Informationsmanagement: Datenbanken

Zusammenfassung

Fabian Damken

6. März 2022

TECHNISCHE UNIVERSITÄT DARMSTADT

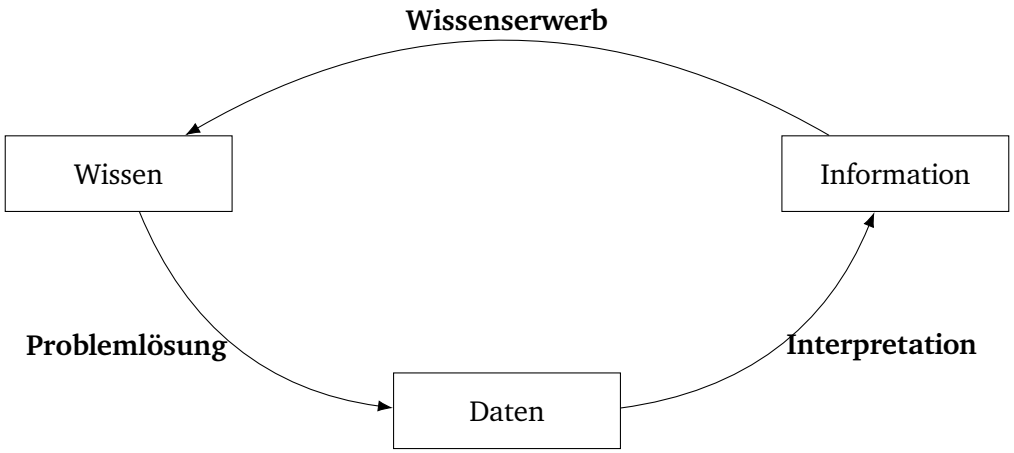
Inhaltsverzeichnis

1 Einführung

1.1 Daten, Informationen und Wissen

- **Daten** sind noch nicht interpretierte Nachrichten, also Zeichenketten, welche nach gewissen Regeln produziert wurden.
- **Informationen** sind interpretierte Daten.
- **Wissen** ist die Gesamtheit der Erkenntnisse, . . . , die aus den Informationen gezogen werden können.
- Daten = Zeichenkette + Syntax
- Information = Daten + Interpretation
- Wissen = Information + Vernetzung

1.1.1 Informationskreislauf



1.1.2 Strukturierungsgrad

- **Strukturierte Daten:** Daten mit gleichartiger Datenstruktur, einem Modell folgend.
- **Semistrukturierte Daten:** Daten ohne festes Modell, die Strukturen implizieren oder andere Modelle enthalten.
- **„Unstrukturierte“ Daten:** Daten ohne formalisierte inhaltliche Struktur.

Unstrukturierte Daten

- Text in natürlicher Sprache gilt als „unstrukturiert“.
- Für Menschen sind Texte oft intuitiv verständlich,
- für Rechner sind die Inhalte nur schwer erfassbar.
- Meistens lassen sich allerdings strukturierte Daten aus Texten ableiten (Text Mining und NLP).

1.2 Informationsmanagement

1.2.1 Informationssystem

Anforderungen:

- Die für die Anwendung nötigen Daten sollen vollständig aus den gespeicherten Daten ableitbar sein (Informationserhalt).
- Die Wiedergewinnung der Daten soll möglichst effizient sein.
- Es sollen nur vernünftige Daten gespeichert werden (je nach Informationsbedarf) und gespeichert werden können (Konsistenz).
- Anwendungsdaten sollten möglichst ohne Redundanzen gespeichert werden.

1.2.2 Typischer Entwurfsprozess

1. **Anforderungsanalyse**
2. **Konzeptioneller Entwurf**
   Abstrakte Modellierung der Domäne, Formale Beschreibung (ER-Modell, UML, . . .)
3. **Verteilungsentwurf**
   Fragmentierung der Daten, Synchronisation und Replikation
4. **Logischer Entwurf**
   Abbildung des konzeptionellen Modells auf die Konzepte des Informationssystems
5. **Datendefinition**
   Deklaration/Programmierung des Modells im Informationssystem
6. **Physischer Entwurf**
   Definition von Zugriffs-/Speicherstrukturen
7. **Implementierung und Wartung**

1.2.3 Datenzugang

Das Speichern von Daten ist sinnlos, wenn diese nicht abgefragt werden können.

- Deklarativer Zugang (Suche), spezifiziert durch Prädikate
- Navigierender Zugang (Blättern), Positionierung und Verfolgung von Pointern

2 Datenbankarchitekturen

2.1 Datenbanksysteme

2.1.1 Ehemalige Systeme, Nachteile und Probleme

- Zu Anfang (um 1960) wurden Daten in Dateien abgelegt, deren Format von Anwendung und Gerät abhängig war.
- Problemfelder
  - Datenredundanz
  - Effiziente Verarbeitung der Daten
    Muss von jeder Anwendung selbst geschaffen werden.
  - Paralleles Arbeiten mit gleichen Daten
  - Fehlende Zugriffskontrolle und Datensicherheit
  - Datenunabhängigkeit
    Nur die Anwendung weiß über das Format der Daten Bescheid, d.h. ohne die Anwendung können die Daten nicht gelesen werden.

2.1.2 Datenintegration und -unabhängigkeit

Datenintegration Die gesamte Anwendungssoftware arbeitet mit den gleichen Daten, der Gesamtbestand der Daten wird *Datenbank* genannt.

- Die Daten werden nur einmal gespeichert → Auflösung der Datenredundanz.
- Abfragesprachen und automatische Optimierung → effizienter Zugriff.
- Transaktionen und einheitliche Schnittstellen → gemeinsame Datennutzung und paralleler Zugriff.
- Benutzerverwaltung und Zugriffskontrolle → Schutz von ungewolltem Zugriff.
- Datensicherheit (z.B. Backups) → Schutz von ungewolltem Verlust.

Datenunabhängigkeit Die Anwendungssoftware ist von der (technischen) Implementierung der Datenbank entkoppelt.

- Gleiche Schnittstelle und gleiche logische Struktur.
- Unabhängig davon, ob die Daten in Tabellen, Bäumen, etc. abgelegt werden.

2.1.3 Abgrenzung: Datenbank vs. Datenbankmanagementsystem vs. Datenbanksystem

Datenbank (DB) Einheitlich beschriebene Darstellung mittels diskreter Daten auf persistenten Speichermedian (z.B. Festplatte, SSD, Hauptspeicher).

Datenbankmanagementsystem (DBMS) Software zur Verwaltung von Datenbanken (ein DBMS, mehrere DBs). Die Daten können nur über das DBMS in die DB eingefügt, gelöscht, geändert, gelesen werden.

