texttt{TRUE}\}$ dar.

Die Zustände werden durch die sog. *Updates* verändert, die besagen, dass der Wert einer Funktion an einer bestimmten Stelle sich ändern soll. **Formal**:

Definitionen:

- Eine *Lokation* in einem Zustand $\mathcal{A}$ ist ein Paar $l = (f, (a_1, \ldots, a_n))$, bestehend aus einem Funktionennamen f und $a_1, \ldots, a_n \in W$. Der *Inhalt* der Lokation, i. Z. $\mathcal{A}(l)$, ist der Wert $f(a_1, \ldots, a_n)$.
- Ein *Update* in einem Zustand $\mathcal{A}$ ist ein Paar (l, v) mit Lokation l und $v \in W$. Ein Update heißt *trivial*, falls $f(a_1, \ldots, a_n) = v$ ist.
- Eine Menge M von Updates U heißt *konsistent*, falls für alle $(l, v), (l, v') \in M$ schon $v = v'$ gilt.

Definition:

- Durch *Anwendung* eines Updates U auf den Zustand $\mathcal{A}$ entsteht ein

[8] Beachte: Funktionennamen sind keine Funktionen, sondern nur Symbole.

neuer Zustand $\mathcal{A} + U$, so dass für alle Lokationen l gilt

$$(\mathcal{A} + U)(l) = \begin{cases} v & \text{für } (l, v) \in U \\ \mathcal{A}(l) & \text{sonst} \end{cases}$$

Welche Updates in jedem Zustand anzuwenden sind, wird mit Hilfe der *Transitionsregeln* beschrieben, die in Form eines Computerprogramms formuliert werden. Die Semantik ergibt sich also dadurch, dass die Regeln, auf einen Zustand angewandt, eine bestimmte Menge von Updates ergeben. Wir geben hier eine induktive Definition an. Dabei beschreiben wir die Semantik nur Umgangssprachlich. [9]

[9] Für formale Definition der Semantik, siehe z.B. [Börb] oder [Stä]

Definition: Seien $\varphi, \alpha(x)$ Formeln der Prädikatenlogik erster Stufe über der Signatur Σ (mit Variablen $x_i, i \in \mathbb{N}$). Seien $s_1, \ldots, s_n, t$ Terme über Σ und P, Q Transitionsregeln. Dann seien auch folgendes Transitionsregeln:

- `skip` – Liefert eine leere Menge von Updates.
- $f(s_1, \ldots, s_n) := t$ – Liefert das Update $(f, (a_1, \ldots, a_n), v)$ von $\mathcal{A}$, wobei $a_1, \ldots, a_n, v$ die Ergebnisse der Auswertung von $s_1, \ldots, s_n, t$ bezüglich $\mathcal{A}$ sind.
- $P \parallel P'$ – Parallele Ausführung (bei Beachtung der Konsistenz von Updates)
- $P_1; P_2$ – Sequentielle Ausführung.
- `if` φ `then` P `else` Q – falls $\mathcal{A} \models \varphi$, führe P aus, sonst Q
- `let` $x = t$ `in` P – führe P aus mit Variabler x ersetzt durch t.
- `forall` x `with` $\alpha(x)$ `do` P – Für alle $a \in W$, für die $\alpha(a)$ wahr ist, führe `let` $x = a$ `in` P parallel aus.
- `choose` x `with` $\alpha(x)$ `do` P – Führe `let` $x = a$ `in` P aus mit einem beliebigen $a \in W$, für den $\alpha(a)$ wahr ist.
- $r(x_1, \ldots, x_n) = P$ – *Regeldeklaration*, die freien Variablen von P sollen in $\{x_1, \ldots, x_n\}$ enthalten sein.
- $r(s_1, \ldots, s_n)$ – Anwendung (Aufruf) einer Regel mit Parametern $s_1, \ldots, s_n$

Definition: Eine *abstrakte Zustandsmaschine* M besteht aus:

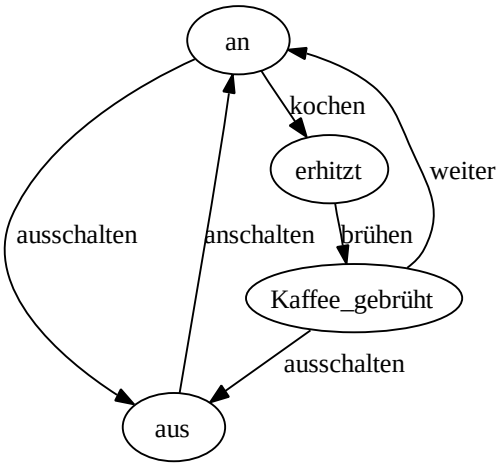
- Einer Signatur Σ
- Einem Anfangszustand $\mathcal{A}_0$
- Einer Menge von Regeldeklarationen mit einem *Hauptprogramm* – einer ausgezeichneten Regeldeklaration ohne freien Variablen (also von der Form $r = P$)

Definition: Ein *Lauf* einer ASM M mit Hauptprogramm r ist eine (ggf. unendliche) Folge $\mathcal{B}_0, \mathcal{B}_1, \ldots$ von Zuständen, so dass gilt:

- $\mathcal{B}_0$ ist der Startzustand von M.
- Falls für ein $\mathcal{B}_i$ die Regel r eine konsistente Menge U_i von Updates liefert, so ist $\mathcal{B}_{i+1} = \mathcal{B}_i + U_i$.
- Falls für ein $\mathcal{B}_i$ die Regel r eine inkonsistente Menge von Updates liefert, so ist $\mathcal{B}_i$ der Endzustand.

