Datenmodellierung und Datenbanken

Zusammenfassung der Vorlesung im Frühlingssemester 2010 von Donald Kossmann

Stefan Heule

2010-08-05

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation für den Einsatz eines DBMS

Datenbankverwaltungssysteme (*database management system, DBMS*) lösen verschiedene Probleme:

- Redundanz und Inkonsistenz
- Beschränkte Zugriffsmöglichkeiten
- Probleme des Mehrbenutzerbetriebs
- Verlust von Daten
- Integritätsverletzungen
- Sicherheitsproblem
- Hohe Entwicklungskosten

1.2 Datenabstraktion

Man unterscheidet drei Abstraktionsebenen:

- *Die physische Ebene*: Auf dieser Ebene wird festgelegt, wie die Daten gespeichert sind. Im Allgemeinen sind die Daten auf dem Hintergrundspeicher abgelegt. Für den normalen User nicht von Bedeutung, hier werden die Speicherstrukturen und eventuell Indexstrukturen für das schnelle Auffinden von Daten festgelegt.
- *Die logische Ebene*: Auf der logischen Ebene wird in einem sogenannten *Datenbankschema* festgelegt, welche Daten abgespeichert werden.
- *Die Sichten*: Während das Datenbankschema ein integriertes Modell der gesamten Informationsmenge darstellt, werden in den Sichten Teilmengen bereitgestellt.

1.3 Datenunabhängigkeit

Die drei Ebenen eines DBMS gewährleisten einen bestimmten Grad der *Datenunabhängigkeit*, analog zum Konzept eines ADTs. Durch eine wohldefinierte Schnittstelle wird die darunterliegende Implementierung verdeckt, und kann diese – bei Beibehaltung der Schnittstelle – beliebig ändern.

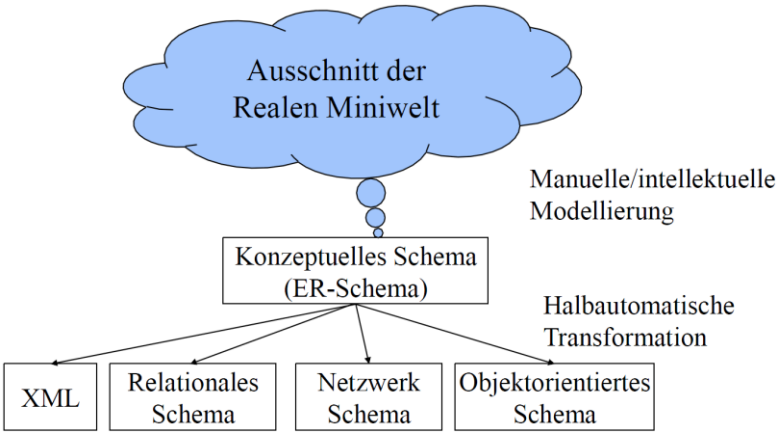
- *Physische Datenunabhängigkeit*: Die Modifikation der physischen Speicherstrukturen belässt die logische Ebene invariant. Zum Beispiel erlauben fast alle Datenbanksysteme das nachträgliche Anlegen eines Indexes.
- *Logische Datenunabhängigkeit*: In den Anwendungen wird (natürlich) Bezug auf die logische Struktur der Datenbasis genommen: Es werden Mengen von Datenobjekten nach einem Namen angesprochen und deren Attribute referenziert. Bei Änderungen auf logischer Ebene können z.B. Attribute umbenannt werden. In einer Sichtendefinition kann man solche kleineren Änderungen verbergen und erzielt dadurch einen gewissen Grad an logischer Datenunabhängigkeit.

Heute erfüllen Datenbanksysteme die physische Datenunabhängigkeit. Die logische Datenunabhängigkeit kann schon rein konzeptuell nur für einfachste Modifikationen des Datenbankschemas gewährleistet werden.

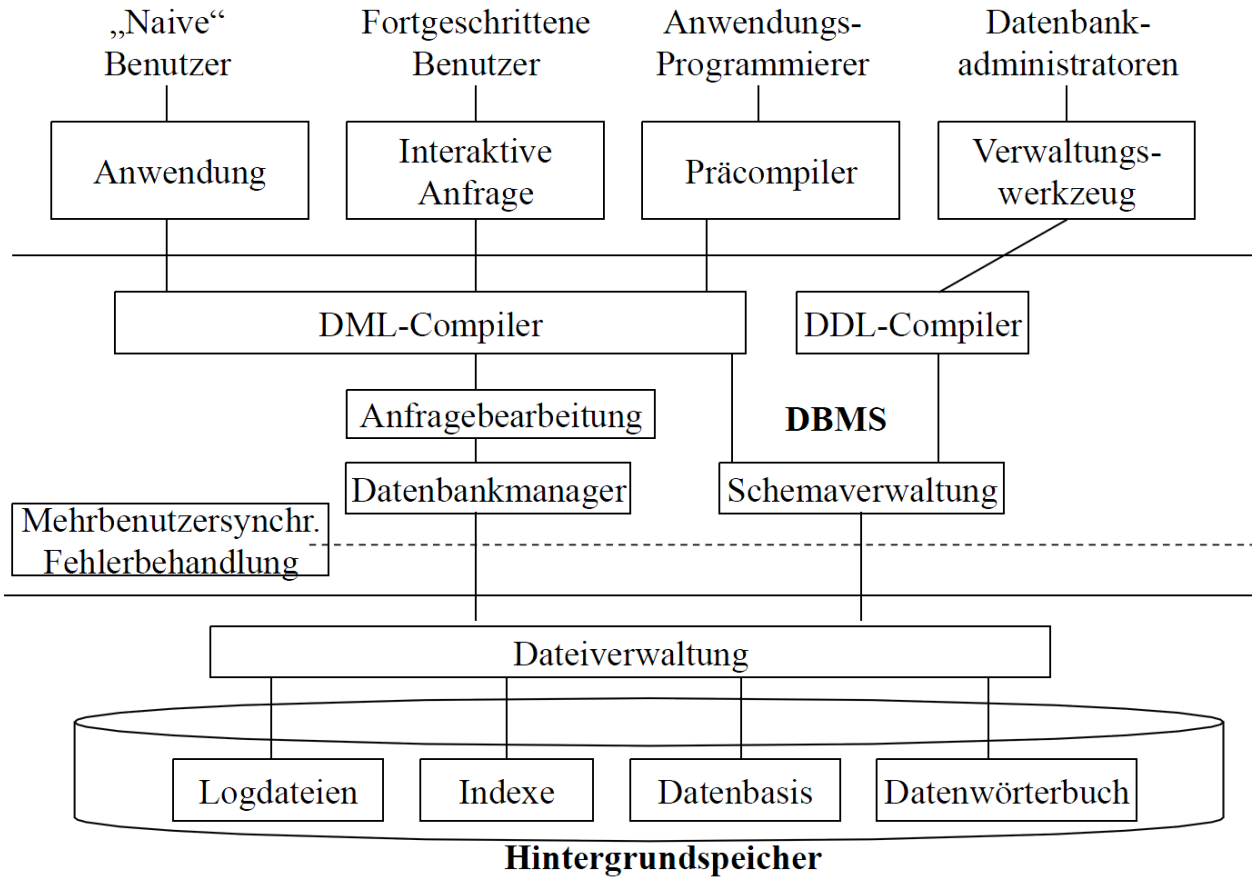
1.4 Datenbankschema und Ausprägung

Es muss klar zwischen dem *Datenbankschema* und einer *Datenausprägung* unterscheiden. Ersteres legt die Struktur der abspeicherbaren Datenobjekte fest. Beim Datenbankschema handelt es sich also um Metadaten. Unter der Datenbankausprägung hingegen versteht man den momentan gültigen (also abgespeicherten) Zustand der Datenbasis.

1.5 Phasen der Datenmodellierung



1.6 Architekturübersicht eines DBMS



2 Datenbankentwurf

2.1 Abstraktionsebenen des Datenbankentwurfs

Beim Entwurf kann man drei Abstraktionsebenen unterscheiden:

- *Konzeptuelle Ebene*: Hier wird der Anwendungsbereich – unabhängig vom eingesetzten Datenbanksystem – strukturiert. Es werden Gegenstände zu Gegenstandsmengen und Beziehungen zwischen den Gegenständen zu Beziehungstypen abstrahiert. Für den konzeptuellen Entwurf wird oft das *Entity-Relationship-Modell* eingesetzt.
- *Implementationsebene*: Auf dieser Ebene wird die Datenbankanwendung in den Konzepten (d.h. in dem Datenmodell) des zum Einsatz kommenden Datenbanksystems modelliert. Beim relationalen Datenmodell hat man es hierbei mit Relationen, Tupeln und Attributen zu tun.
- *Physische Ebene*: Dies ist die „niedrigste" Abstraktionsebene und es geht primär darum, die Leistungsfähigkeit zu erhöhen. Die zu betrachtenden Strukturen sind z.B. Datenblöcke, Zeiger und Indexstrukturen.

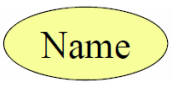
2.2 Das Entity-Relationship-Modell

Die Grundlegenden Modellierungsstrukturen dieses Modells sind Entitäten und Beziehungen dazwischen.

- **Entitäten** sind wohlunterscheidbare physisch oder gedanklich existierende Konzepte der zu modellierenden Welt. Man abstrahiert ähnliche Gegenstände zu Gegenstandstypen, die man grafisch als Rechtecke darstellt, wobei der Name des Typs innerhalb des Rechtecks angegeben wird.

- **Beziehungen** werden analog zu Beziehungstypen zwischen den Gegenstandstypen abstrahiert und als Rauten mit entsprechender Beschriftung dargestellt. Eine Beziehung kann zwischen beliebig vielen Entitäten stattfinden, und es können optional *Rollen* der Entitäten benannt werden (z.B. „Vorgänger" und „Nachfolger" bei der Beziehung „voraussetzen" von zwei Vorlesungen).

- **Attribute** dienen dazu, Gegenstände und Beziehungen weiter zu charakterisieren. Sie werden durch Ovale grafisch dargestellt. Eine minimale Menge von Attributen, deren Werte die zugeordnete Entität eindeutig innerhalb aller Entitäten seines Typs identifiert, nennt man *Schlüssel*. Diese werden durch Unterstreichen optisch gekennzeichnet.



2.2.1 Charakterisierung von Beziehungstypen

Ein Beziehungstyp R zwischen Entitäten E_1, E_2, …, E_n kann als Relation im mathematischen Sinne verstanden werden, also

$$R \subseteq E_1 \times E_2 \times \cdots \times E_n$$

In diesem Fall bezeichnet man n als den Grad der Beziehung, wobei binäre Beziehungen mit Abstand am Häufigsten vorkommen.

So lässt sich nun auch der Begriff Rolle genauer fassen. Betrachtet man die Beziehung „voraussetzen", so gilt

$$\text{voraussetzen} \subseteq \text{Vorlesungen} \times \text{Vorlesungen}$$

Eine einzelne Instanz (v_1, v_2) wird nun durch die Rollen genauer charakterisiert, und zwar wie folgt:

$$(\text{Vorgänger: } v_1, \text{Nachfolger: } v_2)$$

2.2.1.1 Funktionalitäten von binären Beziehungen

Es gibt vier Typen von *Funktionalitäten*:

- *1:1-Beziehung*, falls jeder Entität e_1 aus E_1 höchstens eine Entität e_2 aus E_2 zugeordnet ist, und umgekehrt. Es ist zu beachten, dass nicht jede Entität einen „Partner" haben muss.
- *1:N-Beziehung*, falls jeder Entität e_1 aus E_1 beliebig viele Entitäten e_2 aus E_2 zugeordnet sein können, aber jede Entität e_2 aus E_2 höchstens einer aus E_1 zugewiesen sein kann.
- *N:1-Beziehung*, analog.
- *N:M-Beziehung*, falls es keine Einschränkungen gibt.

Solche Funktionalitäten stellen Integritätsbedingungen dar, die in der zu modellierenden Welt immer gelten müssen.

2.2.1.2 Funktionalitäten von n-stelligen Relationen

Das Konzept der Funktionalitäten kann auf n-stellige Relationen erweitert werden. Sei R also eine mehrwertige Beziehung mit Entitäten E_1, E_2, …, E_n. Bei einer Entität E_k steht nun genau dann eine „1", wenn durch R folgende partielle Funktion vorgegeben wird:

$$R: E_1 \times \ldots \times E_{k-1} \times E_{k+1} \times \ldots \times E_n \to E_k$$

2.2.1.3 Die (min, max)-Notation

Neben den Funktionalitäten gibt es noch einen weiteren Formalismus, mit welchem man etwas über Beziehungen aussagen kann.

Bei der $(\min, \max)$-Notation wird für jede Entität einer Relation ein Paar von Zahlen angegeben. Dieses Zahlenpaar sagt aus, dass jede Entität dieses Typs mindestens $\min$-mal in der Beziehung steht, und höchstens $\max$-mal.

In einer Relation $R \subseteq E_1 \times E_2 \times \cdots \times E_n$ sagt die Angabe (min_i, max_i) aus, dass es für alle $e_i \in E_i$ mindestens min_i und höchstens max_i Tupel der Art $(\ldots, e_i, \ldots)$ geben darf. Ein Stern * wird verwendet, wenn die Entität beliebig oft vorkommen darf.

Für binäre Relationen ist diese Notation mächtiger, bei n-stelligen Beziehungen mit $n > 2$ sind die beiden Formalismen nicht mehr vergleichbar, es gibt für beide Seiten Beispiele, die sich mit der anderen nicht ausdrücken lassen.

2.2.1.4 Multiplizitäten in UML

In UML gibt es Multiplizitäten, um Beziehungen (die in UML gerichtet sein können) genauer zu charakterisieren. Diese orientieren sich an den Funktionalitäten, sind aber (ähnlich wie die $(\min, \max)$-Notation) genauer, indem ein Bereich $k..l$ angegeben werden kann, oder * für einen beliebigen Wert.

2.2.2 Existenzabhängige Entitätstypen

Es gibt sogenannt *existenzabhängige* oder *schwache Entitäten*, die

- in ihrer Existenz von einer andere, übergeordneten Entität abhängig sind, und
- oft nur in Kombination mit dem Schlüssel der übergeordneten Entität eindeutig identifizierbar.

Schwache Entitäten werden durch doppelt umrandete Rechtecke repräsentiert. Die Beziehung zur übergeordneten Entität wird ebenfalls doppelt gezeichnet, inklusive der Raute. Eine solche Relation kann nur vom Typ 1:N oder seltener 1:1 sein. Da die Schlüssel solche Entitäten nur in Zusammenhang mit dem Schlüssel der übergeordneten Entität eindeutig sind, wird der Schlüssel gestrichelt unterstrichen.

2.2.3 Generalisierung

Die *Generalisierung* wird im konzeptuellen Entwurf eingesetzt, um eine bessere, d.h. natürlichere und übersichtlichere Strukturierung der Entitäten zu erzielen. Dabei werden die Eigenschaften ähnlicher Entitätstypen (also Attribute und Beziehungen) einem gemeinsamen *Obertyp* zugeordnet. Eigenschaften, die für einen *Untertyp* speziell sind, bleiben bei diesem.

Entität eines Untertyps erben sämtliche Eigenschaften eines Obertyps, und wird implizit auch als Entität (Element) des Obertyps betrachtet.

Diesen Sachverhalt wird durch eine spezielle *is-a* Beziehung modelliert, wobei oft ein eigenes Symbol verwendet wird, z.B. ein Sechseck.