Datenbanksystem (DBS) Das Gesamtsystems aus Datenbank und Datenbanksystem.

2.1.4 DBMS

Ermöglichung der einheitlichen Beschreibung (Datenmodellierung) und sicheren Bearbeitungen der Daten.
Typische Anforderungen:

- Verwaltung der persistenten Daten (lange Lebensdauer, Gewährleistung der Konsistenz).
- Effizienter Zugriff auf große Datenmengen (siehe Kapitel 9).
- Optimierbare Anfragesprachen (siehe Kapitel 6).
- Flexibler Mehrbenutzerbetrieb (siehe Kapitel 8).
- Sicherheit vor Systemabsturz und fehlerhaften Transaktionen.
- Feinkörnige Zugriffskontrolle.
- Datenunabhängigkeit.

Beispiele:

- Hierarchisch: IMS (IBM)
- Netzwerkmodell: UDS (Siemens), CODASYL
- Relational:
  - DB2, System R, SQL/DS (IBM)
  - ORACLE
  - Ingres, PostgreSQL
  - Informix
  - MySQL, MariaDB
  - Firebird, Interbase
  - Sybase
  - dBASE, Paradox
  - MS-Access, FileMaker
- Objektorientiert: Poet, ODABA (siehe Kapitel 10)
- Dokumentenbasiert: CouchDB, Elasticsearch, MongoDB (siehe Kapitel 10)

Aufgaben eines DBMS nach Codd

1. *Integration:* einheitliche, nicht redundante Verwaltung der Daten
2. *Operationen* zum Speicher, Suchen, Ändern
3. *Katalog* zur Datenspeicherung
4. *Benutzersichten:* Abbildung des Gesamtbestandes auf die für den Nutzer relevanten Daten
5. *Konsistenzüberwachung:* Korrektheit, Integritätssicherung
6. *Zugriffskontrolle:* Schutz vor unautorisiertem Zugriff
7. *Transaktionen:* Ausführung mehrerer Operationen als Ganzes (atomar)
8. *Synchronisation:* z.B. Vermeidung von Schreibkonflikten
9. *Datensicherung:* z.B. Wiederherstellung bei Systemfehlern

2.1.5 DBS

Vorteile

- Datenintegration
- Datenunabhängigkeit
- Verwaltung der Daten durch DBMS
- Optimierbare Anfragesprachen
- Kompatible Mechanismen für Zugriffskontrolle, Ausfallsicherheit, Mehrbenutzerbetrieb
- Anwenderbezogene Sichten
- ...

Nachteile

- Schwergewichtig, zum Teil zu viele Funktionalitäten
- Effizienzeinbußen durch Verallgemeinerung
- Gegebenenfalls konkurrierende Optimierungsziele mehrerer Anwendungen
- Hohe Kosten: DBMS und zusätzliche Hardware
- Qualifiziertes Personal (DB Admin, Data Analyst, Data Scientist, ...)
- ...

2.2 Architekturmodelle

- Schemaarchitektur
  - Welche Daten werden im DBS wo und wie gespeichert?
  - Welche Datenmodelle und Datenstrukturen werden genutzt?
  - Wie kann Datenunabhängigkeit erreicht werden?
- Systemarchitektur
  - Aus welchen Komponenten besteht ein DBS?
- Anwendungsarchitektur
  - Wie können Benutzer und Anwendungen mit dem DBS arbeiten und Daten verwalten?

2.2.1 Drei-Ebenen-Schemaarchitektur

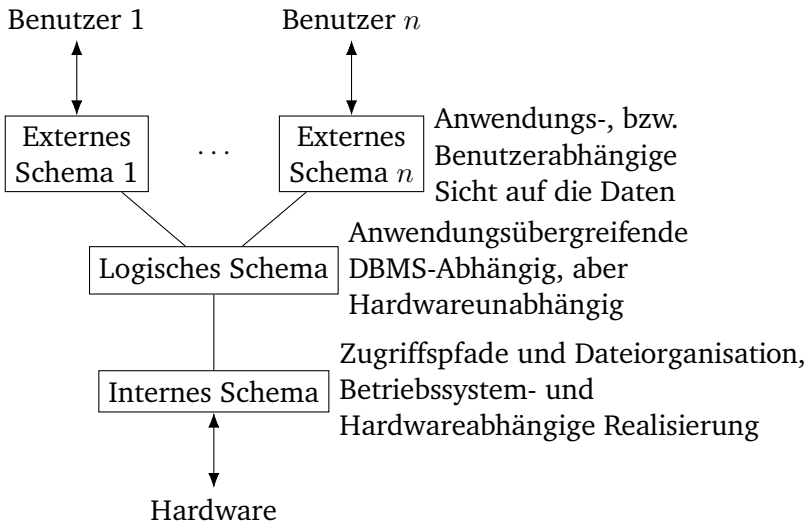


Abbildung 2.1: Drei-Ebenen-Schemaarchitektur

Datenunabhängigkeit

- Die Daten zwischen dem externen Schema, dem logischen Schema und dem internen Schema sind unabhängig, wobei das logische Schema die *logische Datenunabhängigkeit* und das interne Schema die *physische Unabhängigkeit* sichert.
- Physische Datenunabhängigkeit
  - Die Änderungen an Speicherstrukturen und Zugriffspfaden sind für Anwenderprogramme und Anfragen unsichtbar.
- Logische Datenunabhängigkeit
  - Änderungen an der logischen Schicht sind für Anwenderprogramme und Anfragen unsichtbar.
  - Jede Anwendung kann ihre eigene Sicht auf die gemeinsame DB erhalten.

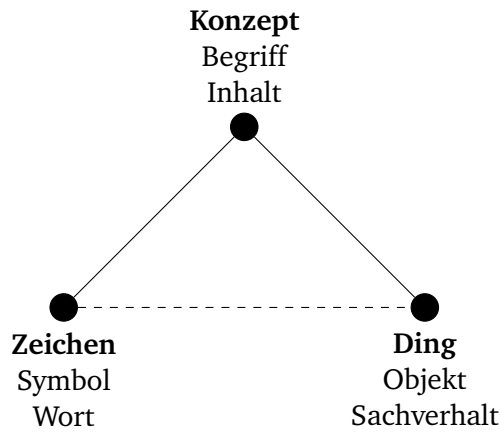
Anfragenbearbeitung und -ergebnis

- Für die *Anfragenbearbeitung* wird die Anfrage immer weiter nach unten durchgereicht, bis auf dem internen Schema das Ergebnis berechnet wird.
- Dieses *Ergebnis*, bzw. die *Ergebnisdarstellung*, wird anschließend nach oben durchgereicht.