Beispiele: [10]

- Für die Abbildung von DB-Maschinen benutzt man 4 Funktionen: Input, Output, DBMS, DB. Damit darstellbar: parallele Ausführung von beliebig vielen Operationen, Input, Output, Berechnung.
- Wir betrachten einen Kaffeeautomaten als Beispiel eines endlichen Automaten:



  Sei Q die Zustandsmenge und Σ' das Alphabet des Kaffeeautomaten. Wir simulieren diesen Automaten auf einer ASM. Sei dazu $W = \mathbb{N} \cup Q \cup \Sigma'$. Die Signatur Σ soll folgende Funktionennamen enthalten:

  - Input – einstellig, der Wert an der Stelle $i \in \mathbb{N}$ soll den i-ten Buchstaben der Eingabe darstellen oder `UNDEF`, falls i größer als Eingabelänge ist.

[10] Für ein weiteres Beispiel siehe [Böra]

– Pos – eine Konstante, die besagt, welchen Buchstaben der Eingabe der Automat gerade liest.
– Zustand – eine Konstante, die den aktuellen Zustand speichert.

Im Anfangszustand soll Input dem Inhalt der Eingabe entsprechen, weiter soll Pos = 0 und Zustand = aus sein. Seien die Regeln wie folgt (sei dabei main die Hauptregel)

```
main =
  if (Input(Pos) = ausschalten) ∧ (Zustand = an)
      then Zustand := aus
  else if (Input(Pos) = anschalten) ∧ (Zustand = aus)
      then Zustand := an
  ⋮
  else stop;
  Pos := Pos + 1

stop =
  Pos := 0 ‖ Pos := 1
```

- Analog lassen sich auch Turing-Maschinen abbilden. ASM ist also zu einer Turing-Maschine äquivalent.

Definition: Eine *Workflow-Maschine* ist eine ASM, die wir als eine Maschine über einem Datenbankschema interpretieren und die eine zusätzliche Menge von dynamischen Integritätsbedingungen enthält.

Wir betrachten hier die **Transitionsbedingungen** der Form (Zustand, Folgezustand), die besagen, welche Transitionen zugelassen sind, z.B. die folgende Regel, die besagt, dass sich ein Gehalt nur erhöhen darf:

$$(\mathsf{Gehalt(Person)} = \mathsf{g},\ \mathsf{Gehalt(Person)} \geq \mathsf{g})$$

4.5.4 Modellierung der Interaktionen

- Akteure
- Story: Graph aus Szenen (Knoten) und Aktionen (Kanten)
- Szene: Sicht (ggf. mit Funktionen), zugelassene Akteure, Graph von Dialogschritten
- Dialogschritt: (Ereignis, Akteure, Sicht, Funktion, Modifikationsfunktionen, Vorbedingungen, Abschlussbedingung)

5 Datenbanktechnologie

5.1 Transaktionen

Definition: Eine *Transaktion* ist ein Modell, das eine sequentielle Abarbeitung für parallel ablaufende Prozesse zulässt. Außerdem sollen die Änderungen dauerhaft und verlässlich sein.

Die wichtigsten **Anforderungen** sind:

- *Unteilbarkeit von Operationen*: Die Aktion gilt nur dann, wenn alle Teilaktionen gelten (z.B. bei Banküberweisungen)
- *Parallel und unabhängig* für eine Vielzahl von Benutzern.

5.1.1 Konzept

- Syntaktisch: Einführung einer Klammer BEGIN_TRANS, END_TRANS.
- Semantisch: Ziel sind die folgenden Eigenschaften
  - Transaktionen sollen *unteilbar* sein
  - Transaktionen können *parallel* ablaufen [11]
  - Parallel ablaufende Transaktionen *unabhängig* voneinander, bzw. beeinflussen einander nur auf eine *vorhersehbare* Weise
  - Transaktionen erhalten *semantische Bedingungen*
  - Wirksam gewordene Transaktionen verändern die Daten *dauerhaft*

ACID:

- *atomicity:* „Alles oder nichts“
- *consistency:* Beginn im konsistenten Zustand ⇒ Beendigung im konsistenten Zustand
- *isolation:* Isoliert von anderen Handlungsfolgen
- *durability:* Persistent – Resultat wird dauerhaft und verlässlich verwaltet

[11] Spitzensysteme liegen bei 180.000 Transaktionen, die parallel ablaufen können. Bei der deutschen Sparkasse laufen 50.000 parallele Transaktionen, bei Lufthansa – 80.000, Parallelität der Neuronen im Gehirn - knapp 500.000.

5.1.2 Zugänge zur Integritätssicherung:

- Durch die Anwendung eigenes Programms zur Kontrolle und Pflege der Integrität. Vorteil: flexibel, effiziente Reaktion. Nachteil: Redundanz, Wartung, Dokumentation.
- Überwachung und Pflege durch *Integritätsmonitor*. Vorteil: deklarative Formulierung möglich. Nachteil: nur für axiomatisierbare Klassen
- Durch Compilierung selbstpflegender Operationenfolgen. Vorteil: automatisch. Nachteil: komplex.
- *Einkapselung:* Zwischenschicht zur Weiterleitung sicherer TA's. Vorteil: sicher. Nachteil: Entwurfsproblem.

Weitere Probleme bei **Parallelisierung:**

- Konkurrierender Zugriff auf gleiche Ressourcen
- Nebeneffekte durch konkurrierende Operationen

Lösungsansatz:

- *Zugriffsmodell* für einen Zugriffsplan und eine Serialisierung.
- *Trennung* vom lesenden und schreibenden Zugriff.
- *Abbruch, Wirksamwerden und Versionen*
- *Scheduler und Protokolle*

5.1.3 Probleme der Parallelverarbeitung

- **Problem der verlorenen Updates:** Was passiert, wenn folgende Transaktionen gleichzeitig ausgeführt werden:

TA_1	TA_2
`read(X)`	`read(X)`
`X := X - N`	`X := X + M`
`write(X)`	`write(X)`

  Durch paralleles Lesen des X-Werten geht ein Update in jedem Fall verloren.

- **Problem des temporären Updates**

TA_1	TA_2
`read(X)`	
`X := X - N`	
`write(X)`	
	`read(X)`
	`X := X + M`
	`write(X)`
`fails`	

T_2 liest den falschen Wert, nach `fails` in T_1 wird X zurückgesetzt auf den ursprünglichen Wert. Man muss die Information mitführen, dass die Prozesse sich über X beeinflussen.