2.2.4 Aggregation

Bei der Aggregation werden unterschiedliche Entitäten, die in ihrer Gesamtheit einen strukturierten Objekttypen bilden, einander zugeordnet. Dies wird ebenfalls durch eine Beziehung verdeutlicht, die sogenannte *part-of* Beziehung. Sie betont, dass die untergeordneten Entitäten Teile (also Komponenten) der übergeordneten (zusammengesetzten) Entitäten sind.

2.2.5 Konsolidierung

Da die zu modellierende Welt oftmals komplex ist, werden von verschiedenen Personen die über einen Teil dieser Welt besonders gut Bescheid wissen, eine Sicht erstellt. Diese Sichten beschreiben natürlich nicht disjunkte Ausschnitte der realen Welt, sondern überschneiden sich vielmehr. Daher müssen die Sichten zu einem redundanzfreien, globalen Schema vereint werden. Dieser Vorgang nennt sich *Konsolidierung* oder *Sichtenintegration*.

Das so entstehende Schema muss *redundanz-* und *widerspruchsfrei* sein. Weiter sollen sowohl *Synonyme* (gleiche Sachverhalte wurden verschieden benannt) als auch *Homonyme* (verschiedene Sachverhalte wurden gleich benannt) bereinigt werden.

3 Das relationale Modell

3.1 Definition des relationel Modells

3.1.1 Mathematischer Formalismus

Gegeben seien n nicht notwendigerweise unterschiedliche *wertebereiche* (auch *Domänen* genannt) $D_1, D_2, \dots D_n$. Diese Wertebereiche dürfen nur atomare Werte enthalten, also insbesondere keine Mengen. Eine Relation R ist nun definiert als eine Teilmenge des kartesischen Produktes der n Domänen:

$$R \subseteq D_1 \times \cdots \times D_n$$

3.1.2 Schema-Definition

Neben dem Typ (Domäne) ist im Datenbankebereich auch noch der Name der Komponenten der Tupel wichtig. Relationsschemata werden nach folgendem Muster spezifiziert:

Telefonbuch: {[Name: string, Adresse: string, Telefonnummer: integer]}

Wie bei ER-Diagrammen wird der Schlüssel unterstrichen. Die eckigen Klammern [] stehen hierbei für den Tupelkonstruktor, während die geschwungenen Klammern {} verdeutlichen, dass es sich hierbei um eine Menge von Tupeln handelt.

Mit $\mathrm{sch}(R)$ oder $\mathcal{R}$ wird die Menge der Attribute einer Relation bezeichnet, und mit $\mathrm{dom}(A_i)$ der Wertebereich eines Attributs.

3.2 Umsetzung eines konzeptuellen Schemas in ein relationales Schema

In ER gibt es Entitäten und Beziehungen, während im relationalen Modell nur ein einziges Strukturierungskonzept – nämlich die Relation – verfügbar ist. Daher werden sowohl Entitäten als auch Beziehungen auf Relationen abgebildet.

3.2.1 Relationale Darstellung von Entitäten

Dies geschieht ganz natürlich, indem jedes Attribut einer Entität ein Attribut der Relation wird. Ebenso werden die Schlüssel direkt übernommen.

3.2.2 Relationale Darstellung von Beziehungen

Im initialen Entwurf wird für jeden Beziehungstyp eine eigene Relation definiert, einige können später jedoch wieder zusammengefasst werden. Eine solche Relation hat die folgende Gestalt:

$$R\colon \left\{ \left[\overbrace{A_{11}, \dots, A_{1k_1}}^{\text{Schlüssel von } E_1},\ \underbrace{A_{21}, \dots, A_{2k_2}}_{\text{Schlüssel von } E_2},\ \dots,\ \overbrace{A_{n1}, \dots, A_{nk_n}}^{\text{Schlüssel von } E_n},\ \underbrace{A_1^R, \dots, A_{k_R}^R}_{\text{Attribute von } R} \right] \right\}$$

Die Relation enthält also neben den Attributen der Beziehung selbst auch noch alle Schlüsselattribute der beteiligten Entitäten. Es ist möglich, dass gewisse Attribute umbenannt werden müssen, damit die Namen eindeutig bleiben.

3.3 Verfeinern des relationalen Schemas

Nun können alle Relationen, die denselben Schlüssel haben, zusammengefasst werden. Dies funktioniert insbesondere für 1:1, 1:N und N:1 Beziehungen.

3.4 Relationale Algebra

Neben der Strukturbeschreibung benötigt man natürlich auch eine Anfragesprache, mithilfe welcher man Informationen aus der Datenbank extrahieren kann. Dafür gibt es zwei formale Sprachen:

- Die relationale Algebra
- Den Relationenkalkül

Beide Sprachen sind *abgeschlossen*, d.h. die Ergebnisse der Anfragen sind wiederum Relationen, und können weiter verwendet werden.

3.4.1 Selektion

Bei der *Selektion* werden diejenigen Tupel einer Relation ausgewählt, die das sogenannte *Selektionsprädikat* erfüllen. Die Selektion wird mit σ bezeichnet, wobei das Selektionsprädikat als Subskript geschrieben wird; zum Beispiel

$$\sigma_{\text{Semester}<10}(\text{Studenten})$$

3.4.2 Projektion

Während bei der Selektion einzelne Zeilen (Tupel) einer Tabelle (Relation) ausgewählt werden, werden bei der *Projektion* Spalten (Attribute) der Argumentrelation extrahiert. Es wird Π als Symbol verwendet, mit der Menge der Attributnamen als Subskript, zum Beispiel:

$$\Pi_{\text{Alter}}(\text{Professor})$$

Bei dieser Operation werden Duplikate automatisch eliminiert, da wir auf Mengen von Tupeln arbeiten, und es da per Definition keine zwei identischen Tupel geben kann.

3.4.3 Vereinigung

Zwei Relationen mit demselben Schema - d.h. mit gleichen Attributnamen und Attributtypen – kann man durch die Vereinigung zu einer Relation zusammenfassen. Dies wird durch das mathematische Symbol für Vereinigung bezeichnet:

$$\text{Relation}_\text{A} \cup \text{Relation}_\text{B}$$

Wiederum wird eine Duplikatelimination durchgeführt.

3.4.4 Mengendifferenz

Für zwei Relationen mit gleichem Schema ist die Mengendifferenz

$$R - S$$

definiert als die Menge der Tupel, die in R aber nicht in S vorkommen.

3.4.5 Kartesisches Produkt

Das Kreuzprodukt zweier Relationen R und S wird als

$$R \times S$$

Gebildet und enthält alle $|R| * |S|$ möglichen Paare von Tupeln. Das entstehende Schema, also $\text{sch}(R \times S)$, ist die Vereinigung aller Attribute aus R und S, also

$$\text{sch}(R \times S) = \text{sch}(R) \cup \text{sch}(S)$$

Haben zwei Attribute aus R und S denselben Namen, wird die eindeutige Benennung dadurch erzwungen, dass dem Atrributnamen der Relationenname, gefolgt von einem Punkt, vorangestellt wird.

3.4.6 Umbenennung von Relationen und Attributen

Manchmal ist es notwendig, dieselbe Relation mehrfach in einer Anfrage zu verwenden. In diesem Fall muss mindestens eine Relation umbenannt werden, und zwar mithilfe des Operators ρ. Als Beispiel sieht die Umbenennung von voraussetzen in V1 folgendermassen aus:

$$\rho_{\mathrm{V1}}(\mathrm{voraussetzen})$$

Ganz ähnlich funktioniert auch das Umbenennen von Attributen:

$$\rho_{\mathrm{ProfID} \leftarrow \mathrm{PersonID}}(\mathrm{Professoren})$$

Hierbei wurde das Attribut PersonID von Professoren in ProfID umbenannt.

3.4.7 Definition der relationalen Algebra

Die bisher vorgestellten Operatoren sind ausreichend, um die relationale Algebra formal zu definieren. Allerdings ist es oft einfacher, noch weitere Operatoren zur Verfügung zu haben. Diese lassen sich aber alle durch die bisher eingeführten Operatoren ausdrücken.

3.4.8 Der relationale Verbund (Join)

Das kartesische Produkt bildet alle Tupelpaare aus zwei Relationen. Oftmals ist man jedoch nicht an all diesen Paaren interessiert, sondern nur an bestimmten. Dies wird durch sogenannte Joins erreicht.

3.4.8.1 Der natürliche Join

Der sogenannte *natürliche Join* zweier Relationen R und S wird mit $R \bowtie S$ gebildet. Wenn R insgesamt $m + k$ Attribute $A_1, \dots, A_m, B_1, \dots, B_k$ und S $n + k$ Attribute $B_1, \dots, B_k, C_1, \dots, C_n$ hat, dann hat $R \bowtie S$ die Stelligkeit $n + m + k$. Die Definition des natürlichen Joins ist wie folgt:

$$R \bowtie S = \Pi_{A_1,\dots,A_m,R.B_1,\dots,R.B_k,C_1,\dots,C_n}\left(\sigma_{R.B_1=S.B_1 \wedge \dots \wedge R.B_k=S.B_k}(R \times S)\right)$$

3.4.8.2 Der allgemeine Join

Beim natürlichen Join werden alle gleichbenannten Attribute der beiden Relationen betrachtet. Der allgemeine Join, auch *Theta-Join* genannt, erlaubt die Spezifikation eines beliebigen Joinprädikates θ und sieht wie folgt aus:

$$R \bowtie_\theta S$$

Das Ergebnis dieses Joins hat $m + n$ Attribute, unabhängig der Namen. Dieser Join lässt sich sehr leicht mittels kartesischem Produkt und einer Selektion ausdrücken:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

3.4.8.3 Weitere Join-Operatoren

Die bislang eingeführten Join-Operatoren nennt man manchmal auch „innere“ Joins. Bei diesen Operatoren gehen im Ergebnis diejenigen Tupel der Argumentrelationen verloren, die keinen „Joinpartner“ gefunden haben. Bei den *äusseren* Join-Operatoren hingegen, werden auch partnerlose Tupel der linken und/oder rechten Relation „gerettet“:

- Linker äusserer Join ($⟕$): Die Tupel der linken Argumentrelation bleiben in jedem Fall erhalten. Fehlt ein Joinpartner, werden die übrigen Attribute mit *null* gefüllt.
- Rechter äusserer Join ($⟖$): Die Tupel der rechten Argumentrelation bleiben in jedem Fall erhalten. Fehlt ein Joinpartner, werden die übrigen Attribute mit *null* gefüllt.
- (Vollständiger) äusserer Join ($⟗$): Die Tupel der beider Argumentrelation bleiben in jedem Fall erhalten. Fehlt ein Joinpartner, werden die übrigen Attribute mit *null* gefüllt.

Weiter gibt es den *Semijoin*, bei welchem das Resultat alle Tupel aus der einen Relation unverändert enthält, welche einen Joinpartner in der anderen Relation haben. Formal:

$$L \ltimes R = \Pi_{\mathcal{L}}(L \bowtie \mathrm{R})$$

Und analog dazu:

$$L \rtimes R = \Pi_{\mathcal{R}}(L \bowtie \mathrm{R})$$

3.4.9 Mengendurchschnitt

Der Mengendurchschnitt von R und S lässt sich leicht über die Mengendifferenz ausdrücken:

$$R \cap S = R - (R - S)$$

3.4.10 Die relationale Division

Der Divisionsoperator wird folgendermassen notiert: $R \div S$. Das Resultat enthält alle Tupel aus R (eingeschränkt auf die Attribute die nur in R vorkommen), für welche gilt, dass alle ihre Kombinationen mit Tupeln aus S in R vorhanden sind.

Konkret lässt sich damit also eine Allquantifizierung realisieren. Enthält R zum Beispiel die Namen von Studenten, zusammen mit den Projekten, die sie in diesem Semester bereits abgeschlossen haben, und R enthält die Namen aller Projekte aus dem Fach Datenbanken, so gilt: $R \div S$ enthält die Namen aller Studenten, die bereits sämtliche Datenbankprojekte abgeschlossen haben.

Offensichtlich macht die Division nur Sinn, wenn $\mathcal{S} \subseteq \mathcal{R}$ gilt, da das Resultat andernfalls sicher leer ist.

Der Divisionsoperator lässt sich wie folgt durch andere Operatoren ausdrücken:

$$R \div S = \Pi_{\mathcal{R}-\mathcal{S}}(\mathrm{R}) - \Pi_{\mathcal{R}-\mathcal{S}}\big((\Pi_{\mathcal{R}-\mathcal{S}}(R) \times S) - R\big)$$

3.5 Der Relationenkalkül

Ausdrücke in der relationalen Algebra spezifizieren, wie das Ergebnis der Anfrage zu berechnen ist. Demgegenüber ist der *Relationenkalkül* stärker *deklarativ* orientiert, d.h. es werden die qualifizierenden Ergebnistupel beschrieben, ohne dass eine Herleitungsvorschrift angegeben wird.

Der Relationenkalkül basiert auf dem mathematischen Prädikatenkalkül erster Stufe, der quantifizierte Variablen und Werte zulässt. Es gibt zwei unterschiedliche, aber gleich mächtige Ausprägungen des Relationenkalküls:

- Der relationale Tupelkalkül
- Der relationale Domänenkalkül

Der Unterschied besteht darin, dass Variablen des Kalküls im ersten Fall an Tupel einer Relation gebunden werden, und im zweiten Fall an Domänen, die als Wertemengen von Attributen vorkommen.

3.5.1 Anfragen im relationalen Tupelkalkül

Anfragen im relationalen Tupelkalkül haben folgende generische Form:

$$\{t \mid P(t)\}$$

Hierbei ist t eine sogenannte Tupelvariable und P ist ein *Prädikat*, das erfüllt sein muss, damit t in das Ergebnis aufgenommen wird. Die Variable t ist eine sogenannte freie Variable des Prädikates, d.h. t darf nicht durch einen Existenz- oder Allquantor quantifiziert sein. Eine konkrete Anfrage könnte dabei folgendermassen aussehen:

$$\{p \mid p \in \text{Professoren} \ \wedge p.\text{Name} = 'Test'\}$$

Mithilfe des Tupelkonstruktors [. .] lassen sich auch neue Tupel bauen, zum Beispiel so:

$$\{[t.A, s.B] \mid P(t, s)\}$$

3.5.1.1 Sichere Ausdrücke des Tupelkalküls

Ausdrücke im Tupelkalkül können unendliche Ergebnisse spezifizieren. So zum Beispiel folgender Ausdruck:

$$\{n \mid \neg(n \in \text{Professoren})\}$$

Um diesen unerwünschten Effekt entgegenzuwirken, wird eine Einschränkung bei der Formulierung von Anfragen im Tupelkalkül auf sogenannte *sichere* Anfragen vollzogen.

Ein Ausdruck des Tupelkalküls heisst sicher, wenn das Ergebnis des Ausdrucks eine Teilmenge der Domäne der Formel ist. Die Domäne einer Formel ist die Menge aller Werte, die als Konstante in der Formel vorkommen, und alle Werte (d.h. Attributwerte in Tupel) der Relationen, die in der Formel referenziert werden.

3.5.2 Der relationale Domänenkalkül

Im Unterschied zum Tupelkalkül werden Variablen im Domänenkalkül an Domänen, d.h. Wertemengen von Attributen, gebunden. Eine Anfrage hat folgende generische Struktur:

$$\{[v_1, v_2, \dots, v_n] \mid P(v_1, \dots, v_n)\}$$

Die v_i sind hierbei Variablen, die einen Attributwert Repräsentieren, und P ist ein Prädikat. Hierbei werden Variablen nun nicht mehr an Relationen gebunden, sondern eine Sequenz von Variablen wird an eine Relation gebunden. Zum Beispiel:

$$\{[m, n] \mid \exists s([m, n, s] \in \text{Studenten} \ \wedge s = 73245)\}$$

3.5.2.1 Sichere Ausdrücke des Domänenkalküls

Natürlich können Ausdrücke auch hier unendliche Ergebnisse produzieren, und so definiert man auch hier *sichere Ausdrücke*. Ein Ausdruck

$$\{[v_1, v_2, \dots, v_n] \mid P(v_1, \dots, v_n)\}$$

Ist sicher, falls folgende drei Bedingungen gelten:

1. Falls das Tupel $[c_1, \dots, c_n]$ mit Konstanten c_i im Ergebnis enthalten ist, so muss c_i in der Domäne von P liegen.