3 Konzeptionelle Datenmodellierung

3.1 Wissensrepräsentation

3.1.1 Semiotisches Dreieck



Intension, Merkmale, Extension, Instanz

- Intension: Die zu speichernden gemeinsamen Merkmale, das Schema der Daten.
- Extension: Eine genau Ausprägung der Intension.

3.1.2 Merksmalsrepräsentation

- Informationsbedarf zu konkreten Objekten.
- Ein reales Objekt kann nicht in einer DB gespeichert werden → Stattdessen Speicherung von Merkmalen, welche das Objekt repräsentieren.
- Identifikation: Merkmale des Objektes, welche das Objekt eindeutig bestimmen.

3.1.3 Objektbeschreibung

- Statische Eigenschaften: Merkmale, eindeutige Identifikation
- Dynamische Eigenschaften: Aktionen/Operationen
- Integritätsbedingung: Was definiert das Objekt? Wann ist das Objekt sinnvoll?

3.2 Datenmodellierung

- Ein Datenmodell ist ein System von Konzepten zur Darstellung von Ausschnitten aus der realen Welt.
- Die Modelle dienen der Erfassung und Darstellung der Informationsstruktur.
- Datenmodelle bestehen aus
  - Strukturen (Statische Eigenschaften)
  - Operatoren (Dynamische Eigenschaften)
  - Constraints (Integritätsbedingungen)
- Ein Datenmodell kann auf unterschiedlichen Abstraktionsebenen erstellt werden:
  - Konzeptuell (Implementierungsabhängig), siehe Kapitel 3.
  - Logisch (Implementierungsspezifisch, Geräteunabhängig), siehe Kapitel 4.
  - Physisch (Geräteabhängig, technisch motiviert), siehe Kapitel 8.
  - Benutzer-/Anwendungsbezogen (fachlich motiviert), siehe Kapitel 6.

3.2.1 Konzeptuelle Datenmodellierung

- Kernfragen
  - Welche Objekte sollen in der DB gespeichert werden?
  - Zu welchen Konzepten (Objekttypen) lassen sich die Objekte zusammenfassen?
  - Welche Merkmale sollen gespeichert werden?
  - Welche Regeln müssen gelten?
- Klassische Datenbankmodelle...
  - modellieren nur statische Eigenschaften und
  - nutzen nur minimale Mechanismen zur Beschreibung von Integritätsbedingungen (z.B. Eindeutigkeit).
  - Operatoren (dynamische Eigenschaften) werden meistens allgemein betrachtet und daher nicht im Modell dargestellt:
    * Einfügen
    * Verändern
    * Löschen

3.3 Entitäten-Beziehungsmodell (ERM)

- Das Entity-Relationship-Model (ERM) ist ein einfacher Formalismus zur konzeptuellen Modellierung von Datenbanken.
- Eigenschaften:
  - Fokus auf statische Eigenschaften (Struktur).

  - Keine dynamischen Eigenschaften (Operatoren).
  - Nur minimale Mechanismen zur Beschreibung von Integritätsbedingungen.
- Sehr einfach verständlich.
- Das Standardmodell in frühen Entwurfsphasen.
- Textuell und grafisch darstellbar.

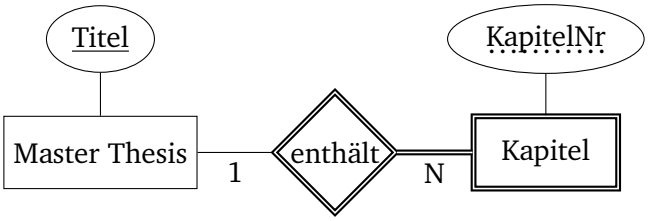
3.3.1 Entitätstypen

- *Entitäten* sind identifizierbare Objekte in der modellierten Welt.
- Ein *Entitätstyp* ist ein gemeinsamer Typ von Objekten mit gleichen Merkmalen.
- Textuelle Notation: $E(\cdots)$
- Grafische Notation: Rechteck
- Beispiel:

$$\text{Konto}(\cdots)$$

Abhängiger/Schwacher Entitätstyp

1. Wird durch eine funktionale Beziehung (`N:1`) identifiziert.
2. Die Existenz hängt von einer anderen, bestimmenden, Entität ab.
3. Abhängige Entitätstypen haben nur einen partiellen Schlüssel.

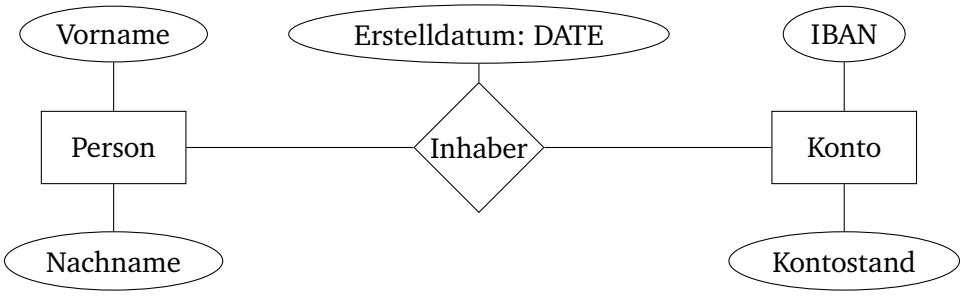


3.3.2 Attribute

- Ein *Attribut* ist eine Eigenschaft eines modellierten Entitäts- oder Beziehungstyps.
- Der *Wert* eines Attributs A kann durch eine Funktion $A : E \to D$ abgebildet werden, wobei D der Wertebereich ist.

- Der *Datentyp* und *Wertebereich* kann frei definiert werden, in vielen Fällen ist die Nutzung von Grundmengen ($\mathbb{N}$, $\mathbb{Q}$, ...) hilfreich.
- Textuelle Notation (Entitätstyp): $E(A_1, \cdots, A_n)$
- Textuelle Notation (Beziehungstyp): $R(E_1, \cdots, E_k; A_1, \cdots, A_n)$
- Optional kann auch die Domäne mit angegeben werden.
- Grafische Notation: Oval an dem Entitätstyp/Beziehungstyp

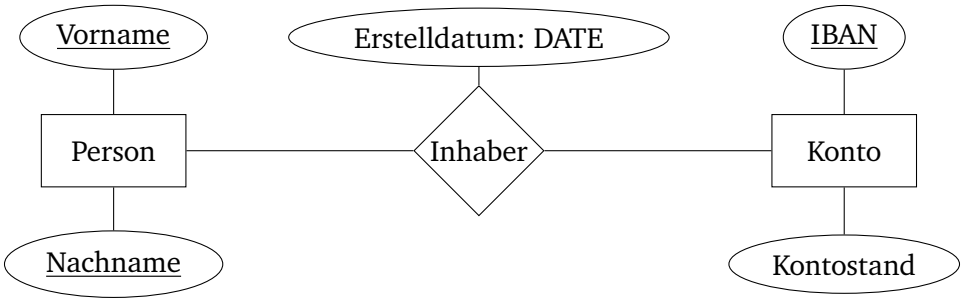
$$\text{Inhaber}(\text{Person}, \text{Konto})$$