- **Problem der falschen Summe**

TA_1	TA_2
	`sum := 0`
`read(X)`	
`X := X - N`	
`write(X)`	
	`read(X)`
	`sum := sum + X`
	`read(Y)`
	`sum := sum + Y`
`read(Y)`	
`Y := Y + M`	
`write(Y)`	

T_2 liest nachdem N hinzugefügt wurde, aber ehe M hinzugefügt wurde.

Lösung: was nicht parallel ausgeführt werden kann, wird sequentiell ausgeführt

5.1.4 Serialisierung

Definitionen:

- Ein *Schedule* ist eine Reihenfolge der Ausführung von Operationen, die zu den Transaktionen gehören.
- Ein *serieller* Schedule ist einer, wo alle Transitionen sequentiell nacheinander ausgeführt werden.
- Ein Schedule ist *Serialisierbar*, falls ein äquivalenter serieller Schedule existiert. Ein Äquivalenzbegriff wird noch benötigt.

[12] Ein möglicher Äquivalenzbegriff wäre die *Resultatäquivalenz:* Zwei Schedules sind äquivalent, falls gleiche Resultate erreicht. Dies ist nicht ausreichend, weil von Daten abhängt. Hier benutzen wir den folgenden Äquivalenzbegriff:

Definition: Zwei Schedules S_1, S_2 über gleicher Transaktionen-Menge sind äquivalent, falls:

- Jeder read eines Wertes X in S_1 den gleichen Wert liest, wie die entsprechende read-Operation von X in S_2
- Nach der Abarbeitung der Schedules wird gleicher Wert endgültig abgespeichert.

Algorithmus zum Testen von Serialisierbarkeit eines Schedules: Einführung eines Graphen mit

- Knoten für jede Transaktion T_i
- Kante (T_i, X, T_j) falls T_j X-Wert liest, der durch T_i vorher erzeugt wurde, ohne dass ein `COMMIT` dazwischen liegt
- Kante (T_i, X, T_j) falls T_j X-Wert schreibt, der durch T_i vorher gelesen wurde, ohne dass ein `COMMIT` dazwischen liegt

Satz: Der Schedule ist Serialisierbar gdw. der erzeugte Graph keine Zyklen besitzt.

Bemerkung: Eine topologische Sortierung des erzeugten Graphen liefert einen äquivalenten seriellen Schedule.

[12] Die Definitionen in diesem Kapitel sind nicht ganz richtig, für die verbesserte Version siehe A.4.

Problem bei diesem vereinfachenden Zugang: Nicht vorher berechenbar, da die Transaktionen nicht vorher bekannt sind, sondern kontinuierlich kommen. Bei Sortierungs-Join wird das Ergebnis nicht explizit berechnet, wir sagen nur, welche Blöcke miteinander gematcht werden Ne, wir rechnen nicht in der Anzahl der Tupel, sondern in der Anzahl der Blöcke I - Cluster-Größe

6 Recovery und verteilte DB-Systeme

6.1 Aus dem Vortrag von Dr. Kowsari

Das DB-System **SIRAX**, das von **Lufthansa Revenue Service** betrieben wird hat folgende Charakteristiken:

- 1095 Tabellen, 25000 Felder, 1095 Indizes, 1398 Fremdschlüssel, 3TB Daten
- 2500 Masken, 840 Funktionsgruppen mit 6200 Funktionsbausteinen und ca. 21000 SQL-Statements
- 1200 Online Programme, 500 Batch Programme
- 5.5 Mio lines of code
- Platzbedarf incl. Image-Archiv 17TB
- SIRAX bedient 70 Schnittstellen zu externen Systemen über 170 Schnittstellenstrukturen
- 17 Tabellen enthalten 80% der Daten

Anforderungen:

- Ca. 2000 Online-Nutzer weltweit
- Ständige Verfügbarkeit
- Antwortzeit für normalen Online-Betrieb < 3sec
- Umsatz 560 GB/Jahr, mit Images und Indizes ca. 3 TB/Jahr, 850 Mio Sätze/Jahr.
- Datenverfügbakeit 2-3 Jahre, dann Archivierung

A Vorlesungen in den Übungen

A.1 Einführung in die Prädikatenlogik

Prädikatenlogik besteht aus

- *Junktoren* $\wedge$ – und, $\vee$ – oder, $\neg$ – nicht, $\rightarrow$ – Implikation, $\leftrightarrow$ – Äquivalenz.
- *Quantoren* $\exists$ – Existenz und $\forall$ – Allquantor.
- *Variablen*, z.B. kleine Buchstaben.
- *Prädikaten*, z.B. große Buchstaben.
- *Konstanten*, z.B. Werte.

Seien a, b „Aussagen“. Wir definieren die Junktoren mit Hilfe von Wahrheitstafeln. Dabei heißt w „wahr“ und f „falsch“.

$a \wedge b$	$a = w$	$a = f$
$b = w$	w	f
$b = f$	f	f

$a \vee b$	$a = w$	$a = f$
$b = w$	w	w
$b = f$	w	f

$\neg a$	$a = w$	$a = f$
	f	w

$a \rightarrow b$	$a = w$	$a = f$
$b = w$	w	w
$b = f$	f	w

$a \leftrightarrow b$	$a = w$	$a = f$
$b = w$	w	f
$b = f$	f	w

Beachte: die Aussage $f \rightarrow w$ ist wahr (aus falsch folgt wahr). Man kann auch definieren $a \rightarrow b := \neg a \vee b$.

De Morgansche Regeln: $\neg(a \vee b)$ ist äquivalent zu $\neg a \wedge \neg b$ und $\neg(a \wedge b)$ ist äquivalent zu $\neg a \vee \neg b$. Äquivalenz zweier Formeln kann man z.B. mit Hilfe ihrer Wahrheitstafeln nachprüfen.