2. Für jede existenz-quantifizierte Teilformel $\exists x(P_1(x))$ muss gelten, dass P_1 nur für Elemente aus der Domäne von P_1 erfüllbar sein kann – oder eventuell für gar keine. Mit anderen Worten, wenn für eine Konstante c das Prädikat $P_1(c)$ erfüllt ist, so muss c in der Domäne von P_1 liegen.
3. Für jede universal-quantifizierte Teilformel $\forall x(P_1(x))$ muss gelten, dass sie dann und nur dann erfüllt ist, wenn $P_1(x)$ für alle Werte der Domäne von P_1 erfüllt ist. Mit anderen Worten, $P_1(d)$ muss für alle d, die *nicht* in der Domäne von P_1 enthalten sind, auf jeden Fall erfüllt sein.

3.6 Ausdruckskraft der Anfragesprachen

Codd hat die Ausdruckskraft von relationalen Anfragesprachen definiert. In seiner Terminologie heisst eine Anfragesprache *relational vollständig*, wenn sie mindestens so mächtig ist wie die relationale Algebra. Es gilt

- Die relationale Algebra,
- der relationale Tupelkalkül, eingeschränkt auf sichere Ausdrücke, und
- der relationale Domänenkalkül, eingeschränkt auf sichere Ausdrücke

besitzen dieselbe Ausdruckskraft.

4 SQL

Anfragesprachen in der Praxis, wie SQL, sind im Allgemeinen *deklarativ*. Die oft sehr komplexen, zur Festlegung der Auswertung nötigen Entscheidungen werden vom Anfrageoptimierer des Datenbanksystems übernommen. Dies hat den zusätzlichen Vorteil, dass die physische Datenunabhängigkeit grösstenteils gewährleistet werden kann.

Weiterhin realisieren Datenbanksysteme nicht Relation im mathematischen Sinne, sondern vielmehr Tabellen, die auch doppelte Eintrage enthalten können.

SQL besteht aus einer Familie von Standards

- Data defintion language (DDL) – Schemas definieren und ändern
- Data manipulation language (DML) – verändern der Daten
- Query language – Anfragen formulieren

4.1 Datentypen

Relationale Datenbanken stellen hauptsächlich drei Datentypen zur Verfügung: Zahlen, Zeichenketten und ein Typ für Datumsangaben.

- $\mathrm{char}(n)$ für feste Zeichenketten
- $\mathrm{varchar}(n)$ für Zeichenketten variabler Länge
- $\mathrm{integer}$ für ganze Zahlen
- $\mathrm{numeric}(p, s)$ für Zahlen mit p Stellen, davon sind s als Nachkommastellen reserviert
- blob oder raw für (sehr) grosse binäre Daten (*binary large object*)
- date für Datumsangaben

4.2 Schemadefinition (data definition language)

Die in SQL enthaltene *Data Definition Language (DDL)* erlaubt die Spezifizierung von Datenbankschemas. Eine neue Tabelle wird mit dem Befehl **create table** erstellt, zum Beispiel so:

```
create table Studenten
    ( StudNr   integer not null,
      Name     varchar(20) not null );
```

Tabellen können auch wieder gelöscht werden, nämlich mit **drop table** gefolgt vom Tabellennamen. Weiter lassen sich Schemata verändern:

```
alter table Studenten
    alter column Name varchar(30);
alter table Studenten
    add column Alter integer;
```

Ebenso können Spalten mit **drop column** gelöscht werden.

4.3 Datenmanipulation (data manipulation language)

Um Daten einzufügen, kann wie folgt vorgegangen werden:

```
insert into Studenten (Alter, Name)
    values (20, 'Peter Pan');
```

Weiter lassen sich natürlich auch Daten Löschen oder verändern:

```
delete Studenten where Alter < 13;
update Studenten set Semester = Semester + 1;
```

Änderungen jeglicher Art in SQL werden über eine sogenannte *Snapshot Semantic* ausgeführt. Dabei werden zuerst alle Tupel markiert, die vom Update betroffen sind, und erst in einem zweiten Schritt werden diese Änderungen schliesslich in die eigentliche Tabelle geschrieben.

4.4 SQL-Anfragen

Der grundsätzliche Aufbau einer Anfrage in SQL besteht aus drei Teilen. Zuerst wird die Menge der Attribute angegeben, welche man auswählen möchte, gefolgt von der Tabelle, aus welcher die Daten stammen sollen. Schlussendlich gibt es die Möglichkeit ein Auswahlprädikat anzugeben, dass von allen Zeilen des Resultats erfüllt werden muss. Ein Beispiel:

```
select Name, Alter
from Studenten
where Semester < 5;
```

Das Ergebnis kann auch sortiert werden, indem am Ende **order by** gefolgt vom Attribut, nach welchem sortiert werden soll angegeben wird. Mittels **desc** und **asc** kann angegeben, wie sortiert werden soll:

```
select Name, Alter
from Studenten
where Semester < 5
order by Alter desc, Name asc;
```

Aus verschiedenen Gründen wird in SQL *keine* Duplikatelimination vorgenommen. Wird dies dennoch gewünscht, kann **select** durch **select distinct** ersetzt werden:

```
select distinct Alter from Studenten;
```

Weiter besteht die Möglichkeit, Daten aus mehreren Tabellen gleichzeitig auszuwählen. Ebenso kann man Relationen eine *Tupelvariable* zuweisen, und Attribute qualifiziert ansprechen, um Mehrdeutigkeiten zu vermeiden:

```
select S.Name, B.Name
from Studenten S, Betreuer B
where S.Betreuer = B.Name;
```

4.4.1 Aggregatfunktionen und Gruppierung

Aggregatfunktionen führen Operationen auf Tupelmengen durch und komprimieren eine Menge von Werten zu einem einzelnen Wert. Zu ihnen gehören **avg**, **sum**, **max**, **min** und **count**, wobei die Bedeutung jeweils offensichtlich ist. Die Verwendung funktioniert wie folgt:

```
select avg(Alter)
from Studenten;
```

Besonders nützlich sind Aggregatfunktionen im Zusammenhang mit Gruppierung. Dabei werden alle Zeilen einer Tabelle zusammengefasst, die in einem oder mehreren Attribut übereinstimmen. Zusätzlich kann man diese Zeilen dann noch aggregieren, z.B.:

```
select avg(Alter), Semester
```

```
from Studenten
group by Semester
        having max(Alter) > 18;
```

Mit **having** kann eine zusätzliche Bedingung an die gebildeten Gruppen gestellt werden. Zu bemerken ist, dass **where** noch immer verwendet werden kann, da sich **where** nur auf die einzelnen Zeilen bezieht, nicht auf die gebildeten Gruppen.

4.4.2 Geschachtelte Anfragen

In SQL können Anfragen auf vielfältige Weise geschachtelt werden. Dabei werden Anfragen, die höchstens ein Tupel zurückliefern von denen unterschieden, die beliebig viele Tupel ergeben. Wenn eine Unteranfrage nur ein Tupel mit nur einem Attribut zurückliefert, so kann diese Unteranfrage dort eingesetzt werden, wo ein skalarer Wert gefordert wird.

4.4.3 Quantifizierte Anfragen

Der Existenzquantor wird in SQL durch **exists** realisiert. **exists** überprüft, ob die von einer Unteranfrage bestimmte Menge von Tupeln leer ist, und liefert entsprechen **true** oder **false**. Da kein Allquantor in SQL vorhanden ist, muss das Prädikat umgeformt werden, sodass eine äquivalente Variante mit Existenzquantor gefunden wird. Alternativ kann man das Problem auch über „zählen“ lösen, hierbei müssen aber weitere Annahmen getroffen werden können. Will man beispielsweise alle Studenten, die *alle* Vorlesungen besuchen, könnte man wie folgt vorgehen:

```
select StudNr
from VorlesungBesuchen
group by StudNr
        having count(*) = (select count(*) from Vorlesungen);
```

4.4.4 Nullwerte

In SQL existiert ein spezieller Wert mit dem Namen **null**, der in jedem Datentyp vorhanden ist. **null** steht dabei für „nicht vorhanden“ oder „nicht bekannt“. Es gibt einige Regeln, die bei diesem speziellen Wert zu beachten sind:

- In arithmetischen Ausdrücken werden Nullwerte propagiert, d.h. sobald ein Operant **null** ist, ist das Ergebnis ebenfalls **null**.
- SQL hat eine dreiwertige Logik mit **true**, **false** und **unknown**. Vergleiche liefern **unkonwn**, wenn mindestens einer der Operanden **null** ist.
- Logische Ausdrücke werden intuitiv behandelt, **unknown or false** bleibt **unknown**, aber **unknown and false** gibt **false**.
- In einer **where**-Bedingung werden nur Tupel weitergereicht, für welche die Bedingung **true** liefert.
- Bei einer Gruppierung wird **null** als eigener Wert behandelt.

Um auf **null** oder **unknow** zu testen, kann nicht = verwendet werden. Stattdessen muss **is null** bzw. **is unknown** benutzt werden.

4.4.5 Weitere Operationen

SQL unterstützt die folgenden *Mengenoperationen*: union, intersect und minus. Weiter gibt es „syntactic sugar“ um gewisse Prädikate schöner ausdrücken zu können:

```
select *
from Studenten
where Semester between 1 and 4;

select *
from Studenten
where Semester in (2,3,6);
```

Für Vergleiche von Zeichenketten kann **like** verwendet werden, wobei unbekannte Teile mit den Platzhalten % und _ ersetzt werden können. % steht für belieb viele Zeichen, _ für genau eines.

Weiter können mit **case** Attributwerte „dekodiert“ werden:

```
select Name, ( case   when Alter < 19 then 'jung'
                      when Alter < 25 then 'normal'
                      else 'alt' end )
from Studenten;
```

Hierbei wird jeweils höchstens eine Klausel ausgeführt, wobei von oben her begonnen wird.

4.4.6 Joins in SQL-92

In SQL-92 wurde die Möglichkeit zur direkten Angabe eines Join-Operators geschaffen, sodass im from Teil nun folgende Schlüsselwörter verwendet werden können:

- **cross join**: Kreuzprodukt
- **natural join**: natürlicher Join
- **join** oder **inner join**: Theta-Join
- **left**, **right** oder **full outer join**: äussere Joins

So entsteht zum Beispiel folgende Anfrage:

```
select *
from Studenten S
        left outer join VorlesungBesuchen V on S.StudNr = V.StudNr;
```

4.5 Sichten

Sichten können aus verschiedenen Gründen eingesetzt werden:

- Logische Datenunabhängigkeit
- Um Daten zu schützen, indem gewissen Usern nur einen Teil der Daten zur Verfügung gestellt werden
- Um Anfragen zu vereinfachen
- Um Generalisierung zu implementieren (Inklusion und Vererbung)

Will man Generalisierung mithilfe von Sichten implementieren, so gibt es verschiedene Ansätze. Entweder wird der Obertyp als Sicht implementiert, und die Untertypen sind direkt als Tabellen verfügbar,

oder umgekehrt. Welche Methode besser ist, hängt stark davon ab, was für Anfragen später gemacht werden.

4.5.1 Update-fähige Sichten

Sichten sind im Allgemeinen nicht update-fähig. Von den theoretisch änderbaren Sichten ist jedoch in SQL nur eine Teilmenge wirklich änderbar, nämlich genau dann, wenn folgende Bedingungen erfüllt sind:

- Die Sicht enthält weder Aggregatfunktionen, noch Anweisungen wie **distinct**, **group by** oder **having**.
- Die **select**-Liste enthält nur eindeutige Spaltennamen, und ein Schlüssel der Basisrelation ist enthalten in dieser Liste.
- Die Sicht verwendet nur gerade eine Tabelle (also Basisrelation oder Sicht), die ebenfalls veränderbar sein muss.

5 Datenintegrität

Ein DBMS ist nicht nur für die Speicherung von Daten zuständig, sondern stellt auch deren Konsistenz sicher. Es wird zwischen *statischen* und *dynamischen Integritätsbedingungen* unterschieden. Eine statische Bedingung muss von jedem Zustand der Datenbank erfüllt werden, etwa dürfen Personen kein negatives Alter haben. Dynamische Bedingungen hingegen werden an Zustandsänderungen gestellt, etwa dass das Alter eines Studenten nicht kleiner werden darf.

Bisher wurden bereits einige implizite Anforderungen an die Datenintegrität gestellt:

- Durch das Bestimmen von Schlüsseln wurde festgelegt, dass eine zwei Tupel mit demselben Schlüssel vorkommen dürfen.
- Die Kardinalitäten von Relation geben Integritätsbedingungen vor.
- Jedes Attribut hat einen Wertebereich.

5.1 Referentielle Integrität

Verwendet man den Schlüssel einer Relation als Attribut einer anderen Relation, so spricht man von einem *Fremdschlüssel* (*foreign key*). Seien R und S zwei Relationen und sei κ Primärschlüssel von R. Dann ist $\alpha \subset \mathcal{S}$ ein Fremdschlüssel, wenn für alle Tupel $s \in S$ gilt:

1. $s.\alpha$ enthält entweder nur Nullwerte, oder nur Werte ungleich **null**.
2. Enthält $s.\alpha$ keine Nullwerte, so existiert ein Tupel $r \in R$ mit $s.\alpha = r.\kappa$.

Die Erfüllung dieser Eigenschaft wird *referentielle Integrität* genannt.

5.1.1 Referenzielle Integrität in SQL

Zur Einhaltung der referentiellen Integrität gibt es für jeden der drei Schlüsselbegriffe eine Beschreibungsmöglichkeit.

- Ein Schlüssel(-kandidat) wird durch die Angabe von **unique** gekennzeichnet.
- Der Primärschlüssel wird mit **primary key** markiert und wird automatisch als **not null** spezifiziert.
- Ein Fremdschlüssel heisst **foreign key**. Fremdschlüssel können auch **null** sein, falls nicht explizit **not null** angegeben wurde. Ein **unique foreign key** modelliert eine 1:1 Beziehung.

Zusätzlich kann noch das Verhalten bei Änderungen an Verweisen oder referenzierten Daten festgelegt werden, wobei es drei Möglichkeiten gibt:

1. Default: Änderungen werden zurückgewiesen und geben einen Fehler.
2. **cascade**: Die Änderung wird weitergereicht/propagiert.
3. **set null**: Die Referenz wird auf null gesetzt.

Insgesamt könnte eine Umsetzung von diesen Bedingungen folgendermassen aussehen:

```
create table example (
id integer  primary key,
name varchar(20),
other_id integer,
..
```

```
constraint other_fk foreign key(other_id)
        references other_table(id) on delete cascade,
... );
```

Falls nur ein Attribut als Fremdschlüssel verwendet wird, so kann statt **foreign key** direkt **references** verwendet werden.

```
create table example (
id integer  primary key,
name varchar(20),
other_id integer references other_table(id) on delete cascade);
```

5.2 Statische Integritätsbedingungen

Statische Integritätsbedingungen werden in SQL durch **check**-Anweisungen gefolgt von einer Bedingung implementiert. Dabei werden Änderungsoperationen an einer Tabelle zurückgewiesen, wenn die Bedingung zu **false** auswertet. Typisch sind Bereichseinschränkungen und die Realisierung von Aufzählungstypen. Beispiele:

… **check** Alter **between** 0 **and** 100 …

… **check** Geschlecht **in** (‚m‘, ‚f‘) …

Es sind allerdings auch kompliziertere Bedingungen möglich, mit Unterabfragen, etc.