Schlüsselattribute

- Die *Schlüsselattribute* sind ein Teilmenge der Attribute, die eine Entität eindeutig identifizieren $\Longrightarrow$ Eindeutigkeitsbedingung.
- Für die Entität $E = (A_1, \cdots, A_n)$ gilt: Schlüsselattribute $\{S_1, \cdots, S_k\} \subseteq \{A_1, \cdots, A_n\}$.
- Beziehungstypen haben keine Schlüsselattribute.
- Textuelle Notation: Unterstrichen
- Grafische Notation: Unterstrichen

$$\text{Inhaber}(\text{Person}, \text{Konto})$$



- Die korrekte Wahl der Schlüsselattribute ist eine Modellierungsentscheidung und benötigt Fachwissen (bspw. gehört eine E-Mail Adresse immer genau einer Person).

Schlüsselattribute: Semantik

Seien $\{S_1, \cdots, S_k\} \subseteq \{A_1, \cdots, A_n\}$ Schlüsselattribute für E, $\sigma(E)$ die Menge aller Entitäten von E und $\sigma(A)(e)$ der Wert, den die Entität e für das Attribut A annimmt.

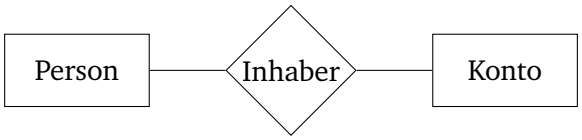
Eindeutigkeitsbedingung

$$\forall e_1, e_2 \in \sigma(E) : (\sigma(S_1)(e_1) = \sigma(S_1)(e_2) \wedge \cdots \wedge \sigma(S_k)(e_1) = \sigma(S_k)(e_2)) \implies e_1 = e_2$$

3.3.3 Beziehungstypen

- Eine *Beziehung* beschreibt ein Rollenverhältnis zwischen zwei oder mehr Entitäten.
- Ein *Beziehungstyp* ist eine Menge von Beziehungen des gleichen Typs.
- Textuelle Notation: $R(E_1, \cdots, E_n)$
- Grafische Notation: Raute

$$\text{Inhaber}(\text{Person}, \text{Konto})$$



Stelligkeit/Grad

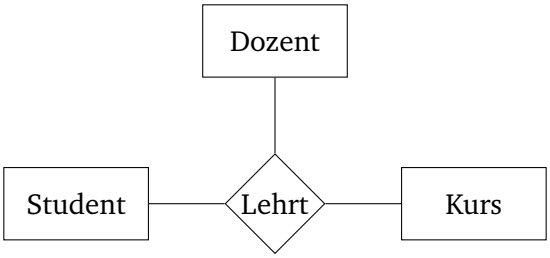
- Die *Stelligkeit* gibt an, wie viele Entitätstypen an einem Beziehungstyp beteiligt sind.
- Am häufigsten sind binäre Beziehungstypen (mit Stelligkeit 2).

Ternäre Beziehungstypen

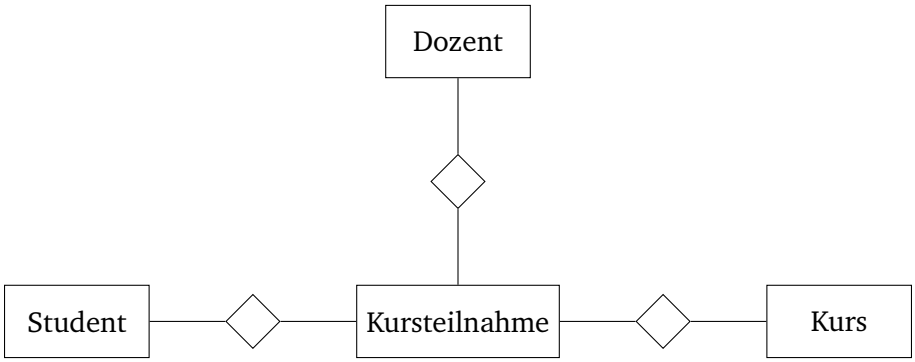
- Eine ternäre Beziehung (mit Stelligkeit 3) kann durch 3 binäre Beziehungen ersetzt werden.

Beispiel

- Die ternäre Beziehungsrelation:



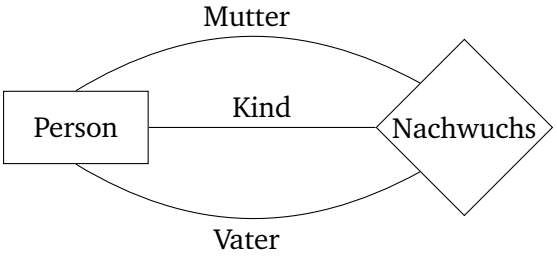
- kann wie folgt durch drei Beziehungen und einer neuen Entität ersetzt werden:



- Damit werden nur noch binäre Relationen genutzt.

Rekursive Beziehungen und Rollennamen

- Ein *rekursiver Beziehungstyp* ist ein Beziehungstyp, an dem ein Entitätstyp mehrfach beteiligt ist.
- Hier ist es nötig, Rollennamen zu vergeben, da die Beziehung sonst nicht korrekt interpretierbar ist.
- Beispiel: ist-Nachwuchs-von$(\text{Mutter} : \text{Person}, \text{Vater} : \text{Person}, \text{Kind} : \text{Person})$



Kardinalitäten

- Die *Kardinalität* einer Relation besagt, wie viele Entitäten jeweils an einer Beziehung beteiligt sind.
- Hierzu gibt es beim ER-Diagramm 2 Grundlegende Notationen: Die Min-Max-Notation und die Chen-Notation.
- Die grundlegenden Beziehungskardinalitäten zwischen zwei Entitäten E1, E1 sind:
  - `1:1`, Ein E1 hat genau ein E2; Ein E2 hat genau ein E1.
  - `1:N`, Ein E1 hat beliebig viele E2; Ein E2 hat genau ein E1.
  - `N:M`, Ein E1 hat beliebig viele E2; Ein E2 hat beliebig viele E1.
- In den folgenden Abbildung wird der Unterschied zwischen Min-Max-Notation und Chen-Notation ersichtlich.