Beispiele:

- Variablen und Quantoren: $\exists x(\varphi x)$ übersetzt heißt: „Es existiert etwas (von uns x genannt), für das gilt φ“.
- Prädikate: grün(x). Man definiert grün $:= \{ z \mid z \text{ ist grün } \}$. Prädikate können auch mehrstellig sein, z.B. ist_enthalten_in(x, y), wobei ist_enthalten_in $:= \{ (a, b) \mid a \text{ ist enthalten in } b \}$.

- „Der Mülleimer ist nicht leer“:

$$\exists x(\text{ist\_enthalten\_in}(x,\ \text{derMülleimer}\ ))$$

  Andere Formulierung:

$$\neg\forall x(\neg\text{ist\_enthalten\_in}(x,\ \text{derMülleimer}\ ))$$

  Es gilt allgemein: $\exists x(\varphi)$ ist äquivalent zu $\neg\forall x(\neg\varphi)$.

- „Mein Weihnachtsbaum ist grün“. „Mein Weihnachtsbaum“ ist eine Konstante. „ist grün“ ist ein Prädikat. Somit kann man die Aussage schreiben als

$$\text{grün}(\text{meinWehnachtsbaum})$$

- „Etwas steht in meinem Wohnzimmer und ist grün“.

$$\exists x(\text{ist\_enthalten\_in}(x, \text{meinWohnzimmer}) \wedge (\text{grün}(x)))$$

  Mit $\forall$ kann man dies schreiben als

$$\neg\forall x\neg(\text{ist\_enthalten\_in}(x, \text{meinWohnzimmer}) \wedge (\text{grün}(x)))$$

  Mit Hilfe von de Morganschen Regeln kann man dies umformen zu

$$\neg\forall x(\neg\text{ist\_enthalten\_in}(x, \text{meinWohnzimmer}) \vee \neg(\text{grün}(x)))$$

- „Das einzige grüne in meinem Wohnzimmer ist mein Weihnachtsbaum“

$$\begin{aligned}\forall x(\quad & (\text{grün}(x) \wedge \text{ist\_enthalten\_in}(x,\ \text{Wohnzimmer}\ )) \\ \rightarrow\ & \text{ist\_gleich}(x,\ \text{meinWeihnachtsbaum}\ ))\end{aligned}$$

- „Eine Katze liegt in meinem Zimmer“: $\exists x(\text{katze}(x) \wedge \ldots)$

- „Softwareentwickler benutzen Softwaretools“

$$\forall e(\text{Softwareentwickler}(e) \rightarrow \exists s(\text{Softwaretool}(s) \wedge \text{benutzt}(e, s)))$$

A.2 Schlüssel und Funktionale Abhängigkeiten

Betrachte das folgende Schema $\underline{R} = (R, K, \Sigma)$ mit

- $R = \{\mathsf{ISBN}, \mathsf{Autor}, \mathsf{Titel}, \mathsf{Verlag}, \mathsf{UVP}\}$
- $K \subseteq R$

- $\Sigma = \{\text{Autor} \neq \texttt{NULL}, \text{UVP} > 0\}$

Gegeben sei folgende konkrete Instanz:

ISBN	Autor	Titel	Verlag	UVP
94533-1	Aaron	Alchemie	Alpha	25.30
48901-7	Buddenbrook	Brasilien	Bauers	9.99
7589-10	Cramer	Clementinen	Alpha	15.45

Dann läßt sich $\underline{R}^C$ darstellen als

$$\{(\mathsf{ISBN} \mapsto \text{94533-1}, \mathsf{Autor} \mapsto \text{Aaron}, \mathsf{Titel} \mapsto \text{Alchemie}, \\ \mathsf{Verlag} \mapsto \text{Alpha}, \mathsf{UVP} \mapsto 25.30), \\ (\mathsf{ISBN} \mapsto \text{48901-7}, \ldots), \qquad \ldots\}$$

Schlüssel sind Teilmengen von R, wobei keine zwei Tupel eines Relationenschemas die gleichen Werte in den Schlüsselattributen haben. Aus *minimalen Schlüsseln* kann kein Attribut entfernt werden, ohne daß die Schlüsseleigenschaft verlorengeht.

Achtung: Schlüssel betrachten wir dabei (hier) nur über konkreten Tupeln, wir betrachten Schlüssel nicht für ganze Schemata!

Funktionale Abhängigkeiten: Eine Menge von Attributen B ist von A abhängig, geschrieben $A \to B$, falls gilt: für alle Tupel $t, s \in \underline{R}^C$ aus $t[A] = s[A]$ auch $t[B] = s[B]$ folgt. Formal:

$$A \to B :\Longleftrightarrow \forall\, t, s \in \underline{R}^C\colon\ (t[A] = s[A] \Rightarrow t[B] = s[B])$$

Betrachte nun folgende erweiterte Relation:

ISBN	Autor	Titel	Verlag	UVP	Auflage
94533-1	Aaron	Alchemie	Alpha	25.30	2
48901-7	Buddenbrook	Brasilien	Bauers	9.99	1
7589-10	Cramer	Clementinen	Alpha	15.45	2
48901-7	Buddenbrook	Brasilien	Bauers	9.99	2

Seien dabei folgende Abhängigkeiten gegeben:

- {Autor, Titel, Verlag, Auflage} $\to$ {UVP}
- {Autor, Titel} $\to$ {ISBN}
- {Autor, Titel, Auflage} $\to$ {Verlag}
- {ISBN} $\to$ {Titel} und {ISBN} $\to$ {Autor}

Die **Hülle** einer Attributmenge α[13] eines relationalen Schemas $\underline{R}$ unter einer Menge FD von funktionalen Abhängigkeiten läßt sich berechnen durch

$\alpha^+ := \alpha$
`while` (α^+ hat sich geändert) `{`
    `for` $((B \rightarrow \gamma) \in FD)$ `{`
        `if` $(B \subseteq \alpha^+)$ `{` $\alpha^+ := \alpha^+ \cup \gamma$; `}`
    `}`
`}`

Damit ergibt sich als Hülle von {ISBN, Auflage} die ganze Menge {ISBN, Auflage, Titel, Autor, Verlag, UVP}.