5.3 Trigger

Trigger sind der allgemeinste Konsistenzsicherungsmechanismus, wurden jedoch erst in SQL-1999 standardisiert.

5.4 Wo werden Integritätsbedingungen überprüft?

Integritätsbedingungen können entweder in der Datenbank oder aber innerhalb der Applikation selbst durchgeführt werden. Es gibt verschiedene Argumente, die Bedingungen auf Datenbankebene zu implementieren:

- Integritätsbedingungen beschreiben und dokumentieren das Datenbankschema.
- Die Datenbank ist ein zentraler Ort; die Bedingungen müssen nur ein Mal implementiert werden, egal wie viele Anwendungen die Datenbanken verwenden.
- Nützlich für Datenbankoptimierungen.

Allerdings gibt es auch (mindestens) ein sehr wichtiges Argument für Bedingungen in der Anwendung, nämlich um aussagekräftige Fehlermeldungen zu generieren. Daher ist es wichtig, beides zu machen.

6 Relationale Entwurfstheorie

6.1 Funktionale Abhängigkeiten

Eine *funktionale Abhängigkeit* (*functional dependency*) stellt eine Bedingung an die möglichen gültigen Ausprägungen des Datenbankschemas dar. Eine funktionale Abhängigkeit, oft abgekürzt als FD, wird wie folgt dargestellt:

$$\alpha \rightarrow \beta$$

Hierbei repräsentieren α und β jeweils Mengen von Attributen. Für ein Schema, für welches diese FD gelten soll, sind nur solche Ausprägungen zulässig, für die folgendes gilt: Für alle Paare von Tupeln $r, t \in R$ mit $r.\alpha = t.\alpha$ muss auch gelten $r.\beta = t.\beta$. Man sagt, dass die α-Werte die β-Werte funktional (d.h. eindeutig) bestimmen.

Alternativ kann man diese Bedingung auch folgendermassen formulieren: Die FD $\alpha \rightarrow \beta$ ist in R erfüllt, wenn für jeden möglichen Wert c von α gilt, dass

$$\Pi_{\beta}(\sigma_{\alpha=c}(R))$$

höchstens ein Element enthält.

6.2 Schlüssel

In einer Relation ist $\alpha \subseteq \mathcal{R}$ ein *Superschlüssel*, falls gilt

$$\alpha \rightarrow \mathcal{R}$$

Das heisst α bestimmt alle anderen Attributwerte innerhalb der Relation. Trivialerweise ist die Menge aller Attribute stets ein Superschlüssel.

β ist *voll funktional abhängig* von α, in Zeichen $\alpha \dot{\rightarrow} \beta$, falls die folgenden beiden Kriterien gelten:

- $\alpha \rightarrow \beta$, d.h. β ist funktional abhängig von α
- α kann nicht mehr „verkleinert“ werden, d.h.

  $$\forall A \in \alpha: \alpha - \{A\} \nrightarrow \beta$$

  Es kann also kein Attribut mehr von α entfernt werden, ohne die FD zu „zerstören“.

Falls $\alpha \dot{\rightarrow} \mathcal{R}$ gilt, so bezeichnet man α als *Kandidatenschlüssel*. Im Allgemeinen wird einer der Kandidatenschlüssel als *Primärschlüssel* ausgewählt, wobei dieser dann als Fremdschlüssel verwendet wird.

6.3 Bestimmung funktionaler Abhängigkeiten

Welche funktionalen Abhängigkeiten für die modellierten Relationen gelten sollen, muss der Datenbankentwerfen entscheiden. Im Allgemeinen sind wir bei einer gegebenen Menge F von FDs daran interessiert, die Menge F^+ aller daraus herleitbaren funktionalen Abhängigkeiten zu bestimmen. Diese Menge F^+ bezeichnet man als die *Hülle* (*closure*) der Menge F.

Für die Herleitung der Hülle reichen die folgenden *Armstrong-Axiome* als Inferenzregeln aus:

- *Reflexivität*: Falls β eine Teilmenge von α ist, dann gilt immer $\alpha \rightarrow \beta$. Insbesondere gilt $\alpha \rightarrow \alpha$.
- *Verstärkung*: Falls $\alpha \rightarrow \beta$ gilt, dann gilt auch $\alpha\gamma \rightarrow \beta\gamma$.
- *Transitivität*: Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, dann gilt auch $\alpha \rightarrow \gamma$.

Auch wenn diese Axiome korrekt und vollständig sind, ist es oftmals nützlich, die folgenden, ebenfalls korrekten Regeln verwenden zu können:

- *Vereinigungsregel*: Wenn $\alpha \to \beta$ und $\alpha \to \gamma$ gelten, dann gilt auch $\alpha \to \beta\gamma$.
- *Dekompositionsregel*: Wenn $\alpha \to \beta\gamma$ gilt, dann gelten auch $\alpha \to \beta$ und $\alpha \to \gamma$.
- *Pseudotransitivitätsregel*: Wenn $\alpha \to \beta$ und $\gamma\beta \to \delta$ gelten, dann auch $\alpha\gamma \to \delta$.

Oftmals ist man nicht an der gesamten Hülle einer Menge von FDs interessiert, sondern nur an der Menge von Attributen α^+, die von α gemäss der Menge F von FDs funktional bestimmt werden kann. Dies kann folgendermassen hergeleitet werden:

Eingabe: eine Menge F von FDs und eine Menge von Attributen α
Ausgabe: die vollständige Menge von Attributen α^+, für die gilt $\alpha \to \alpha^+$
AttrHülle(F,α)
    Erg := α;
    while (Änderung an Erg) **do**
        foreach FD $\beta \to \gamma$ in F **do**
            if $\beta \subseteq$ Erg **then** Erg := Erg $\cup\ \gamma$;
    Ausgabe α^+ = Erg

Mithilfe dieses Algorithmus kann natürlich auch einfach bestimmt werden, ob eine Menge von Attributen ein Superschlüssel einer Relation darstellt.

6.3.1 Kanonische Überdeckung

Im Allgemeinen gibt es viele unterschiedliche äquivalente Mengen von funktionalen Abhängigkeiten. Zwei Mengen F und G von funktionalen Abhängigkeiten heissen genau dann *äquivalent*, $F \equiv G$, wenn ihre Hüllen gleich sind, d.h. $F^+ = G^+$.

Da die Hülle in der Regel sehr gross und unübersichtlich ist, ist man an einer kleinstmöglichen noch äquivalenten Menge von FDs interessiert. Zu einer gegebenen Menge F von FDs nennt man F_c eine *kanonische Überdeckung*, falls folgende drei Eigenschaften erfüllt sind:

- $F_c \equiv F$, d.h. $F_c^+ = F^+$
- In F_c exisiteren keine FDs $\alpha \to \beta$, bei denen α oder β überflüssige Attribute enthalten. Das heisst, es muss gelten:
    - $\forall A \in \alpha: \left(F_c - (\alpha \to \beta) \cup \left((\alpha - A) \to \beta\right)\right) \not\equiv F_c$
    - $\forall B \in \beta: \left(F_c - (\alpha \to \beta) \cup \left(\alpha \to (\beta - B)\right)\right) \not\equiv F_c$
- Jede linke Seite einer funktionalen Abhängigkeit in F_c ist einzigartig. Dies kann durch sukzessive Anwendung der Vereinigungsregel erreicht werden.

Die kanonische Überdeckung lässt sich wie folgt berechnen

1. Führe für jede FD $\alpha \to \beta \in F$ die Linksreduktion durch, also:
    a. Überprüfe für alle $A \in \alpha$, ob A überflüssig ist, also ob
    $$\beta \subseteq \text{AttrHülle}(F, \alpha - A)$$
    gilt. Falls dies der Fall ist, ersetze $\alpha \to \beta$ durch $(\alpha - A) \to \beta$.
2. Führe für jede (verbleibende) FD $\alpha \to \beta$ eine Rechtsreduktion durch, also:

a. Überprüfe für alle $B \in \beta$, ob B überflüssig ist, also ob

$$B \in \text{AttrHülle}(F - (\alpha \to \beta) \cup (\alpha \to (\beta - B), \alpha)$$

gilt. Falls dies der Fall ist, ersetze $\alpha \to \beta$ durch $\alpha \to (\beta - B)$.

3. Entferne alle FDs der Form $\alpha \to \emptyset$, die im 2. Schritt möglicherweise entstanden sind.
4. Fasse mittels der Vereinigungsregel FDs mit derselben linken Seite zusammen.

6.4 Dekomposition von Relationen

Werden verschiedene Sachverhalte in einer Relation vermischt, so kommt es zu gewissen Anomalitäten. Solche Relationen werden durch eine sogenannte *Normalisierung*, in mehrere Relationschemata aufgespalten. Dabei gibt es zwei sehr grundlegende Korrektheitskriterien für eine solche Zerlegung:

- *Verlustlosigkeit*: Die in der ursprünglichen Relationsausprägung R mit Schema $\mathcal{R}$ enthaltenen Informationen müssen aus den Ausprägungen $R_1, \ldots, R_n$ der neuen Relationsschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ rekonstruierbar sein.
- *Abhängigkeitserhaltung*: Die für $\text{sch}(\mathcal{R})$ geltenden funktionalen Abhängigkeiten müssen auf die Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ übertragbar sein.

Der Einfachheit halber beschränken wir uns bei der Diskussion dieser Kriterien auf die Dekomposition in zwei Relationen.

6.4.1 Verlustlosigkeit

Eine Zerlegung ist gültig, wenn alle Attribute aus R erhalten bleiben, d.h. es muss gelten:

$$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$$

Für eine Ausprägung R gilt nun offensichtlich:

$$R_1 = \Pi_{\mathcal{R}_1}(R)$$

$$R_2 = \Pi_{\mathcal{R}_2}(R)$$

Die Zerlegung heisst nun verlustlos, falls für jede mögliche (gültige) Ausprägung R gilt:

$$R = R_1 \bowtie \mathrm{R}_2$$

6.4.2 Kriterien für die Verlustlosigkeit einer Zerlegung

Eine Zerlegung von $\mathcal{R}$ ist verlustlos, wenn mindestens eine der folgenden funktionalen Abhängigkeiten herleitbar ist:

- $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_1 \in F^+_{\text{sch}(R)}$
- $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_2 \in F^+_{\text{sch}(R)}$

Dies ist eine hinreichende, aber keine notwendige Bedingung.

6.4.3 Abhängigkeitsbewahrung

Funktionale Abhängigkeiten stellen Konsistenzbedingungen dar, die von jeder Ausprägung von erfüllt sein müssen. Zerlegt man nun eine Relation, will man verhindern, dass zum überprüfen der Bedingungen die ursprüngliche Relation durch einen teuren Join wiederhergestellt werden muss. Deshalb wird durch die *Abhängigkeitsbewahrung* gefordert, dass alle Abhängigkeiten lokal auf den $\mathcal{R}_i$ überprüft werden können. Dazu bestimmt man für jedes $\mathcal{R}_i$ die Einschränkung F_{R_i} der Abhängigkeiten aus F_R^+, d.h. F_{R_i}

enthält die Abhängigkeiten aus der Hülle, deren Attribute alle in $\mathcal{R}_i$ enthalten sind. Bei der Abhängigkeitsbewahrung wird dann folgendes gefordert:

$$F_R \equiv \left(F_{R_1} \cup \cdots \cup F_{R_n}\right) \quad \text{beziehungsweise dazu äquivalent} \quad F_R^+ = \left(F_{R_1} \cup \cdots \cup F_{R_n}\right)^+$$

Entsprechend dieser Bedingung nennt man eine abhängigkeitsbewahrende Zerlegung oft auch eine *hüllentreue* Dekomposition.

6.5 Erste Normalform

Die erste Normalform ist bei der von uns benutzten Definition der relationalen Modells automatisch eingehalten. Die erste Normalform verlangt, dass alle Attribute atomare Wertebereiche (Domänen) haben. Demnach wären zusammengesetzte mengenwertige oder gar relationenwertige Attributdomänen nicht zulässig.

6.6 Zweite Normalform

Intuitiv verletzt ein Relationschema die zweite Normalform (2NF), wenn in der Relation Informationen über mehr als ein einziges Konzept modelliert werden. Demnach soll jedes Nichtschlüsselattribut der Relation einen Fakt zu dem dieses Konzept identifizierenden Schlüssel ausdrücken.

Formal ausgedrückt: Eine Relation mit Schema $\mathcal{R}$ mit zugehörigen FDs F ist in zweiter Normalform, falls jedes Nichtschlüssel-Attribut $A \in \mathcal{R}$ voll funktional abhängig ist von jedem Kandidatenschlüssel der Relation.

Seien also $\kappa_1, \ldots, \kappa_i$ die Kandidatenschlüssel von $\mathcal{R}$, einschliesslich des gewählten Primärschlüssels. Sei $A \in \mathcal{R} - (\kappa_1 \cup \cdots \cup \kappa_i)$. Ein solches Attribut A wird auch als *nicht-prim* bezeichnet, da es im Gegensatz zu den Schlüsselattributen nicht Teil eines Kandidatenschlüssels ist. Schlüsselattribute werden demzufolge als *prim* bezeichnet. Nun muss für alle κ_j gelten:

$$\kappa_j \dot{\rightarrow} A \in F^+$$

Das heisst, es muss die FD $\kappa_j \rightarrow A$ gelten, und diese FD ist linksreduziert.

6.7 Dritte Normalform

Die dritte Normalform wird intuitiv verletzt, wenn ein Nichtschlüsselattribut einen Fakt einer Attributmenge darstellt, die keinen Schlüssel bildet. Eine Verletzung könnte also dazu führen, dass derselbe Fakt mehrfach abgespeichert wird.

Ein Relationschema $\mathcal{R}$ ist in *dritter Normalform*, wenn für jede geltende funktionale Abhängigkeit der Form $\alpha \rightarrow B$ mit $\alpha \subseteq \mathcal{R}$ und $B \in \mathcal{R}$ *mindestens eine* von drei Bedingungen gilt:

- $B \in \alpha$, d.h. die FD ist trivial.
- Das Attribut B ist in einem Kandidatenschlüssel enthalten, also ist B prim.
- α ist Superschlüssel der Relation.

6.7.1 Synthesealgorithmus

Der folgende *Synthesealgorithmus* gibt eine Zerlegung eines Relationenschemas mit gegebenen funktionalen Abhängigkeiten an, sodass alle der folgenden drei Bedingungen erfüllt sind:

- Es ist eine verlustlose Zerlegung

- Die Zerlegung ist abhängigkeitsbewahrend
- Alle entstehenden Relationen sind in dritter Normalform

Der Algorithmus arbeitet wie folgt:

1. Bestimme die kanonische Überdeckung F_c zu F, also
    a. Linksreduktion der FDs
    b. Rechtsreduktion der FDs
    c. Entfernung von FDs der Form $\alpha \rightarrow \emptyset$
    d. Zusammenfassung von FDs mit gleichen linken Seiten
2. Für jede funktionale Abhängigkeit $\alpha \rightarrow \beta \in F_c$:
    a. Kreiere ein Relationenschema $\mathcal{R}_\alpha = \alpha \cup \beta$.
    b. Ordne $\mathcal{R}_\alpha$ die FDs $F_\alpha = \{\alpha' \rightarrow \beta' \in F_c | \alpha' \cup \beta' \subseteq \mathcal{R}_\alpha\}$ zu
3. Falls eines der im zweiten Schritt erzeugten Schemata $\mathcal{R}_\alpha$ einen Kandidatenschlüssel enthält, sind wir fertig; sonst wähle einen Kandidatenschlüssel $\kappa \in \mathcal{R}$ aus und definiere folgendes zusätzliches Schema:
    a. $\mathcal{R}_\kappa = \kappa$ mit $F_\kappa = \emptyset$
4. Eleminiere diejenigen Schemata $\mathcal{R}_\alpha$, die in einem anderen Relationenschema $\mathcal{R}_{\alpha'}$ enthalten sind, d.h.

$$\mathcal{R}_\alpha \subseteq \mathcal{R}_{\alpha'}$$

6.8 Boyce-Codd Normalform

Die Boyce-Codd Normalform (BCNF) stellt nochmals eine Verschärfung dar. Das Ziel der BCNF besteht darin, dass Informationseinheiten (Fakten) nicht mehrmals, sondern nur genau einmal gespeichert werden. Ein Relationschema ist in BCNF, falls für jede funktionale Abhängigkeit $\alpha \rightarrow \beta$ mindestens eine der folgenden zwei Bedingungen gilt:

- $\beta \subseteq \alpha$, d.h. die Abhängigkeit ist trivial.
- α ist ein Superschlüssel der Relation.