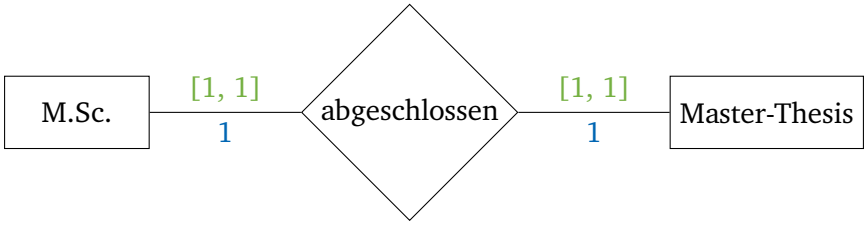


Abbildung 3.1: 1 : 1 Beziehung

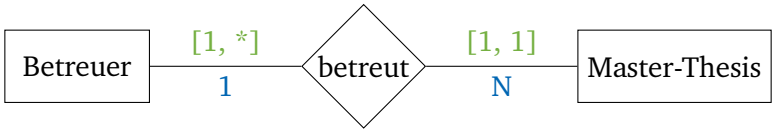


Abbildung 3.2: 1 : N Beziehung

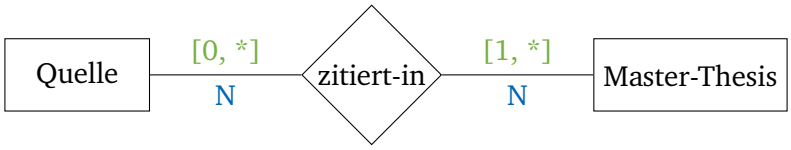


Abbildung 3.3: N : M Beziehung

Funktionale Beziehungen

- Ein funktionaler 1 : N Beziehungstyp $R = E_1 \times E_2$ lässt sich auch als Funktion $R : E_1 \to E_2$ darstellen.
- Im ER-Diagramm wird dies wie in Abb. 3.4 gekennzeichnet (die Kardinalitäten können ausgelassen werden, da sie durch die funktionale Beziehung gegeben sind).
- Beispiel, Textuell: betreut : Master-Thesis $\to$ Betreuer
  betreut(thesis1) = betreuer1, betreut(thesis2) = betreuer1, ...

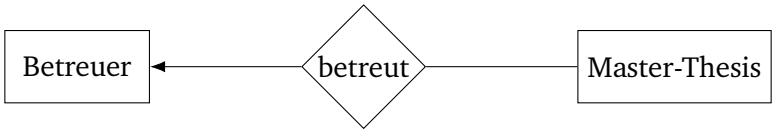
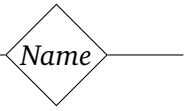


Abbildung 3.4: Funktionale Beziehungen im ER-Modell

3.3.4 Notationsübersicht

Entitätstyp *Name*

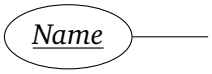
Abhängiger Entitätstyp *Name*

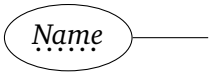
Beziehungstyp 

Bestimmender Beziehungstyp 

Funktionaler Beziehungstyp 

Attribut 

Schlüsselattribut 

Partieller Schlüssel 

Min-Max-Notation `[0, 1]`, `[1, *]`, `[1, 1]`, ...

Chen-Notation `N`, `1`

4 Logische Datenmodellierung

4.1 Relationales Modell

- Die zu modellierende Welt wird ausschließlich durch *endliche Relationen* dargestellt.
  $R_1, \cdots, R_n$
- Diese werden mit *Relationenschemata* beschrieben.
- Die grundlegenden Mengen werden als *Wertebereiche* (oder *Domänen*) bezeichnet.
  $D_1, \cdots, D_n$

4.1.1 Tupel

- Ein Tupel t ist eine Liste von Werten mit bestimmten Datentypen, d.h. $t \in D_1 \times \cdots \times D_n$.
- 2-Tupel werden dabei „Paare“, 3-Tupel „Tripel“ genannt.

4.1.2 Relationenschema

- Ein *Relationenschema* stellt eine Beschreibung von Tupeln dar.
- Diese haben strukturelle Eigenschaften (Attribute, also Namen und Datentypen) und dynamische Eigenschaften (Operationen, z.B. zum Löschen).
- Außerdem können *Integritätsbedingungen* vorliegen, bspw. ein Maximalwert für das Alter.
- **Beispiel:** `(„Paul“, 23, 50.00), („Anna“, 24, 55.00), . . .` mit dem Schema `(Name, Alter, Kontostand)` und den Integritätsbedingungen:
  - Der Name enthält nur Buchstaben und ist maximal 40 Zeichen lang,
  - das Alter ist eine Zahl von 0 bis 150 und
  - der Kontostand ist eine Dezimalzahl größer 0.

4.1.3 Relation

- Eine *Relation* $R \subseteq D_1 \times \cdots D_n$ ist eine Menge von n-Tupeln, deren Elemente bestimmte Eigenschaften erfüllen.
- $n = \deg(R)$ heißt *Stelligkeit* oder *Grad* von R.

- Relationen lassen sich auch als Tabellen darstellen, bspw. die Relation $<_{\text{bis4}} \subseteq \{1,2,3,4\}^2 = \{(1,2),(1,3),(1,4),(2,$ mit $(a,b) \in <_{\text{bis4}}$:

a	b
1	2
1	3
1	4
2	3
2	4
3	4

4.1.4 Notation

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

Relationenschema $\text{Schema}(R) = \{A_1, \cdots, A_n\}$

Attribut Funktion $A_i : R \to D_i$, die jedem Tupel aus R einen Wert aus dem Wertebereich zuordnet.

- Wert einer Spalte A_i des Tupels $t \in R$: $t(A_i) = A_i(t) \in D_i$
- Werte mehrerer Spalten $\alpha \subseteq \text{Schema}(R)$: $t(\alpha) = (\alpha_1(t), \cdots, \alpha_k(t)$

Teiltupel $t(\alpha)$

Tupel $t = t(\text{Schema}(R)) = t(A_1, \cdots, A_n) = (A_1(t), \cdots, A_n(t) \in R$

4.2 Schlüssel

- Schlüsselattribute von Entitätstypen
- Eindeutige Identifikation einer Entität
- **Beispiel:** ISBN $\to$ Buch

4.2.1 Formale Definition

- Definiertheit: Eine Menge von Attributen K heißt *definiert*, wenn keine NULL-Werte enthalten sind.
- Es gilt Primärschüssel $\subseteq$ Schlüsselkandidaten $\subseteq$ Superschlüssel $\subseteq \mathcal{P}(\text{Schema}(R))$.

Superschlüssel

- Eine Attributmenge K heißt *Superschlüssel* gdw. jedes Teiltupel genau ein Tupel identifiziert, d.h. K ist *schlüsseleindeutig:*

$$\forall t_1, t_2 \in R : t_1(K) = t_2(K) \implies t_1 = t_2$$

- Ein Superschlüssel K für eine Relation R heißt:
  - *Einfach*, falls $|K| = 1$.