A.3 Normalformen

Definitionen:

- Mit *Kandidatenschlüssel* bezeichnen wir einen minimalen Schlüssel.
- Ein *Nichtschlüsselattribut* ist ein Attribut, das kein Element eines Kandidatenschlüssels ist.
- B ist *voll funktional abhängig* von A, gdw:
  - B ist funktional abhängig von A.
  - B ist nicht funktional abhängig von jeder echten Teilmenge von A.
- *Determinant* ist eine Attributmenge, von der eine andere Attributmenge voll funktional abhängig ist.
- C ist *transitiv abhängig* von A genau dann, wenn es B gibt mit $A \rightarrow B$ und $B \rightarrow C$.

Wir definieren die Hierarchie der Normalformen:

Definitionen:

- Eine Relation ist in der *0. Normalform (NF)*, wenn sie keine berechneten Attribute besitzt.
- Eine Relation ist in der *1. Normalform*, wenn sie in der 0. Normalform ist und keine zusammengesetzte Attribute enthält.
- Eine Relation ist in der *2. Normalform*, wenn sie in der 1. Normalform ist und jedes Nichtschlüsselattribut von jedem Kandidatenschlüssel

[13] Die Hülle ist die Menge aller Attribute, die von α funktional abhängig sind.

voll Funktional abhängig ist.

- Eine Relation ist in der *3. Normalform*, wenn sie in der 2. Normalform ist und jedes Nichtschlüsselattribut von keinem Kandidatenschlüssel transitiv abhängig ist über Nichtschlüsselattribute.
- Eine Relation ist in der *Boyce-Codd-Normalform (BCNF)*[14], wenn sie in der 3. Normalform ist und jeder Determinant ein Kandidatenschlüssel ist.

  Reicht auch: Eine Relation ist in BCNF genau dann, wenn sie in der 2. Normalform ist und jeder Determinant ein Kandidatenschlüssel ist.
- Für die Definition der 4. Normalform siehe A.5

Beispiele: Betrachte die folgende Relation, in der die Reisekosten der Mitarbeiter gespeichert sind:

RechnungsNr	Datum	Alter	Name	Vorname	
1000	1.1.2004	3	Müller	Tim	
1000	1.1.2004	3	Müller	Tim	...
1001	2.1.2004	2	A.	B.	

	Anschrift	Kostenart	Anzahl	Vergütung
...	Olshausenstrasse 3, 24118, Kiel	Essen	2	20
...	Olshausenstrasse 3, 24118, Kiel	Bahn	2	100
	...			

Der Primärschlüssel ist {RechnungsNr, Kostenart}, somit ist es auch ein Kandidatenschlüssel. Im Folgenden bringen wir unsere Tabelle schrittweise in eine höhere Normalform.

0.NF In unserer Tabelle wird Alter aus Datum berechnet – Redundanz, außerdem muss jeden Tag das Alter neu berechnet werden. Lösung: Alter streichen.

1.NF In unserer Tabelle kann man Anschrift als zusammengesetzt betrachten – sie besteht aus Strasse, PLZ, Ort. Problem: An die einzelnen Bestandteile der Anschrift kommt man nur schwer an. Lösung: Anschrift in mehrere Attribute aufspalten.

2.NF Datum, Name, Vorname, Straße, Ort hängen nur von RechnungsNr ab, sind also Nichtschlüsselattribute, die von einer echten Teilmenge eines Kandidatenschlüssels abhängen. Probleme:

[14] Auch 3 1/2. Normalform genannt

– Redundanz, da Datum, Name, ... für jede Kostenart einer Dienstreise immer neu abgespeichert werden müssen.
– Inkonsistenz: Zwei Tupel derselben Dienstreise können mit unterschiedlichen Daten gespeichert werden.

Lösung: Die Relation aufspalten (Primärschlüssel unterstrichen):

Reisen = {RechnungsNr, Datum, Name, Vorname, Straße, PLZ, Ort}

Kostenarten = {Kostenart, Einzelvergütung}

Positionen = {RechnungsNr, Kostenart, Anzahl}

3.NF Es gilt {Name, Vorname} → {Straße, PLZ, Ort}, falls man davon ausgeht, dass die Namen eindeutig sind, und {PLZ} → {Ort}[15]. Problem: Redundanz und Integritätsverletzungen. Lösung: Aufspalten der Relation:

Reisen = {RechnungsNr, Datum, Name, Vorname}

Personal = {Name, Vorname, Straße, PLZ}

Orte = {PLZ, Ort}

Kostenarten wie oben

Positionen wie oben

BCNF Für ein Beispiel betrachte abgewandelte Positionen-Tabelle:

Positionen = {RechnungsNr, Kostenart, KostenartNr, Anzahl}

Dabei sei Kostenart von KostenartNr abhängig. Lösung: aufspalten

Positionen = {RechnungsNr, KostenartNr, Anzahl}

Kostenarten = {KostenartNr, Kostenart}

A.3.1 Kanonische Überdeckung

Sei F eine Menge von FDs über einer Attributmenge α.

Definition: F_C ist eine kanonische Überdeckung von α unter F, wenn folgende Bedingungen erfüllt sind:

1. $F_C^+ = F^+$ (das Wesen von F wird nicht verändert) [16]

[15] Wir sehen davon ab, dass es Orte mit gleicher PLZ gibt.