6.8.1 Dekompositionsalgorithmus

Man kann grundsätzlich jedes Relationenschema mit zugeordneten FDs in n Relationen zerlegen, sodass gilt:

- Die Zerlegung ist verlustlos.
- Die entstehenden Relationen sind in BCNF.

Leider kann man nicht immer eine BCNF-Zerlegung finden, die auch abhängigkeitsbewahrend ist. Diese Fälle sind in der Praxis jedoch selten.

Die Zerlegung eines Schemas $\mathcal{R}$ in BCNF-Teilrelationen wird nach dem *Dekompositionsalgorithmus* durchgeführt, der die Menge $Z = \{\mathcal{R}_1, \dots, \mathcal{R}_n\}$ von Zerlegungen sukzessive generiert:

1. Starte mit $Z = \{\mathcal{R}\}$
2. Solange es noch ein Relationschema $\mathcal{R}_i \in Z$ gibt, das nicht in BCNF ist, mache folgendes:
    a. Finde eine für $\mathcal{R}_i$ geltende nicht-triviale FD ($\alpha \rightarrow \beta$) mit
        - $\alpha \cap \beta = \emptyset$

- $\alpha \nrightarrow \mathcal{R}_i$

  Man sollte die funktionale Abhängigkeit so wählen, dass β alle von α funktional abhängigen Attribute $B \in (R_i - \alpha)$ enthält, damit der Algorithmus möglichst schnell terminiert.

b. Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i_1} \coloneqq \alpha \cup \beta$ und $\mathcal{R}_{i_2} \coloneqq \mathcal{R}_i - \beta$

c. Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i_1}$ und $\mathcal{R}_{i_2}$ ein.

6.9 Mehrwertige Abhängigkeiten

Mehrwertige Abhängigkeiten (*multivalued dependencies*, abgekürzt MVD) sind eine Verallgemeinerung der funktionalen Abhängigkeiten, d.h. jede FD ist auch eine MVD, aber nicht umgekehrt.

Seien α und β disjunkte Teilmengen von $\mathcal{R}$ und $\gamma = \mathcal{R} - (\alpha \cup \beta)$. Dann ist β mehrwertig abhängig von α, in Zeichen $\alpha \twoheadrightarrow \beta$, wenn in jeder gültigen Ausprägung von $\mathcal{R}$ gilt: Für jedes Paar von Tupeln t_1 und t_2 mit $t_1.\alpha = t_2.\alpha$ existieren zwei weitere Tupel t_3 und t_4 mit folgenden Eigenschaften:

$$t_1.\alpha = t_2.\alpha = t_3.\alpha = t_4.\alpha$$
$$t_3.\beta = t_1.\beta$$
$$t_3.\gamma = t_2.\gamma$$
$$t_4.\beta = t_2.\beta$$
$$t_4.\gamma = t_1.\gamma$$

Mit anderen Worten: Bei zwei Tupeln mit gleichem α-Wert kann man die β-Werte vertauschen, und die resultierenden Tupel müssen auch in der Relation sein. Aus diesem Grund nennt man mehrwertige Abhängigkeiten auch *tupel-generierende* Abhängigkeiten, da eine Relationsausprägung bei Verletzung einer MVD durch das Einfügen zusätzlicher Tupel in einen gültigen Zustand überführt werden kann.

Ein Relationenschema $\mathcal{R}$ mit einer Menge D von zugeordneten funktioanlen und mehrwertigen Abhängigkeiten kann genau dann verlustlos in die beiden Schemata $\mathcal{R}_1$ und $\mathcal{R}_2$ zerlegt werden, wenn gilt:

- $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$
- Mindestens eine der folgenden MVDs gilt:
  - $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$
  - $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$

Es gelten folgende Inferenzregeln zum Bestimmen der Hülle D^+:

- *Reflexivität*: Falls β eine Teilmenge von α ist, dann gilt immer $\alpha \rightarrow \beta$. Insbesondere gilt $\alpha \rightarrow \alpha$.
- *Verstärkung*: Falls $\alpha \rightarrow \beta$ gilt, dann gilt auch $\alpha\gamma \rightarrow \beta\gamma$.
- *Transitivität*: Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, dann gilt auch $\alpha \rightarrow \gamma$.
- *Komplement:* Sei $\alpha \twoheadrightarrow \beta$. Dann gilt $\alpha \twoheadrightarrow \mathcal{R} - \beta - \alpha$.
- *Mehrwertige Verstärkung*: Sei $\alpha \twoheadrightarrow \beta$ und $\delta \subseteq \gamma$. Dann gilt $\gamma\alpha \twoheadrightarrow \delta\beta$.
- *Mehrwertige Transitivität:* Sei $\alpha \twoheadrightarrow \beta$ und $\beta \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \gamma - \beta$
- *Verallgemeinerung*: Sei $\alpha \rightarrow \beta$, dann gilt auch $\alpha \twoheadrightarrow \beta$.
- *Koaleszenz*: Sei $\alpha \twoheadrightarrow \beta$ und $\gamma \subseteq \beta$. Existiert ein $\delta \subseteq \mathcal{R}$, so dass $\delta \cap \beta = \emptyset$ und $\delta \rightarrow \gamma$, so gilt $\alpha \rightarrow \gamma$.
- *Mehrwertige Vereinigung*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \gamma\beta$.

- *Schnittmenge*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt auch $\alpha \twoheadrightarrow \beta \cap \gamma$.
- *Differenz*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt auch $\alpha \twoheadrightarrow \beta - \gamma$ und $\alpha \twoheadrightarrow \gamma - \beta$

6.10 Vierte Normalform

Die vierte Normalform (4NF) ist eine Verschärfung der Boyce-Codd Normalform, und somit auch der ersten, zweiten und dritten Normalform. Hierbei wird die durch mehrwertige Abhängigkeiten verusachte Redundanz ausgeschlossen.

Eine Relation $\mathcal{R}$ mit einer zugeordneten Menge D an funktionalen und mehrwertigen Abhängigkeiten ist in *vierter Normalform* 4NF, wenn für jede MVD $\alpha \twoheadrightarrow \beta \in D^+$ eine der folgenden Bedinungen gilt:

1. Die MVD ist trivial.
2. α ist ein Superschlüssel von $\mathcal{R}$.

Eine MVD heisst dabei trivial, wenn eine der folgenden Aussagen gilt:

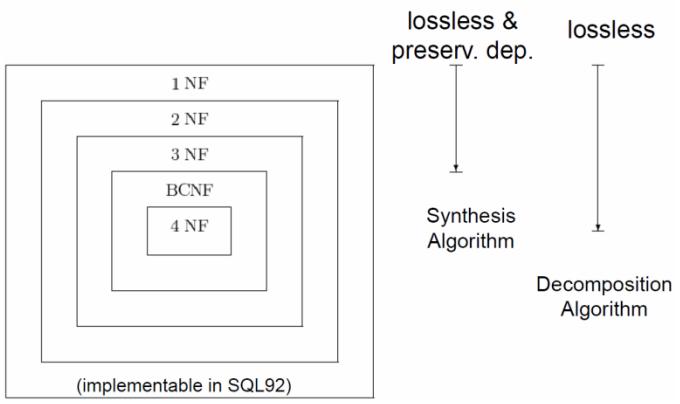
1. $\beta \subseteq \alpha$ oder
2. $\beta = \mathcal{R} - \alpha$

6.10.1 Dekompositionsalgorithmus

Auch für die vierte Normalform gibt es ein Dekompositionsalgorithmus:

1. Starte mit $Z = \{\mathcal{R}\}$
2. Solange es noch ein Relationschema $\mathcal{R}_i \in Z$ gibt, das nicht in 4NF ist, mache folgendes:
    a. Finde eine für $\mathcal{R}_i$ geltende nicht-triviale MVD ($\alpha \twoheadrightarrow \beta$) mit
        - $\alpha \cap \beta = \emptyset$
        - $\alpha \nrightarrow \mathcal{R}_i$
    b. Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i_1} \coloneqq \alpha \cup \beta$ und $\mathcal{R}_{i_2} \coloneqq \mathcal{R}_i - \beta$
    c. Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i_1}$ und $\mathcal{R}_{i_2}$ ein.

6.11 Zusammenfassung



7 Transaktionsverwaltung

Unter einer Transaktion versteht man die „Bündelung“ mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen als *Einheit* fehlerfrei ausgeführt werden sollen.

Es gibt zwei grundlegende Anforderungen:

- Recovery, d.h. die Behebung von eingetretenen, oft unvermeidbaren Fehlersituationen.
- Synchronisation von mehreren gleichzeitig auf der Datenbank ablaufenden Transaktionen.

7.1 Operationen auf Transaktions-Ebene

Eine Transaktion besteht aus einer Folge von elementaren Operationen (lesen, verändern, einfügen, löschen), wobei die Datenbasis von einem konsistenten Zustand in einen anderen – nicht notwendigerweise unterschiedlichen – konsistenten Zustand überführt wird. Aus der Sicht des Datenbanksystems handelt es sich hierbei um die Operationen **read** und **write**. Für die Steuerung der Transaktionsverarbeitung sind zusätzlich noch Operationen auf der Transaktionsebene notwendig:

- **begin of transaction** (**BOT**): Mit diesem Befehl wird der Beginn einer eine Transaktion darstellenden Befehlsfolge gekennzeichnet.
- **commit**: Hierdurch wird die Beendigung der Tranaktion eingeleitet. Alle Änderungen der Datenbasis werden durch diesen Befehl *festgeschrieben*, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort**: Dieser Befehl führt zu einem Selbstabbruch der Transaktion. Das Datenbanksystem muss sicherstellen, dass die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.

Es gibt drei Möglichkeiten, wie eine Transaktion beendet werden kann. Zum einen durch die beiden bereits erwähnten Befehle **commit** und **abort**, oder durch einen Fehler (z.B. Hardware- oder Software-Fehler, Stromausfall, Deadlock). Dabei wird nur nach einen **commit** die Datenbasis wirklich verändert, in den beiden anderen Fällen wird der ursprüngliche Zustand wiederhergestellt. Dies wird auch *rollback* genannt.

Neben den drei erwähnten Befehlen gibt es auch noch zwei weitere Befehle eines Transaktionsverwaltungssystemes:

- **define savepoint**: Hierdurch wird ein Sicherungspunkt definiert, auf den sich die (noch aktive) Transaktion zurücksetzen lässt. Das DBMS muss sich dazu alle bis zu diesem Zeitpunkt ausgeführten Änderungen an der Datenbasis „merken“. Diese Änderungen dürfen aber noch nicht in der Datenbasis festgeschrieben werden, da die Transaktion durch ein **abort** noch immer gänzlich aufgegeben werden kann.
- **backup transaction**: Dieser Befehl dient dazu, die noch aktive Transaktion auf den zuletzt angelegten Sicherungspunkt zurückzusetzen. Je nach System können auch Sicherungspunkte, die weiter in der Vergangenheit liegen, angesprochen werden.

7.2 Eigenschaften einer Transaktion

Die Eigenschaften des Transaktionskonzepts werden oft unter der Abkürzung ACID zusammengefasst. Das sogenannte *ACID-Paradigma* steht dabei für vier Eigenschaften:

- **Atomicity.** Diese Eigenschaft verlangt, dass eine Transaktion als kleinste, nicht weiter zerlegbare Einheit behandelt wird. Das heisst, entweder werden alle Änderungen der Transaktion in der Datenbasis festgeschrieben, oder gar keine.
- **Consistency.** Eine Transaktion hinterlässt nach Beendigung einen konsistenten Datenbasiszustand. Andernfalls wird sie komplett zurückgesetzt. Zwischenzustände, die während der TA-Bearbeitung entstehen, dürfen inkonsistent sein, aber der resultierende Endzustand muss die im Schema definierten Konsistenzbedingungen (z.B. referentielle Integrität) erfüllen.
- **Isolation.** Diese Eigenschaft verlangt, dass parallel ausgeführte Transaktionen sich nicht gegenseitig beeinflussen. Jede Transaktion muss – logisch gesehen – so ausgeführt werden, als wäre sie die einzige Transaktion, die während ihrer gesamten Ausführungszeit auf dem Datenbanksystem aktiv ist. Alle anderen Transaktionen bzw. deren Effekte dürfen also nicht sichtbar sein.
- **Durability (Dauerhaftigkeit).** Die Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten. Die Transaktionsverwaltung muss sicherstellen, dass dies auch nach einem Systemfehler (Hardware oder Systemsoftware) gewährleistet ist. Die einzige Möglichkeit, die Wirkung einer erfolgreich abgeschlossenen Transaktion ganz oder teilweise aufzuheben, besteht darin, eine andere sogenannte kompensierende Transaktion auszuführen.

7.3 Transaktionsverwaltung in SQL

In SQL-92 gibt es keinen **begin of transaction**-Befehl; die Ausführung der ersten Anweisung beginnt automatisch eine Transaktion. Abgeschlossen wird selbige dann durch einen der folgenden Befehle:

- **commit work**: Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzungen aufgedeckt werden – festgeschrieben. Das Schlüsselwort **work** ist dabei optional.
- **rollback work**: Alle Änderungen werden zurückgesetzt, wobei dieser Befehl immer erfolgreich vom DBMS ausgeführt können werden muss.

8 Mehrbenutzersynchronisation

8.1 Fehler bei unkontrolliertem Betrieb

8.1.1 Verlorengegangene Änderungen (lost updates)

Dieses Problem tritt auf, wenn zwei Transaktionen unkontrolliert gleichzeitig stattfinden. Ein Beispiel:

T_1	T_2
$\text{read}(A, a_1)$	
$a_1 := a_1 - 300$	
	$\text{read}(A, a_2)$
	$a_2 := a_2 \cdot 1.03$
	$\text{write}(A, a_2)$
$\text{write}(A, a_1)$	

In diesem Beispiel ist nach Beendigung von Transaktion T_1 keine Auswirkung mehr von T_2 mehr sichtbar, die Veränderung ging vollständig verloren, da sie von T_1 überschrieben wurde.

8.1.2 Phantomproblem

Man spricht vom *Phantomproblem*, wenn das Ausführen derselben Anfrage innerhalb einer Transaktion verschiedene Resultate erzeugt werden. Dies ist der Fall, wenn eine andere Transaktion ein Element einfügt, verändert oder entfernt.

8.2 Serialisierbarkeit

Offensichtlich will man Transaktionen aus Effizienzgründen nicht vollständig seriell, also hintereinander, ausführen. Allerdings wären dann die beschriebenen Probleme nicht vorhanden, da sich Transaktionen nicht beeinflussen können.

Intuitiv entspricht die *serialisierbare Ausführung* einer Menge von Transaktionen einer kontrollierten, nebenläufigen, verzahnten Ausführung, wobei die Kontrollkomponenten dafür sorgt, dass die (beobachtbare) Wirkung der nebenläufigen Ausführung einer möglichen seriellen Ausführung der Transaktionen entspricht.