- *Zusammengesetzt*, falls $|K| > 1$.
- *Trivial*, falls $K = \mathrm{Schema}(K)$.
- *Minimal*, falls $\forall K' \subset K : K'$ ist kein Superschlüssel.

Schlüsselkandidat

- Ein minimaler Superschlüssel ist ein *Schlüsselkandidat*.
- Mit $\mathrm{KEYS}(R)$ wird die Menge aller Schlüsselkandidaten für R bezeichnet.

Primärschlüssel (PK)

- Der *Primärschlüssel* ist ein frei gewählter Schlüsselkandidat für R, also $\mathrm{PK} \in \mathrm{KEYS}(R)$.
- Es darf maximal einen Primärschlüssel für jedes R geben.
- Alle anderen Schlüssel $L \in \mathrm{KEYS}(R)$ mit $K \neq \mathrm{PK}$ heißen *alternative Schlüssel* für R.

4.2.2 (Nicht-) Primattribute

- Sei $K \in \mathrm{KEYS}(R)$. Alle Attribute $A \in R$ heißen *Primattribute* für K.
- Nichtprimattribute (NPA) sind an keinem Schlüsselkandidaten beteiligt, d.h. $N \in \mathrm{Schema}(R)$ heißt nichtprim gdw. $N \notin \bigcup \mathrm{KEYS}(R)$, kurz $N \in \mathrm{NPA}(R)$.

4.2.3 Künstliche Schlüssel

- Ein *künstlicher Schlüssel* (oder auch *Surrogatschlüssel*) wird nicht aus Daten gebildet.
- Häufig wird dieser durch eine fortlaufende Nummer (Sequenz) gebildet.
- **Beispiele:** ISBN, Sozialversicherungsnummer, Steueridentifikationsnummer, Personalausweisnummer, Führerscheinnummer, Reisepassnummer, Kontonummer, IBAN, BIC, Bankleitzahl, Matrikelnummer, Nutzernamen, TU-ID, ...

4.2.4 Schlüsselbeziehungen

- Eine Attributmenge R' einer Relation S heißt *Fremdschlüssel* bezüglich einer Relation R gdw. $\rho(K')$ der PK von R ist und die Bedingung der *referenziellen Integrität* erfüllt ist.
- Die Bedingung der *referenziellen Identität* heißt, dass das Attribut K' in S nur Werte annimmt, für die in R ein entsprechender Wert existiert, d.h.:

$$\forall s \in S : \exists r \in R : s(K') = r(\rho(k'))$$

4.3 Logischer Entwurf

- **Ziel:** Abbildung des konzeptionellen Modells auf ein logisches Datenmodell.

4.3.1 Notation für Relationenschemata

R					} Name des Schemas
	A_1	A_2	$\cdots$	A_n	} Attribute
PK	X		$\cdots$		} Primärschlüssel
FK		S.A	$\cdots$		} Fremdschlüssel

4.3.2 Abbildung von Entitätstypen

- Jeder Entitätstyp $E(A_E; K_E)$ mit Attributen A_E und Schlüsseln K_E wird auf das Relationsschema $R_E(A_E; K_E)$ mit Primärschlüssel $K_E \subseteq A_E$ abgebildet.
- Das Relationsschema jedes von E abhängigen Entitätstyps $F(A_F; K_F)$ mit partiellen Schlüsselattributen K_F wird um die Schlüsselattribute aus E erweitert: $R_F(A_F \cup K_E; K_F \cup K_E)$. Der Schlüssel K_E von E komplettiert somit den partiellen Schlüssel K_F.

4.3.3 Abbildung von Beziehungstypen

- Beziehungen eines ER-Modells können eigene Relationen bilden.
- Ein Beziehungstyp $B(E_1, \cdots, E_n; A_B)$ zwischen Entitätstypen $E_i(A_{E_i}; K_{E_i})$ bildet das Relationsschema: $R_B(A_B \cup K_1 \cup \cdots \cup K_n; K_B)$.
- Die Attributmenge von R_B enthält
  - beziehungseigene Attribute (falls vorhanden) und
  - Schlüsselattribute der beteiligten Entitätstypen als Fremdschlüssel, wobei
  - Namenskonflikte wenn nötig durch Umbenennung gelöst werden.
- Der PK von R_B (K_B) enthält alle Fremdschlüssel aus $K_1 \cup \cdots \cup K_n$.
- Besonderheiten mit $B(E, F)$ als Beziehung zwischen den Entitätstypen $E(A_E; K_E)$ und $F(A_F; K_F)$:
  - Bei einer `M:N`-Beziehung ist $K_E \cup K_F$ der PK der Beziehung.
  - Bei einer `1:N`-Beziehung ist K_F der PK der Beziehung.
  - Bei einer `1:1`-Beziehung ist kann K_E oder K_F als PK der Beziehung gewählt werden.
- Bei Beziehungen $B(E, F; \beta)$ mit Kardinalität $1:1$ oder $1:N$ wird die eigene Relation häufig weg gelassen, sondern mittels Fremdschlüsseln in R_F abgebildet:
  - $R_E(A_E; K_E)$
  - $R_F(A_F \cup K_E \cup \beta; K_F)$ (Fremdschlüssel bzgl. R_E und beziehungseigene Attribute β kommen hinzu.)
  - Die Schlüssel von R_E und R_F bleiben gleich, es gibt keine eigene Relation für B.

4.4 Datenbanknormalisierung

4.4.1 Redundanz und Anomalien

- Redundanz ist Speicherplatzverschwendung.

- Redundanz führt meistens zu Inkonsistenzen, wenn Daten editiert werden $\implies$ Anomalien.

Update-Anomalie

Bei einem Update der Daten müssen alle Tupel mit den Daten aktualisiert werden. Dies führt zu *Update-Anomalien*.

Einfüge-Anomalie

Sind bei dem Einfügen von Daten noch nicht alle erforderlichen Daten verfügbar, kommt es zu *Einfüge-Anomalien*.

Lösch-Anomalie

Beim Löschen einzelner Daten kann es passieren, dass auch andere Daten verschwinden. Es kommt zu *Lösch-Anomalien*.

4.4.2 Funktionale Abhängigkeit (FD)

Sei $\mathrm{Schema}(R) = \{A_1, \cdots, A_n\}$ ein Relationenschema mit Attributteilmengen α, β. Die Menge β ist *funktional abhängig* gdw. für jeden Wert aus α genau ein Wert für β existiert, d.h. α bestimmt β.