2. Überdeckung besitzt keine doppelten linken Seiten:

$$\forall(\beta_1 \to \gamma_1), (\beta_2 \to \gamma_2) \in F_C \colon \beta_1 = \beta_2 \Rightarrow \gamma_1 = \gamma_2$$

3. Überdeckung ist linksminimal:

$$\forall(\beta \to \gamma) \in F_C, \forall b \in \beta \colon \\ ((F_C \setminus \{(\beta \to \gamma)\}) \cup \{(\beta \setminus \{b\} \to \gamma)\})^+ \neq F^+$$

und rechtsminimal:

$$\forall(\beta \to \gamma) \in F_C, \forall c \in \gamma \colon \\ ((F_C \setminus \{(\beta \to \gamma)\}) \cup \{(\beta \to \gamma \setminus \{c\})\})^+ \neq F^+$$

Algorithmus zur Berechnung der kanonischen Überdeckung:

0. Setze $F_C := F$
1. Linksreduktion: Überprüfe für alle $(\beta \to \gamma) \in F_C$ und $b \in \beta$, ob beim Entfernen von b aus β die Hülle unverändert bleibt. Ist dies der Fall, so entferne b aus β.
2. Rechtsreduktion — analog zur Linksreduktion.
3. Entferne FDs mit leeren rechten Seiten
4. Fasse FDs mit gleichen linken Seiten zusammen.

$$\beta \to \gamma_1, \beta \to \gamma_2 \rightsquigarrow \beta \to \gamma_1 \cup \gamma_2$$

A.3.2 Forderungen an eine Zerlegung

Eine Überdeckung soll **verlustfrei** und **abhängigkeitserhaltend** sein.

Definitionen:

- Eine Zerlegung $R = R_1 \cup R_2$ einer Relation R ist *verlustfrei*, wenn für jede Klasse R^C von R gilt: $R^C = R_1^C \bowtie R_2^C$.
- Eine Zerlegung $R = R_1, \ldots, R_n$ ist *abhängigkeitserhaltend*, wenn für

[16] Unter F^+ verstehen wir hier die Menge aller aus F ableitbaren Regeln.

Mengen F_{R_i} von funktionalen Abhängigkeiten gilt

$$F_R = F_{R_1} \cup \ldots \cup F_{R_n}$$

Bemerkung: Eine Zerlegung $R = R_1 \cup R_2$ ist verlustfrei genau dann, wenn gilt

$$(R_1 \cap R_2 \rightarrow R_1) \vee (R_1 \cap R_2 \rightarrow R_2)$$

Beispiele:

- Beispiel einer nicht verlustfreien Zerlegung: Betrachte die Relation

Laden	Produkt	Kunde
MediaMarkt	DVD-Player	Rüdiger
MediaMarkt	Digicam	Wolfgang
Saturn	Digicam	Rüdiger

  Sei die Zerlegung wie folgt:

Laden	Produkt
MediaMarkt	DVD-Player
MediaMarkt	Digicam
Saturn	Digicam

Laden	Kunde
MediaMarkt	Rüdiger
MediaMarkt	Wolfgang
Saturn	Rüdiger

  Das Ergebnis des natürlichen Verbunds wäre dann:

Laden	Produkt	Kunde
MediaMarkt	DVD	Rüdiger
MediaMarkt	DVD	Wolfgang
MediaMarkt	Digicam	Rüdiger
MediaMarkt	Digicam	Wolfgang
Saturn	Digicam	Rüdiger

- Beispiel einer nicht abhängigkeitserhaltenden Zerlegung: Betrachte die Relation {Filiale, Kunde, Berater}. Es sollen folgende Abhängigkeiten gelten:
  - {Berater} → {Filiale}
  - {Kunde, Filiale} → {Berater}

  Mit der Zerlegung {Kunde, Berater} und {Filiale, Berater} kann nun folgendes passieren:

Kunde	Berater
A	B
A	C

Filiale	Berater
D	B
D	C

A.3.3 Synthesealgorithmus für 3NF

Eingabe: Relation R und eine Menge F von FDs. **Ausgabe:** Relationen $R_1, \ldots, R_n$ mit $R_1 \cup \ldots \cup R_n = R$ mit der Eigenschaft, dass $R_1, \ldots, R_n$ in 3NF sind und die Zerlegung verlustfrei und abhängigkeitserhaltend ist.

1. Bestimme die kanonische Überdeckung F_C von F.
2. Für jede FD $(\alpha \rightarrow \beta) \in F_C$ erstelle eine neue Relation $R_\alpha = \alpha \cup \beta$.
3. Falls keine der erstellten Relationen einen Kandidatenschlüssel von R enthält, wähle Kandidatenschlüssel $K \subseteq R$ und erstelle neue Relation $R_K = K$.
4. Eliminiere alle Relationen, die in anderen enthalten sind.

Beispiel: Betrachte die Relation

Einkauf(Anbieter, <u>Ware</u>, WGruppe, <u>Kunde</u>, KOrt, KLand, <u>Kaufdatum</u>)

Funktionale Abhängigkeiten (schon in kanonischer Überdeckung):

{Kunde, WGruppe} → {Anbieter}

{Anbieter} → {WGruppe}

{Kunde} → {KOrt}

{KOrt} → {KLand}

Schritt 1:

A(<u>Kunde</u>, <u>WGruppe</u>, Anbieter)

B(<u>Anbieter</u>, WGruppe)

C(<u>Kunde</u>, KOrt)

D(<u>KOrt</u>, KLand)

Schritt 2: es entsteht eine zusätzliche Relation

E(<u>Ware</u>, <u>Kunde</u>, <u>Kaufdatum</u>)

Schritt 3: Relation B eliminieren, da sie in A enthalten ist.

A.3.4 Dekompositions-/Analysealgorithmus für BCNF

Eingabe: Relation R, eine Menge F von FDs. **Ausgabe:** Relationen $R_1, \ldots, R_n$ mit $R_1 \cup \ldots \cup R_n = R$ mit $R_1, \ldots, R_n$ in BCNF, verlustfrei, *nicht immer* abhängigkeitserhaltend!