8.2.1 Definition einer Transaktion

Eine Transaktion T_i besteht aus folgenden elementaren Operationen:

- $r_i(A)$ zum Lesen des Datenobjekts A
- $w_i(A)$ zum Schreiben des Datenobjekts A
- a_i zum Durchführen eines **aborts**
- c_i zum Durchführen eines **commits**

Offensichtlich kann eine Transaktion nur eine der beiden Operation **abort** oder **commit** durchführen. Weiter ist eine Reihenfolge der Operationen zu spezifizieren, wobei in der Regel eine totale Ordnung angegeben wird, obschon eine partielle genügen würde. Mindestens müssen aber folgende Bedingungen eingehalten werden:

- Falls T_i ein **abort** durchführt, müssen alle anderen Operationen $p_i(A)$ vor a_i ausgeführte werden, also $p_i(A) <_i a_i$.
- Analog für **commit**, d.h. $p_i(A) <_i c_i$.
- Wenn T_i ein Objekt sowohl liest als auch schreibt, dann muss die Reihenfolge festgelegt werden, also entweder $r_i(A) <_i w_i(A)$ oder $w_i(A) <_i r_i(A)$.

Auf die Angabe von **BOT** wird verzichtet, eine Transaktion wird immer automatisch mit der ersten Operation gestartet.

8.2.2 Historie

Eine Historie H für eine Menge von Transaktionen $\{T_1, \dots, T_n\}$ ist eine Menge von Elementaroperationen mit partieller Ordnung $<_H$, so dass gilt:

- $H = \bigcup_{i=1}^{n} T_i$
- $<_H$ ist verträglich mit allen $<_i$, also $\bigcup_{i=1}^{n} <_i \subseteq <_H$
- Für zwei Konfliktoperationen $p, q \in H$ gilt entweder $p <_H q$ oder $q <_H p$.

Zwei Operationen von zwei Transaktionen heissen dabei *Konfliktoperationen*, wenn beide auf dasselbe Datenobjekt zugreifen, und mindestens eine davon dieses modifiziert. Solche Operationen können bei unkontrollierter, paralleler Ausführung zu Inkonsistenzen führen. Operationen *derselben* Transaktion stehen immer in Konflikt.

8.2.3 Äquivalenz zweier Historien

Zwei Historien H und H' über der gleichen Menge von Transaktionen sind äquivalent, $H \equiv H'$, wenn die folgenden beiden Bedingungen erfüllt sind:

- Alle **read**-Operationen (von Transaktionen die **commited** wurden) liefern dasselbe Resultat.
- Am Ende, der Zustand der Datenbank ist derselbe.

8.2.3.1 Konflikt-Äquivalenz

Zwei Historien H und H' über der gleichen Menge von Transaktionen sind konflikt-äquivalent (siehe auch Abschnitt 8.2.4.1 Konfliktserialisierbarkeit), $H \equiv H'$, wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen. Formaler ausgedrückt: Wenn p_i und q_i Konfliktoperationen sind und $p_i <_H q_j$ gilt, dann muss auch $p_i <_{H'} q_j$ gelten.

Die Anordnung der nicht in Konflikt stehenden Operationen ist also für die Konflikt-Äquivalenz irrelevant. Offensichtlich bleibt die Reihenfolge der Operationen innerhalb einer Transaktion invariant, d.h. zwei Operationen v_i und w_i der Transaktion T_i sind in H und H' in derselben Reihenfolge auszuführen.

8.2.4 Serialisierbare Historien

Eine Historie H ist serialisierbar, wenn sie äquivalent zu einer seriellen Historie H_s ist, das heisst genau dasselbe Ergebnis liefert. Um effizient entscheiden zu können, ob eine Historie serialisierbar ist, gibt es verschiedene hinreichende Kriterien.

8.2.4.1 Konfliktserialisierbarkeit

Ein solches Kriterium ist die sogenannte *Konfliktserialisierbarkeit*, die fordert, dass alle Konfliktoperationen in derselben Reihenfolge ausgeführt werden müssen, wie eine serielle Historie. Es ist offensichtlich,

dass dieses Kriterium hinreichend ist. Alle Operationen, die nicht in Konflikt stehen, sind nämlich nur lesend, oder betreffen andere Datenobjekte als die anderen Operationen. Diese anders anzuordnen ist natürlich kein Problem. Andererseits ist Konfliktserialisierbarkeit *nicht notwendig* für Serialisierbarkeit. Einfach zu veranschaulichen ist dies mithilfe des folgenden Beispiels:

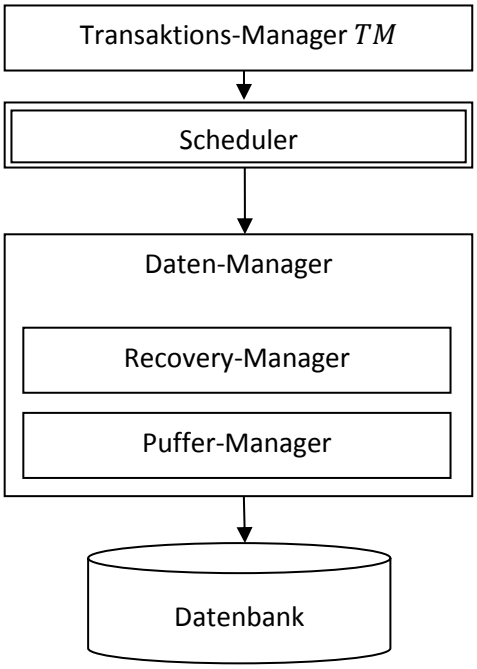
$$r_1(A), \underbrace{r_2(A)}_{=41}, w_2(A, 41), r_1(A)$$

Die erste Transaktion T_1 liest also zweimal den Wert von A, während T_2 dasselbe Datenobjekt zuerst liest, und dann *den gelesenen Wert schreibt* (dazu wurde etwas neue, aber völlig intuitive Notation eingeführt). Dadurch hat T_2 natürlich keinen Einfluss, da in der Datenbasis nichts verändert wird, und daher ist diese Historie serialisierbar, da das Ergebnis dasselbe ist, wie von der seriellen Historie, die zuerst T_1 und dann T_2 ausführt (oder auch umgekehrt).

Um für eine Historie zu entscheiden, ob diese konfliktserialisierbar ist, kann der sogenannte Serialisierbarkeitsgraph ($SG(H)$) verwendet werden. Die Knoten von $SG(H)$ sind die beteiligten Transaktionen $T_1, \dots, T_n$. Weiter gibt es für alle Paare von Konfliktoperationen p_i und q_j aus der Historie H mit $p_i <_H q_j$ (wenn also p_i vor q_j ausgeführt wird) eine Katen $T_i \rightarrow T_j$, falls es diese Kante nicht bereits gibt. Die Historie H ist nun genau dann konflikt-serialisierbar, wenn $SG(H)$ azyklisch ist.

8.3 Der Datenbank-Scheduler

Hinsichtlich der Transaktionsverarbeitung kann man sich eine Datenbank-Architektur mit einem *Scheduler* – stark vereinfacht – wie in der folgenden Abbildung gezeigt vorstellen:



Die Aufgabe des Schedulers besteht darin, die Operationen – d.h. Einzeloperationen verschiedener Transaktionen $T_1, \dots, T_n$ - in einer derartigen Reihenfolge auszuführen, dass die resultierende Historie „vernünftig“ ist. Unter „vernünftig“ versteht man in der Regel mindestens serialisierbar, oftmals sogar noch weitere, hier nicht betrachtete Bedingungen.

8.4 Sperrbasierte Synchronisation

Bei der sperrbasierten Synchronisation wird während des laufenden Betriebs sichergestellt, dass die resultierende Historie serialisierbar bleibt.

8.4.1 Zwei Sperrmodi

Je nach Operation (**read** oder **write**) wird zwischen zwei Sperrmodi unterschieden:

- S (shared, read lock, Lesesperre): Wenn Transaktion T_i eine S-Sperre für ein Datum A besitzt, kann T_i ein **read** auf A ausführen. Mehrere Transaktionen können gleichzeitig eine S-Sperre auf demselben Objekt besitzen.
- X (exclusive, write lock, Schreibsperre): Ein **write** auf A darf nur die *eine* Transaktion ausführen, welche die X-Sperre auf A besitzt.

Die folgende *Verträglichkeitsmatrix* fasst nochmals zusammen, wann eine Sperre mit einer bereits existierenden verträglich ist:

Request:

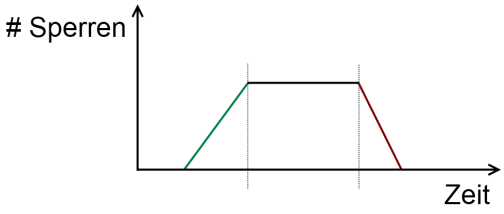
	No lock	**S**	**X**
S	ok	ok	-
X	ok	-	-

8.4.2 Zwei-Phasen-Sperrprotokoll

Die Serialisierbarkeit ist bei Einhaltung des folgenden Zwei-Phasen-Sperrprotokolls (*two phase locking*, 2PL) durch den Scheduler gewährleistet. Bezogen auf eine individuelle Transaktion wird folgendes verlangt:

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an. Ein „Aufwerten“ der Sperre (von einer Lesesperre zu einer Schreibsperre) ist aber durchaus möglich.
3. Die Verträglichkeitstabelle muss eingehalten werden. Wenn eine Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht, bis die Sperre verfügbar ist.
4. Jede Transaktion durchläuft zwei Phasen:
    a. Eine *Wachstumsphase*, in der sie Sperren anfordert, aber keine freigeben darf.
    b. Eine *Schrumpfungsphase*, in der erworbene Sperren wieder freigegeben, aber keine neuen Sperren angefordert werden.
5. Bei **EOT** (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben.

Alle Historien, die durch 2PL generiert werden, sind serialisierbar.

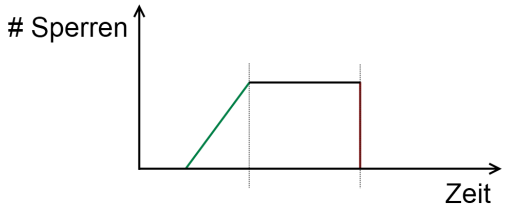


8.4.3 Striktes Zwei-Phasen-Sperrprotokoll

2PL garantiert in der einfachen Form zwar bereits Serialisierbarkeit, aber es vermeidet nicht das sogenannte *kaskadierende Rücksetzen*. Dabei wird durch den Abbruch einer Transaktion (zum Beispiel durch ein **abort**) eine Transaktion in Mitleidenschaft gezogen, die bereits **commited** wurde. Dieses Verhalten kann einen Schneeballeffekt auslösen, und ist offensichtlich nicht erwünscht.

Die Lösung besteht darin, das 2PL-Protokoll zum sogenannten *strengen 2PL-Protokoll* wie folgt zu verschärfen:

- Die ersten fünf Anforderungen bleiben erhalten.
- Es gibt keine Schrumpfungsphase mehr, sondern *alle* Sperren werden erst zum Ende der Transaktion (**EOT**) freigegeben.



Unter Einhaltung dieser Verschärfung entspricht die Reihenfolge, in der die Transaktionen beendet werden, einer äquivalenten seriellen Abarbeitungsreihenfolge (*commit order serializability*).

8.4.4 Deadlocks (Verklemmung)

Deadlocks sind ein schwerwiegendes und inherentes Problem bei sperrbasierten Synchronisationsmethoden. Dabei blockieren sich zwei Transaktionen gegenseitig, indem sie beide auf ein Lock warten, das von der jeweils anderen Transaktion gehalten wird.

8.4.4.1 Erkennung von Deadlocks

Die wohl einfachste Möglichkeit, Deadlocks versuchen zu erkennen, ist eine Time-out Strategie. Man wählt ein Zeitlimit, und wenn während dieser Zeit eine Transaktion keinen Fortschritt macht, dann wird angenommen, dass ein Deadlock vorliegt. In diesem Fall wird die Transaktion zurückgesetzt. Offensichtlich ist dies keine besonders genaue Strategie, und so gibt es gravierende Nachteile: Wird das Limit zu klein gewählt, werden Transaktionen abgebrochen, obwohl eigentlich gar kein Deadlock vorliegt. Andererseits, wenn das Zeitmass zu gross gewählt wird, so wird unnötig lange gewartet, wenn wirklich ein Deadlock vorliegt.

Eine präzise, aber auch teure Methode Deadlocks zu erkennen basiert auf dem sogenannten Wartegraphen. Die Knoten in diesem Graphen sind die gerade aktiven Transaktionen, und wenn immer eine Transaktion T_a auf die Freigabe einer Sperre durch eine Transaktion T_b wartet, gibt es eine (gerichtete) Kante von T_a nach T_b. Ein Deadlock liegt nun genau dann (und nur dann) vor, wenn es einen Zyklus in diesem Graphen gibt. Um dieses Deadlock zu beheben, genügt es, eine Transaktion im Zyklus zurückzusetzen, wobei die Wahl aufgrund verschiedener Kriterien erfolgen kann:

- Minimierung des Rücksetzaufwandes
- Maximierung der freigegebene Ressourcen (sodass nicht gleich wieder ein Deadlock entsteht)
- Vermeiden von Starvation

- Lösen von mehrfachen Zyklen

8.4.4.2 Vermeidung von Verklemmung

Eine einfache Möglichkeit Deadlocks ganz zu vermeiden ist *Preclaiming*. Dabei wird eine Transaktion erst begonnen, wenn alle nötigen Sperren vorhanden sind, sodass bei Transaktionsbeginn gleich alle Sperren angefordert werden können. Das grosse Problem ist, dass dafür bekannt sein muss, was für Sperren von der Transaktion überhaupt benötigt werden, was in der Praxis oft nicht der Fall ist. Daher muss man eine Obermenge der Sperren beantragen, was zu einer Einschränkung der Parallelität führt.

Eine andere Möglichkeit sind Zeitstempel. Dabei werden Zeitstempel monoton wachsend vom Transaktionsmanager an jede Transaktion vergeben. Wenn nun T_1 eine Sperre anfordert, die T_2 aber erst freigeben müsste, gibt es zwei in der Wirkung sehr unterschiedliche Alternativen:

- *wound-wait*: Wenn T_1 älter ist als T_2, so wird T_2 abgebrochen und zurückgesetzt, so dass T_1 weiterlaufen kann. Sonst wartet T_1 auf die Freigabe der Sperre.
- *wait-die*: Wenn T_1 älter ist als T_2, so wartet T_1 auf die Freigabe, andernfalls wird T_1 zurückgesetzt.

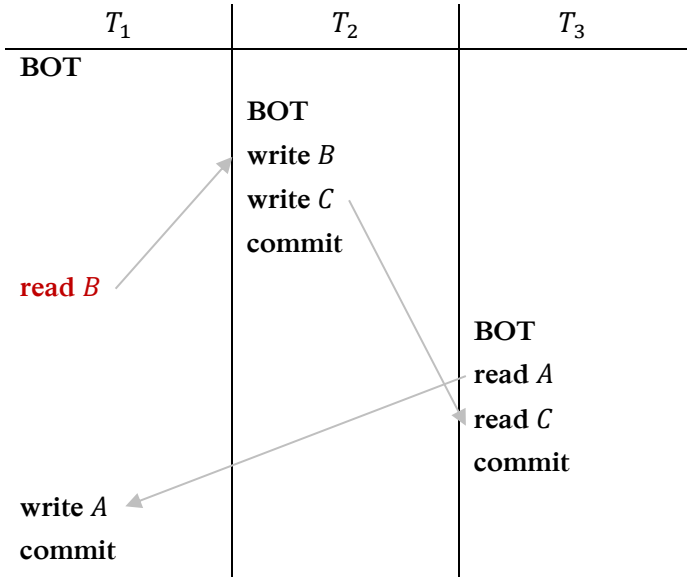
Diese Methode ist garantiert Verklemmungsfrei, es werden aber im Allgemeinen zu viele Transaktionen zurückgesetzt.