- **Schreibweise:** $R.\alpha \to R.\beta$, kurz $\alpha \to \beta$
- **Formal:** $\forall t_1, t_2 \in R : t_1(\alpha) = t_2(\alpha) \implies t_1(\beta) = t_2(\beta)$
- Für alle Schlüssel K von R gilt immer $K \to \mathrm{Schema}(R)$.

Semantik

Die funktionalen Abhängigkeiten definieren die *Semantik* einer Datenbank.

- Andere Interpretationen führen oft zu anderen Schemata.
- Die Normalisierung ist abhängig von den FDs.

FD-Hülle

- Die FDs beeinflussen die Relationenschemata über den Normalisierungsprozess.
- Funktionale Abhängigkeiten müssen in ihrer Gesamtheit betrachtet werden, inklusive den nicht aufgelisteten FDs.
- Die FD-Hülle F^+ ist die Menge aller FDs, die aus der bekannten FD-Menge F impliziert werden.

Attributhülle

- Analog zur FD-Hülle kann die Attributhülle A^+ definiert werden.
- Die Attributhülle A^+ enthält alle Attribute, die von A funktional abhängig sind.

Armstrong-Axiome

Sei F eine FD-Menge und $\alpha, \beta, \gamma, \delta$ Attributmengen.

1. **Reflexivität** $\beta \subseteq \alpha \implies \alpha \to \beta$
2. **Erweiterung** $\alpha \to \beta \implies \alpha\delta \to \beta\delta$
3. **Transitivität** $\alpha \to \beta \wedge \beta \to \delta \implies \alpha \to \delta$

Da Armstrongs Axiome korrekt und vollständig sind, erlauben sie das Ermitteln der Hülle F^+ aller implizierten FDs.

Außerdem kann aus den Axiomen abgeleitet werden, dass gilt:

- **Additivität** $\alpha \to \beta \wedge \alpha \to \delta \implies \alpha \to \beta\delta$
- **Projektivität** $\alpha \to \beta\delta \implies \alpha \to \beta \wedge \alpha \to \delta$
- **Pseudotransitivität** $\alpha \to \beta \wedge \beta\delta \to \gamma \implies \alpha\delta \to \gamma$

Volle/Partielle Funktionale Abhängigkeiten

- Volle funktionale Abhängigkeit
  - β heißt *voll funktional abhängig* von α gdw. $\alpha \to \beta$ gilt und kein $\alpha' \subset \alpha, \alpha' \neq \emptyset$ existiert mit $\alpha' \to \beta$.
- Partielle Funktionale Abhängigkeit
  - β heißt *partiell funktional abhängig* von α gdw. β nicht voll funktional abhängig ist von α.

Transitive Abhängigkeit

Eine Attributmenge β heißt *transitiv abhängig* von α gdw. $\alpha \to \gamma$ und $\gamma \to \beta$ voll funktional abhängig sind, aber $\gamma \to \alpha$ keine voll funktionale Abhängigkeit darstellt (Schreibweise: $\alpha \to \gamma \to \beta$).

4.4.3 Normalformen und Normalisierung

- Anforderungen an den Datenbankentwurf:
  - Möglichst redundanzfreie Speicherung.
  - Nur semantisch sinnvolle und konsistente Daten.
  - Alle benötigten Daten müssen aus den Basisrelationen herleitbar sein (verlustfrei).
- **Normalisierung**
  - Regeln zur Transformation von Schemata, um Anomalien zu vermeiden und den Entwurf zu verbessern.

> „The key, the whole key and nothing but the key. So help me Codd!“

Erste Normalform

Eine Relation ist in 1NF, wenn *alle Attributwerte atomar* sind.

Flapsig: Attribute sind automar, alles hängt vom Schlüssel ab.

Zweite Normalform

Eine Relation ist in 2NF, wenn sie in 1NF ist und *alle Nichtprimtattribute vollständig* von *jedem Schlüsselkandidaten* abhängen.

$$R \text{ in 1NF} \land \forall A \in \text{NPA}(R) : \forall K \in \text{KEYS}(R) : K \to A \text{ ist volle FD}$$

Flapsig: NPA hängen vom gesamte Schlüssel ab.

Dritte Normalform

Eine Relation ist in 3NF, wenn sie in 2NF ist und *alle Nichtprimattribute nicht transitiv* von *einem Schlüsselkandidaten* abhängen.

$$R \text{ in 2NF} \land \forall A \in \text{NPA}(R) : \forall K \in \text{KEYS}(R) : \forall B \in \text{Schema}(R) : \neg(K \to B \to A \text{ ist transitive FD})$$

Flapsig: NPA hängen nur vom Schlüssel ab, keine transitiven FDs.

Boyce-Codd Normalform

Eine Relation ist BCNF, wenn sie in 2NF ist und *alle Attribute nicht transitiv* von *einem Schlüsselkandidaten* abhängen.

$$R \text{ in 2NF} \land \forall A \in \text{Schema}(R) : \forall K \in \text{KEYS}(R) : \forall B \in \text{Schema}(R) : \neg(K \to B \to A \text{ ist transitive FD})$$

Flapsig: Attribute hängen nur vom Schlüssel ab, keine transitiven FDs.

4.5 Syntheseverfahren

- Bei der *Synthese* werden die Redundanzen in Relationen durch algorithmische Äquivalenzumformungen einer gegebenen Attribut- und FD-Menge entfernt.
- Einfache Syntheseverfahren sorgen dabei dafür, dass nur so viele Aufteilungen wie nötig gemacht werden, polynomiale Laufzeit erreicht wird und die Abhängigkeitstreue erhalten bleibt (es dürfen keine FDs verloren gehen).

- Vollständige Syntheseverfahren garantieren, dass alle Relationen in 3NF sind.
- Vereinfachte Syntheseschritte:
  1. Linksreduktion.
     Entfernung überflüssiger Attribute aus der linken Seite der FDs.
  2. Rechtsreduktion.
     Entfernung überflüssiger FDs.
  3. FDs mit identischer linker Seite gruppieren.
  4. Gruppen mit äquivalenter linker Seite zusammenfassen.
     Äquivalente linke Seite heißt hier, dass die Hülle gleich ist.
  5. Relationen bilden.
     Jede Gruppe bildet eine Relation, jede linke Seite einer Gruppe bildet einen Schlüsselkandidaten der Relation.

5 Logische Abfragemodelle

Sei $R = (A_1, \cdots, A_n; \{K_1, \cdots, K_m\})$ ein Relationenschema mit Attributen $A_1, \cdots, A_n$ und Schlüsselkandidaten $K_1, \cdots, K_m$.