1. Setze $Z = \{R\}$
2. Solange es $R_i \in Z$ gibt, das noch nicht in BCNF ist:
   (a) Finde FD $(\alpha \rightarrow \beta)$ in R_i mit $\alpha \cap \beta = \emptyset$ und $\alpha \not\rightarrow R_i$
   (b) Zerlege R_i in $R_{i1} = \alpha \cup \beta$ und $R_{i2} = R_i \setminus \beta$
   (c) Ersetze R_i in Z durch R_{i1} und R_{i2}

Beispiel: Betrachte die Relation

Städte(Ort, BLand, Ministerpräsident, Einwohnerzahl)

Mit den funktionalen Abhängigkeiten

{BLand} → {Ministerpräsident}

{Ort, Bundesland} → {Einwohnerzahl}

{Ministerpräsident} → {Bundesland}

Mit $\alpha =$ BLand, $\beta =$ Ministerpräsident werden folgende Relationen gebildet:

(Ministerpräsident, BLand)

(Ort, BLand, Einwohnerzahl)

Somit ist die Zerlegung schon in der BCNF. [17]

A.4 Transaktionen

Definition: Eine *Transaktion* ist eine Bündelung logisch zusammenhängender Operationen. Die Transaktion kann mit `COMMIT` oder `ROLLBACK` beendet werden. Bei `COMMIT` werden die Änderungen beibehalten, bei `ROLLBACK` zurückgesetzt. Die Transaktionen werden entweder voll wirksam oder gar nicht wirksam – das Prinzip „Ganz oder gar nicht“.

Wir betrachten hier die atomaren Operationen `read` und `write`, die tupelweise lesen und nicht unterbrechbar sind.

[17] Den Schlüssel kann man mit dem Algorithmus wohl nicht bestimmen.

Definition: Ein *Schedule* ist eine Reihenfolge der Ausführung von Operationen, die zu den Transaktionen gehören.

Beispiel: Betrachte den folgenden Schedule:

BEGIN_TRANS	BEGIN_TRANS
x_1 = read(Kontostand)	
x_1 := $x_1 - 100$	
	x_2 := read(Kontostand)
	x_2 := $x_2 + 500$
	write(Kontostand, x_2)
write(Kontostand, x_1)	
ROLLBACK	COMMIT

Dieser Schedule führt zu einem Problem (hier ein „Lost Update“). Um ähnliche Probleme algorithmisch zu entdecken, benutzt man den Konfliktgraphen.

Definition: Zwei Operationen x, y stehen in *Konflikt*[18] zueinander genau dann, wenn

- x und y zu zwei verschiedenen Transaktionen T_i, T_j gehören und
- mindestens eine der Operationen ein Schreibvorgang ist und
- kein commit zwischen den beiden Operationen liegt

Definition: Ein *Konfliktgraph* zu einem Schedule ist ein gerichteter Graph mit:

- Transaktionen als Knoten
- Für jedes Paar $x \in T_i, y \in T_j$ von Operationen, die in Konflikt stehen eine Kante (T_i, T_j), falls x vor y ausgeführt wird.

Satz: Ist der Konfliktgraph azyklisch, so ist der entsprechende Schedule serialisierbar, d.h. es existiert ein dazu konfliktäquivalenter serieller Schedule. Eine topologische Ordnung auf dem Graphen liefert einen äquivalenten seriellen Schedule.

Beispiel: Betrachte folgende Transaktionen

$$T_1 = r_1(A), w_1(C), w_1(B)$$

$$T_2 = r_2(A), r_2(B), w_2(D)$$

$$T_3 = r_3(C), w_3(A)$$

[18] Dies ist die sog. *pessimistische* Definition der Konfliktäquivalenz.

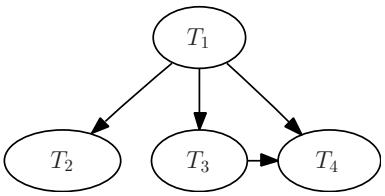
$$T_4 = r_4(B), r_4(C), w_4(B), w_4(A)$$

Sei der Schedule wie folgt (r, w bedeuten „Lesen“ bzw. „Schreiben“):

$$r_2(A), r_1(A), w_1(C), r_3(C), w_1(B), r_4(B),$$

$$r_4(C), r_2(B), w_3(A), w_4(B), w_2(D), w_4(A)$$

Aus $w_1(C)$ vor $r_3(C)$ folgt die Kante $T_1 \to T_3$, aus $w_1(C)$ vor $r_4(C)$ folgt $T_1 \to T_4$ usw. Es ergibt sich der folgende Graph:



Dieser Graph ist azyklisch. Eine topologische Ordnung wäre T_1, T_2, T_3, T_4, was einen äquivalenten seriellen Schedule liefert.

Bemerkung: Serialisierbare Schedules sind nicht immer abbrechbar, d.h. es können trotzdem Probleme beim `Rollback` auftreten.

A.5 Mehrwertige Abhängigkeiten

Definition: Sei R^C eine Klasse über einer Relation R. Seien $\alpha, \beta \subseteq R$. Wir sagen, β *hängt mehrwertig von* α *ab*, i.Z. $\alpha \twoheadrightarrow \beta$, wenn gilt

$$\begin{aligned}\forall t_1, t_2 \in R^C : (t_1(\alpha) = t_2(\alpha)) \Longrightarrow (\exists t_3, t_4 \in R^C : \\ t_3(\alpha) = t_4(\alpha) \wedge t_3(\alpha) = t_2(\alpha) \\ \wedge\, t_3(\beta) = t_1(\beta) \wedge t_4(\beta) = t_2(\beta) \\ \wedge\, t_3(R \setminus \beta) = t_2(R \setminus \beta) \\ \wedge\, t_4(R \setminus \beta) = t_1(R \setminus \beta))\end{aligned}$$

Informell: finde ich 2 Tupel mit gleichen α-Werten, so sind die beiden Tupel, die dadurch entstehen, dass die β-Werte der ursprünglichen Tupel vertauscht werden, auch in der Relation enthalten.

Mehrwertige Abhängigkeiten entstehen, wenn Daten in einer Relation gespeichert werden, die in „keiner Beziehung zueinander stehen“.

Definition: Eine mehrwertige Abhängigkeit $\alpha \twoheadrightarrow \beta$ heißt *trivial* genau dann, wenn $\beta \subseteq \alpha$ oder $\beta = R \setminus \alpha$ ist.

Definition: Ein ER-Typ (R, Σ) ist in der *4. Normalform (4NF)*, falls für nichttriviale $X \twoheadrightarrow Y \in \Sigma^+$ gilt, dass X Schlüssel von (R, Σ) ist.