8.5 Snapshot Isolation (SI)

Bei *Snapshot Isolation* scheint jede Transaktion auf einem Schnappschuss der Datenbank zu arbeiten, der zu Beginn der Transaktion aufgenommen wird. Jegliche Operationen werden sofort auf diesem Schnappschuss ausgeführt. Am Ende der Transaktion wird diese erfolgreich **commited**, wenn kein Datum, welches von der Transaktion *verändert* wurde, auch extern verändert wurde. Ein solcher write-write-Konflikt führt zum Abbruch der Transaktion. Snapshot Isolation garantiert also, dass sämtliche Leseoperationen einen konsistenten Schnappschuss der Datenbank sehen.

8.5.1 Serialisierbarkeit

Snapshot Isolation bietet hingegen *keine Serialisierbarkeit*. Dies lässt sich an folgendem Beispiel sehen:



Snapshot Isolation lässt diese Historie zu, obwohl diese nicht serialisierbar ist, wie an den grauen Pfeilen (vergleiche Serialisierbarkeitsgraph) erkennbar ist. Vergleicht man das Verhalten von SI mit 2PL, so stellt man leicht fest, dass die Historie auch von 2PL akzeptiert wird, mit einem kleinen, aber signifikaten Unterschied: Die Leseoperation von T_1 auf B (rot markiert) wird bei SI der Wert von ganz zu Beginn gelesen, während 2PL den Wert, welcher von T_2 geschrieben wurde, sieht. Daher ist die Historie mit 2PL serialisierbar (die grauen Pfeile gelten nur für SI, für 2PL ist der Pfeil oben links in die entgegengesetzte Richtung).

8.5.2 write-skew Anomalität

Eine sogenannte *write-skew Anomalität* tritt auf, wenn zwei Transaktionen T_1 und T_2 gleichzeitig zwei Datenobjekte (z.B. A und B) lesen, gleichzeitig disjunkte Veränderungen machen (z.B. T_1 ändert A und T_2 ändert B). Wenn beide Transaktionen **commiten**, haben beide die Änderung der jeweils anderen Transaktion nicht gesehen. In einem serialisierbaren System könnte dies nicht passieren, da entweder T_1 oder T_2 „zuerst“ hätte stattfinden müssen, und die andere Transaktion dann das Update gesehen hätte.

Konkret könnte dies so aussehen: In einer Praxis gibt es zwei Ärzte, wobei immer mindestens einer im Dienst sein muss. Zu Beginn sind beide im Dienst, doch nun beginnen beide gleichzeitig, sich abzumelden. Da beide auf einem Schnappschuss arbeiten, in dem zwei Ärzte im Dienst sind, können sich beide erfolgreich abmelden, und so werden beide Transaktionen erfolgreich **commited**, sodass nun die Bedingung verletzt wird. Dieses Beispiel könnte gelöst werden, indem die Bedinung als Teil des **commits** geprüft wird.

8.5.3 Performance

Der Grund, weshalb Snapshot Isolation von vielen Datenbankherstellern verwendet wird, ist die erhöhte Performance im Vergleich zu Serialisierbarkeit, ohne Anomalien zu erzeugen, die in der Praxis nur schwer umgangen werden können. Bei Snapshot Isolation kann es zu keinen Deadlocks kommen, dafür gibt es aber unnötige Rollbacks. Der Overhead um alle Versionen eines Objektes zu speichern (für den Schnappschuss) ist oftmals vernachlässigbar, da dies sowieso nötig ist.

8.6 Isolationslevel in SQL92

Isolation in Datenbanksystemen ist eine Eigenschaft die bestimmt, wie und wann Änderungen einer Transaktion für andere Transaktionen sichtbar werden (vergleiche ACID).

set transaction
[read only, |read write,]
[**isolation level**]
read uncommitted, |
read commited, |
repeatable read, |
serializable,]
[**diagnostic size** ...,]

Diese Level wurden für sperrbasierte Datenbanken festgelegt, was zu einigen Inkonsistenzen führt, beispielsweise im Zusammenhang mit Snapshot Isolation (siehe weiter unten). Die Level sind folgendermassen definiert:

- *read uncommited*: Lesevorgänge benötigen kein *S*-Lock, und werden auch nicht von exklusiven Sperren aufgehalten. Daher sind sogenannte *dirty reads* möglich, wobei eine Transaktion Änderungen einer Transaktion sieht, die noch nicht **commited** wurde.
- *read commited*: Datenobjekte, die von einer Transaktion gelesen werden, werden nicht vor Änderung durch andere Transaktionen geschützt. Read locks werden zwar angefordert, aber sofort wieder freigegeben, während die write locks erst am Ende der Transaktion freigegeben werden. Im Unterschied zu *read uncommited* sind hierbei aber nur Änderungen von Transaktionen die bereits **commited** wurden, sichtbar.
- *repeatable read*: Datenobjekte werden beim Lesen mit der entsprechenden Sperre versehen, dennoch sind Phantome möglich, da keine *range locks* angefordert werden.
- *serializable*: Vollständige Isolation, sodass Serialisierbarkeit garantiert ist. Allerdings implementieren viele Hersteller wie Oracle oder PostgreSQL hier Snapshot Isolation, welches keine Serialisierbarkeit garantiert.

	Range lock	**Read lock**	**Write lock**
Read uncommited	-	-	-
Read commited	-	-	x
Repeatable read	-	x	x
Serializable	x	x	x

	Dirty reads	**Non-repeatable reads**	**Phantoms**
Read uncommited	x	x	x
Read commited	-	x	x
Repeatable read	-	-	x
Serializable	-	-	-

9 Datenorganisation und Anfragebearbeitung

9.1 Der Datenbankpuffer

Alle Operationen auf Daten müssen innerhalb des Hauptspeichers ausgeführt werden, wo sie im sogenannten *Datenbankpuffer* verwaltet werden. Es ist natürlich sinnvoll, Seiten auch länger im Hauptspeicher zu halten als nur für den Zeitraum der Operation, da man meistens im Verhalten der Anwendung eine *Lokalität* feststellt. Dabei wird mehrmals hintereinander auf dieselben Daten zugegriffen.

Da der Hauptspeicher aber nicht nur wesentlich schneller, sondern auch deutlich kleiner ist, müssen Seiten auch wieder aus dem Puffer entfernt werden. Dazu wird eine *Ersetzungsstrategie* eingesetzt, die idealerweise stets eine Seite entfernt, die möglichst lange nicht mehr gebraucht wird. Möglichkeiten dafür sind die folgenden:

- LRU, Clock
- LRU-k, wobei die Seite, welche am k wenigsten verwendet wurde, zuerst ersetzt wird
- 2Q mithilfe von zwei Queues

9.2 Speicherung von Relationen im Sekundärspeicher

Für jede Relation werden mehrere Seiten auf dem Hintergrundspeicher zu einer Datei zusammengefasst. Die Tupel einer Relation werden in den Seiten der Datei so gespeichert, dass sie nicht über eine Seitengrenze hinausgehen. Ausser einem Geschwindigkeitsverlust würde dies auch Probleme bei der Adressierung, der Mehrbenutzersynchronisation und der Fehlerbehandlung hervorrufen.

Um ein bestimmtes Tupel direkt referenzieren zu können, verwendet man einen sogenannten *Tupel-Identifikator* (TID). Ein TID besteht dabei aus zwei Teilen: Einer Seitennummer und einer Nummer eines Eintrages in der internen Datensatztabelle, der auf das entsprechende Tupel verweist. Diese zusätzliche Indirektion ist nützlich, da so jede Seite intern reorganisiert werden kann.

Falls auf einer Seite kein Platz mehr vorhanden ist, kann an dieser Stelle ein Platzhalten eingeführt werden, wobei einfach eine TID gespeichert wird, die auf den eigentlichen Speicherort zeigt. So ist man sehr flexibel, und benötigt dennoch höchstens zwei Seitenzugriffe.

9.3 Anfragebearbeitung

Zunächst wird eine Anfrage syntaktisch und semantisch analysiert und in einen äquivalenten Ausdruck der relationalen Algebra (beziehungsweise einer Erweiterung der relationalen Algebra) umgewandelt. Hierbei werden vorkommende Sichten auch durch ihre definierende Anfrage ersetzt.

Konkret wird ein Graph von Bäumen im sogenannten *Query Graph Model* erzeugt, wobei die Knoten (Bäume) Anfragen darstellen, und die Kanten Verbindungen dazwischen darstellen.

Mit der relationalen Algebra als Eingabe wird die Anfrageoptimierung gestartet, wobei ein möglichst effizienter Auswertungsplan gesucht wird. Dieser kann dann entweder kompiliert oder bei interaktiven Anfragen auch direkt interpretativ gestartet werden.

9.4 Algorithmen

9.4.1 Zwei-Phasen externes Sortieren

In einer ersten Phase werden sogenannte *Läufe* (*runs*) erstellt, indem der Puffer mit Tupeln gefüllt wird, und dann diese im Hauptspeicher sortiert werden. Darauf werden diese wieder in den Hintergrundspeicher geschrieben, bis alle Tupel behandelt wurden. In der zweiten Phase wird dann eine Priority-Queue verwendet, um die Tupel zu verschmelzen.

Wenn n die Eingabegrösse in Seiten ist, und b die Puffergrösse darstellt, gibt es folgende Spezialfälle:

- $b \geq n$: Phase Zwei ist nicht nötig
- $b < \sqrt{n}$: Mehrere Verschmelzungsphasen sind nötig.

Die Komplexität für das Sortieren beträgt $n \cdot \log(n)$.

9.4.2 (Grace) Hash Join

Die Idee des Hash-Joins besteht darin, die Eingabedaten so zu partitionieren, dass die Verwendung einer Hauptspeicher-Hashtabelle möglich ist. Zuerst werden beide Relationen gemäss einer Hashfunktion h partitioniert, und diese Partitionen werden dann wieder auf den Hintergrundspeicher geschrieben.

Dann werden jeweils Paare von Partitionen in den Hauptspeicher geladen, und für die Relation mit weniger Tupeln in der momentanen Partition wird eine Hashtabelle mit einer Hashfunktion h' gebildet. Dann wird mit allen Tupeln der anderen Partition mit Hilfe der Hashtabelle getestet, und so werden alle Join-Partner gefunden. Falls die Hashtabelle nicht im Hauptspeicher Platz findet, kann das Verfahren rekursiv angewendet werden.

9.5 Iteratoren-Modell

Eine elegante Lösung um Auswertungspläne modular zusammenzusetzen, stellen die sogenannten *Iteratoren* dar. Ein Iterator ist ein abstrakter Datentyp, der Operationen wie **open**, **next** und **close** zur Verfügung stellt. Die Operation **open** ist dabei eine Art Konstruktor, welcher die Eingaben öffnet und eventuell eine Initialisierung vornimmt. Die Schnittstellenoperation **next** liefert das nächste Tupel des Ergebnisses, und **close** schliesst die Eingaben wieder.

Jeder Operator der relationalen Algebra wird nun als Iterator implementiert, sodass diese beliebig kombiniert werden können.

10 Security

Die Schutzmechanismen eines DBMS können in drei Kategorien unterteilt werden:

- **Identifikation und Authentisierung.** Um Zugang zu einem Datenbanksystem zu erhalten, muss der User sich in der Regel identifizieren (z.B. durch Eingeben eines Benutzernamens) und authentisieren (üblicherweise durch ein Passwort) um sicherzustellen, dass der Benutzer auch wirklich derjenige ist, für den er sich ausgibt.
- **Autorisierung und Zugriffskontrolle.** Eine Autorisierung besteht aus einer Menge von Regeln, die die erlaubten Arten des Zugriffs auf *Sicherheitsobjekte* durch *Sicherheitssubjekte* festlegt.
- **Auditing.** Um die Richtigkeit und Vollständigkeit der Autorisierungsregeln zu verifizieren, und Schäden rechtzeitig zu erkennen, kann über jede sicherheitsrelevante Datenbankoperation Buch geführt werden.

Datensicherheit setzt sich grundsätzlich aus zwei Aspekten zusammen: *confidentiality* und *integrity*. Daten sollen also vor unerlaubtem Zugriff und unerlaubter Änderung geschützt werden.

Um einen Schutz von sicherheitsrelevanten Daten zu gestalten, müssen die Schwachstellen eines Systems bekannt sein. Typische Arten von Angriffen sind dabei:

- **Missbrauch von Autorität.** Diebstahl, Veränderung oder Zerstörung von Daten oder Programmen.
- **Inferenz und Aggregation.** Inferenz bezeichnet das Schliessen auf sensitive Daten durch Ansammlung und Kombination von nicht sensitiven Daten. Dabei können auch Daten von ausserhalb des Datenbanksystems eine Rolle spielen. Umgekehrt bezeichnet Aggregation den Fall, dass einzelne Daten nicht sensitiv sind, aber eine grosse Anzahl von Daten zusammen schon.
- **Maskierung.** Unautorisierter Zugriff auf Daten durch jemanden, der sich als ein autorisierter Benutzer ausgibt.
- **Umgehung der Zugriffskontrolle.** Ausnutzung von Sicherheitslücken im Betriebssystemcode oder in Anwendungsprogrammen.
- **Browsing.** Geschützte Informationen können manchmal auch durch Betrachten des Datenwörterbuchs oder von Dateiverzeichnissen erhalten werden.
- **Trojanische Pferde.**
- **Versteckte Kanäle.** Der Zugriff auf Informationen durch nicht bestimmungsgemässe Kanäle, wie z.B. des direkte Auslesen einer Datebankdatei unter Umgehung des Datenbankverwaltungssystems.

In diesem Kapitel werden zwei grundlegende Sicherheitsstragegien vorgestellt: die *discretionary access control (DAC)* und die *mandatory access control (MAC)*. Bei der DAC werden Regeln zum Zugriff auf Objekte angegeben, während die MAC zusätzlich den Fluss der Informationen zwischen Objekten und Subjekten regelt.

10.1 Discretionary Access Control

Die Zugriffsregeln der DAC geben zu einem Subjekt s die möglichen Zugriffsarten t auf ein Objekt o an. Formal ausgedrückt ist eine Regel ein Quintupel (o, s, t, p, f), wobei

- $o \in O$, der Menge der Objekte (z.B. Relationen, Tupel oder Attribute)

- $s \in S$, der Menge der Subjekte (z.B. Benutzer oder Prozesse)
- $t \in T$, der Menge der Zugriffsrechte (z.B. $T = \{\text{lesen}, \text{schreiben}, \text{löschen}\}$)
- p ein Prädikat, das eine Art Zugriffsfenster auf o festlegt (z.B. $\text{Rang} = 'C4'$ für eine Relation Professoren)
- f ein boolscher Wert ist, der angibt, ob s das Recht (o, t, p) an ein anderes Subjekt s' weitergeben darf.

Implementiert werden solche Regeln oftmals mithilfe einer Zugriffsmatrix: Die Subjekte werden in den Zeilen der Matrix abgelegt, und die Objekte in den Spalten. Je nach Granularität der Autorisierung werden solche Matrizen jedoch schnell sehr gross.

DAC geht davon aus, dass die Erzeuger der Daten auch deren Eigner und damit verantwortlich für deren Sicherheit sind, wobei die Erzeuger freie Hand haben, diese Rechte auch an Dritte weiterzugeben. In der Praxis ist diese Annahme aber oft nicht gegeben, beispielsweise in einem Firmenumfeld.

10.2 Zugriffskontrolle in SQL

10.2.1 Autorisierung und Zugriffskontrolle

Eine Autorisierung erfolgt mit dem grant-Kommando. Beispielsweise könnte dies folgendermassen aussehen:

```
grant select (name)
    on students
    to username;
```

Neben **select** exisiteren auch noch die Standardprivilegien **delete**, **insert**, **update** und **references**. Die Rechte **insert**, **update** und **references** lassen eine Qualifizierung der Attribute zu, auf denen das Recht besteht.

Das **references**-Privileg erlaubt Benutzern, Fremdschlüssel auf das spezifizierte Attribut anzulegen. Das ist wichtig, da Benutzer durch Fremdschlüssel anderer aufgrund der referentiellen Integrität am Löschen von Tupeln in ihrer eigenen Relation gehindert werden. Zudem kann aus demselben Grund durch geschicktes Testen die Schlüsselwerte einer ansonsten lesegeschützten Tabelle herausgefunden werden.

Das Recht zur Weitergabe von Privilegien an andere Benutzer wird durch das Anhängen von **with grant option** gewährt. Das Entziehen eines Rechtes erfolgt über eine **revoke**-Anweisung. Bei diesem Befehl kann man durch anhängen von **restrict** das Datenbanksystem dazu bringen, mit einer Fehlermeldung abzubrechen, wenn das Recht weitergegeben wurde. Mit **cascade** werden kaskadierend alle Rechte zurückgenommen, die aus dem Weitergaberecht entstanden sind.

10.2.2 Sichten

Im DAC-Modell besteht die Möglichkeit, ein Recht von einer bestimmten Bedingung abhängig zu machen. In SQL wird dies durch Sichten realisiert, z.B. folgendermassen:

```
create view FirstSemester as
        select *
        from students
        where semester = 1;
```

Sichten sind auch geeignet, um Daten zu aggregieren. Dadurch können schützenswerte Individualdaten den Benutzern verborge bleiben, wohingegen aggregierte, einen Überblick vermittelnde Daten den Benutzern zugänglich gemacht werden.

10.3 Verfeinerung des Autorisierungsmodells

Bisher wurde nur die *explizite Autorisierung* behandelt, bei welcher einzelne Objekte autorisiert werden. Existieren aber sehr viele Objekte, so entstehen Probleme, wobei die sogenannte *implizierte Autorisierung* helfen soll.

Für implizite Autorisierung werden Subjekte, Objekte und Operationen hierarchisch angeordnet. Eine Autorisierung auf einer bestimmten Stufe der Hierarchie bewirkt implizite Autorisierungen auf anderen Stufen der Hierarchie.

Als Gegenstück zur *positiven Autorisierung*, die einen Zugriff erlaubt, kann auch eine *negative Autorisierung* eingeführt werden, die ein Verbot des Zugriffs darstellt, wobei auch hierbei zwischen positiv und negativ unterschieden wird. Negative Autorisierungen werden durch $\neg$ gekennzeichnet. Wenn die Regel (o, s, t) dem Subjekt s den Zugriff t auf ein Objekt o erlaubt, dann ist die entsprechende negative Autorisierung $(o, s, \neg t)$.

Zuletzt wird zwischen *schwacher* und *starker Autorisierung* unterschieden. Eine schwache Autorisierung kann dabei als Standardeinstellung verwendet werden. Beispielsweise kann eine in verschiedene andere Benutzergruppen unterteilte Benutzergruppe Alle standardmässig das schwache Recht zum Lesen eines Objektes erhalten. Der Teil von Alle allerdings, der in der Gruppe Aushilfe ist, erhält ein starkes Verbot zum Lesen des Objektes. Ohne die Unterscheidung zwischen starker und schwacher Autorisierung hätten alle Gruppen in Alle explizit ein Recht oder Verbot erhalten müssen. Im Folgenden wird eine starke Autorisierung mit runden Klammern $(\dots)$ und eine schwache mit eckigen Klammern $[\dots]$ notiert.

Der Algorithmus zur Überprüfung einer Autorisierung kann wie folgt formuliert werden:

wenn es seine explizite oder implizite starke (positive) Autorisierung (o, s, t) gibt,
  dann erlaube die Operation
wenn es seine explizite oder implizite starke negative Autorisierung $(o, s, \neg t)$ gibt,
  dann verbiete die Operation
ansonsten
  wenn es seine explizite oder implizite schwache (positive) Autorisierung $[o, s, t]$ gibt,
    dann erlaube die Operation
  wenn es seine explizite oder implizite schwache negative Autorisierung $[o, s, \neg t]$ gibt,
    dann verbiete die Operation

Es wird vorausgesetzt, dass keine Konflikte zwischen den Regeln vorhanden sind.

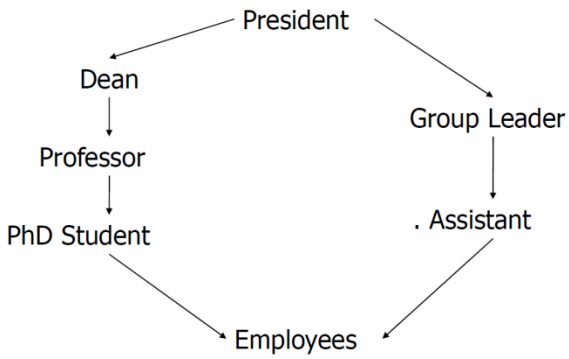
10.3.1 Rollenbasierte Autorisierung

Für eine implizite Autorisierung von Subjekten werden so genannte *Rollen* und *Rollenhierarchien* eingeführt. Bei der rollenbasierten Autorisierung (oft RBAC für *role-based access control*) werden Benutzern Rollen zugeordnet. Zugriffsrechte werden dabei nicht mehr direkt den Benutzern gewährt (oder verboten), sondern diesen Rollen zugeordnet.

Die Rollen können in sogenannte *Rollenhierarchien* strukturiert werden, wobei es stets eine eindeutige Rolle mit der maximalen Menge an Rechten und eine eindeutige, grundlegende Rolle, gibt. Dabei gelten folgende Regeln:

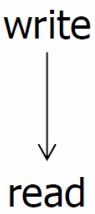
- Eine explizite positive Autorisierung auf einer Stufe resultiert in einer impliziten positiven Autorisierung auf allen höheren Stufen.
- Eine explizite negative Autorisierung auf einer Stufe resultiert in einer impliziten negativen Autorisierung auf allen niedrigeren Stufen.

Eine solche Hierarchie könnte beispielsweise wie folgt aussehen:



10.3.2 Implizite Autorisierung von Operationen

Analog können auch Operationshierarchien festgelegt werden. Die Regeln hierfür sind jedoch genau umgekehrt:



- Eine explizite positive Autorisierung auf einer Stufe resultiert in einer impliziten positiven Autorisierung auf allen *niedrigeren* Stufen. Eine Schreibberechtigung impliziert also eine Leseberechtigung, das eine Schreiboperation im Allgemeinen auch eine Leseoperation beinhaltet.
- Eine explizite negative Autorisierung auf einer Stufe resultiert in einer impliziten negativen Autorisierung auf allen *höheren* Stufen. Wenn nicht gelesen werden kann, so kann auch nicht geschrieben werden.

10.3.3 Implizite Autorisierung von Objekten

Auch für Objekte, etwa die Datenbank, Schemata, Relationen, Tuppel oder Attribute kann eine solche Hierarchie festgelegt werden. Die Implikationen hängen hierbei jedoch von den Operationen ab.

10.3.4 Implizite Autorisierung entlang einer Typenhierarchie

Eine zusätzliche Dimension für implizite Autorisierung bieten Typenhierarchien, die durch Generalisierung entstehen. Hierbei können drei Regeln formuliert werden:

- Benutzer mit einem Zugriffsrecht auf einen Objekttypen haben auf die geerbten Attribute in den Untertypen ein gleichartiges Zugriffsrecht.
- Ein Zugriffsrecht auf einen Objekttypen impliziert auch ein Zugriffsrecht auf alle von Obertypen geerbten Attributen in diesem Typ.
- Ein Attribut, das in einem Untertyp definiert wurde, ist nicht von einem Obertyp aus erreichbar.

Diese Regeln lassen sich anhand von folgenden Beispielen verdeutlichen:

- Ein Recht auf $A.x$ impliziert auch ein Recht auf $B.x$ wenn B ein Untertyp von A ist. Beispielsweise darf der Name von Professoren gelesen werden, wenn der Name von Angestellten gelesen werden darf.
- Geerbte Attribute können gelesen werden.
- Ein Recht auf dem Typ A impliziert *nicht* das Recht für $B.x$, wenn B ein Untertyp von A ist und x in B definiert wurde. Wenn die Relation $\mathrm{Angestellte}$ gelesen werden kann, dann darf deshalb nicht $\mathrm{Professor.Level}$ gelesen werden.

10.4 Mandatory Access Control

Besonders in militärischen Einrichtungen ist es üblich, Dokumente nach ihrere Sicherheitsrelevanz hierarchisch zu klassifizieren, etwas „streng geheim“, „geheim“, „vertraulich“ und „unklassifiziert“. Im MAC-Modell erhalten alle Subjekte und Objekte eine Markierung mit ihrere Sicherheitseinstufung. Bei Subjekten ist dies die Vertrauenswürdigkeit (bezeichnet mit $clear(s)$), bei Objekten deren Sensitivität ($class(o)$). Es gilt:

- Ein Subjekt s darf ein Objekt o nur lesen, wenn das Objekt eine geringere Sicherheitseinstufung besitzt, also $class(o) \leq clear(s)$.
- Ein Objekt o muss mit mindestens der Einstufung des Subjektes s geschrieben werden; $clear(s) \leq class(o)$.

Die zweite Regel wird zur Kontrolle des Informationsflusses verwendet, und soll Missbrauch durch autorisierte Benutzer verhindern.

10.5 Multilevel-Datenbanken

Es ist wünschenswert, dass Benutzer nicht wissen, auf welche Informationen sie keinen Zugriff haben. Haben aber nicht alle Benutzer Zugriff auf alle Tupel einer Tabelle, so gibt es neue Probleme: Was, wenn ein User ein Tupel einfügen möchte, das aus seiner Sicht noch nicht in der Tabelle vorkommt, aber trotzdem (mit einer höheren Sicherheitseinstufung) bereits vorhanden ist? Die Lösung nennt sich *Polyinstanziierung*. Dabei darf ein Tupel mehrfach, mit unterschiedlichen Sicherheitseinstufungen vorkommen. Eine Datenbank mit Polyinstanziierung heisst *Multilevel-Datenbank*, da sie sich Benutzern mit unterschiedlichen Einstufungen unterschiedlich darstellt.

Es gibt noch weitere Beispiele, wo Polyinstanziierung nötig ist. Da **null** nun nicht mehr nur für unbekannt stehen kann, sondern auch bedeuten kann, dass das Attribut eine höhere Klassifizierung hat, muss beim Verändern ebenfalls Polyinstanziierung verwendet werden. Ebenso, wenn ein als „streng geheim“ eingestufter Benutzer eine Veränderung an einem geheimem Tupel vornimmt, müssen die geheimen Tupel gestehen bleiben, und weitere streng geheime Tupel eingefügt werden. Diese Notwendigkeit ergibt sich direkt aus der zweiten Regel des MAC-Modells, gemäss welcher keine Informationen herunterklassifiziert werden können.

In einer Multilevel-Datenbank lassen sich aufgrund der Polyinstanziierung natürlich nicht die normalen Integritätsbedingungen des einfachen relationen Modells anwenden.

Das Schema einer Multilevel-Relation wird wie folgt beschrieben:

$$\mathcal{R} = \{A_1, C_1, A_2, C_2, \dots, A_n, C_n, TC\}$$

Dabei sind die A_i Attribute, und die C_i repräsentieren die Klassifizierung dieser Attribute. TC bezeichnet die Klassifizierung des gesamten Tupels.

Eine Multilevel-Relation wird dann, je nach Zugriffsklasse c, durch die *Relationeninstanz* R_c repräsentiert. R_c ist eine Menge unterschiedlicher Tupel der Form $[a_1, c_1, \dots, a_n, c_n, tc]$ mit $tc \geq c$. Ein a_i ist sichtbar, wenn die Zugriffsklasse grösser oder gleich c_i ist, ansonsten ist der Wert **null**.

In einer Multilevel-Relation heisst der benuztedefinierte Schlüssel *sichtbarer Schlüssel*. Sei κ der sichtbare Schlüssel einer Multilevel-Relation R mit dem wie oben definierten Schema $\mathcal{R}$. Dann werden die folgenden Integritätsbedingungen gefordert:

Entity-Integrität. R erfüllt die Entity-Integrität genau dann, wenn für alle Instanzen R_c und $r \in R_c$ die folgenden Bedingungen gelten:

1. $A_i \in \kappa \Rightarrow r.A_i \neq \text{null}$
2. $A_i, A_j \in \kappa \Rightarrow r.C_i = r.C_j$
3. $A_i \notin \kappa \Rightarrow r.C_i \geq r.C_\kappa$, wobei C_κ die Zugriffsklasse des Schlüssels ist

Ein Schlüsselattribut darf also keinen Nullwert beinhalten, und alle Schl2usselattribute müssen die gleiche Klassifizierung haben. Dies ist nötig, damit eindeutig bestimmt werden kann, ob ein Zurgriff auf ein Tupel möglich ist. Nichtschlüssel-Attribute müssen mindestens die Zugriffsklasse des Schlüssels besitzen, andernfalls könnte ein nicht-identifizierbares Tupel ein Attribut mit einem Wert ungleich **null** besitzen.

Null-Integrität. R erfüllt die Null-Integrität genau dann, wenn für jede Instanz R_c gilt:

1. $\forall r \in R_c, r.A_i = \text{null} \Rightarrow r.C_i = r.C_\kappa$
2. R_c ist subsumierungsfrei, d.h. es existieren keien zwei Tupel r und s, bei denen für alle Attribute entweder
   a. $r.A_i = s.A_i$ und $r.C_i = s.C_i$ oder
   b. $r.A_i \neq \text{null}$ und $s.A_i = \text{null}$ gilt.

Damit erhalten Nullwerte immer die Klassifizierung des Schlüssels. Die Subsumtionsfreiheit bewirkt das „Verschlucken“ von Tupeln, über die schon mehr bekannt ist.

Interinstanz-Integrität. R erfüllt die Interinstanz-Integrität genau dann, wenn für alle Instanzen R_c und $R_{c'}$ mit $c' < c$

$$R_{c'} = f(R_c, c')$$

Gilt. Die Filterfunktion f arbeitet wie folgt:

1. Für jedes $r \in R_c$ mit $r.C_\kappa \leq c'$ muss ein Tupel $s \in R_{c'}$ existieren, mit
$$s.A_i = \begin{cases} r.A_i & \text{wenn } r.C_i \leq c' \\ \text{null} & \text{sonst} \end{cases}$$
$$s.C_i = \begin{cases} r.C_i & \text{wenn } r.C_i \leq c' \\ r.C_\kappa & \text{sonst} \end{cases}$$
2. $R_{c'}$ enthält ausser diesen keine weiteren Tupel.
3. Subsummierte Tupel werden eliminiert.

Mit dieser Regel wird die Konsistent zwischen den einzelnen Instanzen der Multilevel-Relation gewährleistet.

Polyinstanziirungsintegrität. R erfüllt die Polyinstanziierungsintegrität genau dann, wenn für jede Instanz R_c für alle A_i die folgende funktionale Abhängigkeit gilt: $\{\kappa, C_\kappa, C_i\} \rightarrow A_i$. Diese Bedingung entspricht der Schlüsselintegrität im normalen relationalen Modell: Ein Tupel ist eindeutig bestimmt, wenn der Schlüssel und die Klassifizierung aller Attribute bekannt sind.