Beispiel: Die folgende Relation ist nicht in 4. Normalform:

Kette	Filiale	Produkt
MediaMarkt	HH	Kühlschrank
MediaMarkt	Kiel	TV
MediaMarkt	Kiel	Kühlschrank
MediaMarkt	HH	TV
Saturn	HH	TV
Saturn	HH	PC
Saturn	Berlin	TV
Saturn	Berlin	PC

Eine Zerlegung der Relation in 4. Normalform wäre:

{Kette, Filiale}

{Kette, Produkt}

Satz: Eine Zerlegung $R = R_1 \cap R_2$ ist verlustlos genau dann, wenn

$$(R_1 \cap R_2) \twoheadrightarrow R_1 \text{ oder } (R_1 \cap R_2) \twoheadrightarrow R_2$$

B Ausgewählte Übungsaufgaben

Folgendes sind die HiWi-Lösungen von einigen Übungsaufgaben.

Serie 7

3. Gesucht ist der größte, der kleinste Folder und die durchschnittliche Größe der Folder, gruppiert nach Benutzern

```
SELECT ID, Name, FMax, FMin, AvgSize
FROM (SELECT ID, FolderID, FName AS FMax, MaxSize
      FROM Folder)
     CROSS JOIN
     (SELECT ID AS ID2, FName AS FMin,
             FolderID as FID2, MaxSize AS MinSize
      FROM Folder)
     CROSS JOIN
     (SELECT ID AS ID3, Name, MAX(MaxSize) AS MaxS,
             MIN(MaxSize) AS MinS,
             AVG(MaxSize) AS AvgSize
      FROM Folder NATURAL JOIN Users
      GROUP BY ID, Name)
WHERE ID = ID2 AND
      ID2 = ID3 AND
      MaxSize = MaxS AND
      MinSize = MinS
```

Serie 8

1d) Welche Emails haben mindestens zweimal das gleiche File als Attachment?

$$\Pi_{\mathsf{EmailID}}(\sigma_{\mathsf{DateOfSave} \neq \mathsf{DoS2} \,\wedge\, \mathsf{FileID}=\mathsf{FID2} \,\wedge\, \mathsf{EmailID}=\mathsf{EID2}} (\mathsf{Attachment} \times \rho_{\substack{\mathsf{DateOfSave}\to\mathsf{DoS2},\\ \mathsf{FileID}\to\mathsf{FID2},\\ \mathsf{EmailID}\to\mathsf{EID2}}}(\mathsf{Attachment})))$$

1f) Welche Emails hatten alle FileTypen als Anhang?

$$\Pi_{\mathsf{EmailID}}(\mathsf{Attachment}) \setminus \Pi_{\mathsf{EmailID}}((\Pi_{\mathsf{EmailID},\ \mathsf{FileType}}(\mathsf{Attachment} \times \mathsf{File})) \setminus (\Pi_{\mathsf{EmailID},\ \mathsf{FileType}}(\mathsf{Attachment} \bowtie \mathsf{File})))$$

1g) Welche Emails haben Attachments, die in der Summe der Speichergröße größer als 300 MB sind? Es ist kein rationaler Ausdruck möglich — `GROUP BY` kann man nicht abbilden.

1h) Welcher FileTyp wurde insgesamt am häufigsten als Attachment gespeichert?

```
SELECT Filetype
FROM (SELECT Filetype, COUNT(Filetype) AS C
      FROM Attachment NATURAL JOIN
           Files
      GROUP BY Filetype)
WHERE C =
      (SELECT MAX(C2)
      FROM (SELECT Filetype, COUNT(Filetype) AS C2
            FROM Attachment NATURAL JOIN Files
            GROUP BY Filetype))
```

2 Abbildung des ER-Schemas im DDL-Script. Hier wird kein DDL-Script angegeben, sondern nur eine schematische Darstellung. Dies müsste noch in einen SQL-artigen Text überführt werden.

SpecialPraedikat(p'ID, pID → Praedikat(pID));

Praedikat(pID);

Email(eID);

Folder(fID, eID → Email, z);

Save(eID → Email, spID → SpecialPraedikat, fID → Folder);

Literatur

[Böra] Egon Börger. Abstract State Machine Tutorial. http://www.di.unipi.it/~boerger/ASMTutorialEtaps.html.

[Börb] Egon Börger. Abstract State Machines. www.di.unipi.it/~boerger/Papers/Methodology/AsmDefinition.PDF.

[BS76] N.D. Belnap and T.B. Steel. *The Logic of Questions and Answers*. Yale University Press, 1976.

[Cha98] Don Chamberlin. *A Complete Guide to DB2 - Universal Database*. Morgan Kaufmann, 1998.

[KE97] Alfons Kemper and André Eickler. *Datenbanksysteme – eine Einführung*. R. Oldenbourg Verlag München Wien, 2. auflage edition, 1997.

[Kel04] Andreas Kelz. 4.6.2 Mehrwertige Abhängigkeit (Multivalued Dependency, MVD). http://v.hdm-stuttgart.de/~riekert/lehre/db-kelz/chap4.htm, 2004.

[Kle04] Hans-Joachim Klein. Einführung in SQL. http://www.is.informatik.uni-kiel.de/~hjk/DBS1/, 2004.

[Lin04] Tilman Linneweh. Definitionen von Abhängigkeiten. http://www.arved.de/refa/klassen.html, 2004.

[Stä] Robert F. Stärk. Detailed Definition of ASMs. www-madlener.informatik.uni-kl.de/ag-madlener/teaching/ss2004/fsvt/30.04.04.main.4.pdf.

[Tha04] Bernd Thalheim. Datenbanksysteme 1 – Script zur Vorlesung. http://www.is.informatik.uni-kiel.de/~thalheim/vorlesungen/DB1/DB1.html, 2004.

[Wik04] Wikipedia. ANSI-SPARC-Architektur. http://de.wikipedia.org/wiki/ANSI-SPARC-Architektur, 2004.

Index