- Um mit den Relationen arbeiten zu können sind Operationen nötig, welche die Daten verändern und lesen können.
- Idee: Eine Operation produziert aus bestehenden Relationen neuen Relationen, wodurch das Modell in sich geschlossen bleibt.
- Die drei wesentlichen Ansätze sind die *Relationenalgebra*, das *Relationentupelkalkül* und das *Relationenwertebereichkalkül*, welche im folgenden behandelt werden.
- Dabei haben alle Ansätze die gleiche Aussagekraft.

5.1 Relationenalgebra (RA)

- **Algebraische Struktur:** Menge mit Operationen, welche auf dieser Menge definiert sind.
- Relationenalgebra RA $= (\mathcal{R}, \cup, -, \times, \rho_{a,b}, \pi_\alpha, \sigma_F)$:

$$\begin{aligned}
\mathcal{R} &\text{ Menge aller Relationen, seien } R, S \in \mathcal{R} \\
\cup : \mathcal{R} \times \mathcal{R} &\to \mathcal{R} \text{ Vereinigungsmenge} \\
- : \mathcal{R} \times \mathcal{R} &\to \mathcal{R} \text{ Mengendifferenz} \\
\times : \mathcal{R} \times \mathcal{R} &\to \mathcal{R} \text{ Kartesisches Produkt} \\
\rho_{a,b} : \mathcal{R} &\to \mathcal{R} \text{ Spaltenumbenennung} \\
\pi_\alpha : \mathcal{R} &\to \mathcal{R} \text{ Projektion} \\
\sigma_F : \mathcal{R} &\to \mathcal{R} \text{ Selektion}
\end{aligned}$$

  Alle Operationen sind dabei unter $\mathcal{R}$ abgeschlossen.

5.1.1 Grundlegende Operatoren

Vereinigung

Die Vereinigung $R \cup S$ ist die Menge aller Tupel, die in R oder S enthalten sind.

Dabei müssen die Relationen Vereinigungsverträglich sein, das heißt die gleiche Anzahl und Art an Spalten haben.

Mengendifferenz

Die Mengendifferenz $R - S$ ist die Menge aller Tupel, die in R aber nicht in S enthalten sind.
Dabei müssen die Relationen Vereinigungsverträglich sein.

Kartesisches Produkt

Das kartesische Produkt $R \times S$ ist die Menge aller möglichen $\deg(R) + \deg(S)$-Tupel mit $\deg(R)$ Elementen aus R und $\deg(S)$ Elementen aus S.

Spaltenumbenennung

Sei $a \in \text{Schema}(R)$ ein Attribut von R mit $a : R \to D_a$. Die Spaltenumbenennung $\rho_{a,b}(R)$ ersetzt dann die Spalte a durch eine Spalte $b : R \to D_a$ mit anderem Namen aber gleicher Definition und Werten, d.h. $\forall r \in R : a(r) = b(r)$.

Projektion

Sei $\alpha \subseteq \text{Schema}(R)$ eine Attributmenge von R. Die Projektion $\pi_\alpha(R)$ reduziert alle Tupel in R auf die Attribute α. Es entsteht somit eine Relation aus $|\alpha|$-Tupeln.
Da innerhalb der Relationenalgebra das Ergebnis eine Menge ist, werden Duplikate entfernt.

Selektion (Restriktion)

Sei F eine Formel (Selektionsbedingung). Die Selektion $\sigma_F(R)$ wählt dann alle Tupel aus R aus, für welche die Formel F erfüllt ist.
Eine Formel F besteht aus:

- Konstantenselektionen: Attribut $\circ$ Konstante mit $\circ \in \{=, <, >, \leq, \geq\}$
- Attributselektionen: Attribut $\circ$ Attribut mit $\circ \in \{=, <, >, \leq, \geq\}$
- Logischen Verknüpfungen: $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$

5.1.2 Abgeleitete Operatoren

- Die sechs Basisoperatoren $\cup, -, \times, \rho, \pi, \sigma$ reichen aus, um alle anderen Operatoren abzuleiten.
- Es werden dennoch weitere Operatoren eingeführt, welche den Umgang mit der Relationenalgebra erleichtern.

Schnittmenge

Die Schnittmenge $R \cap S$ ist die Menge aller Tupel, die in R uns S enthalten sind.

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

Dabei müssen die Relationen Vereinigungsverträglich sein, das heißt die gleiche Anzahl und Art an Spalten haben.

Quotient

Seien $\alpha = \text{Schema}(R)$ und $\beta = \text{Schema}(R)$. Der Quotient $R\boldsymbol{\nabla} \cdot S$ ist die Menge aller Tupel, deren Attribute nur in R existieren und für die alle Kombinationen mit Tupeln aus S vorkommen.

$$R\boldsymbol{\nabla} \cdot S = \pi_{\alpha-\beta}(R) - \pi_{\alpha-\beta}((\pi_{\alpha-\beta}(R) \times S) - R)$$

Beispiel

$$\begin{bmatrix} A & B & C & D \\ \hline a & b & c & d \\ a & b & e & f \\ b & c & e & f \\ e & d & c & d \\ e & d & e & f \\ a & b & d & e \end{bmatrix} \boldsymbol{\nabla} \cdot \begin{bmatrix} C & D \\ \hline c & d \\ e & f \end{bmatrix} = \begin{bmatrix} A & B \\ \hline a & b \\ e & d \end{bmatrix}$$

Verbund (Join)

Ein Verbund (Join) ist die wichtigste nicht-elementare Operation, bestehend aus kartesischem Produkt und Selektion:

$$R \bowtie S = \sigma_F(R \times S)$$

Theta-Join Die allgemeine Form eines Joins ist dabei ein Θ-*Join* mit einem arithmetischen Vergleichsoperator $\Theta \in \{=, <, >, \leq, \geq\}$ mit Attributen $a \in \text{Schema}(R)$ und $b \in \text{Schema}(S)$.

Notation

$$R \bowtie_{a\Theta b} S = \sigma_{a\Theta b}(R \times S)$$

Equijoin Gilt $\Theta = =$, so wird dies *Equijoin* genannt.

Notation

$$R \bowtie_{a=b} S = \pi_{a=b}(R \times S)$$

Natural Join Ein *Natural Join* Equijoin für alle Attribute mit gleichem Namen.

Notation

$$R \bowtie S$$

Semijoin Ein Semijoin entspricht einem Natural Join, wobei nur die Attribute der ersten Relation erhalten bleiben.

Notation

$$R \ltimes S = \pi_\alpha(R \bowtie S) = R \bowtie \pi_{\alpha\cup\beta}(S), \quad \alpha = \text{Schema}(R), \beta = \text{Schema}(S)$$

Entarteter Join Ein *Entarteter Join* entspricht dem kartesischen Produkt, wobei alle Spalten mit doppeltem Namen im Vorhinein umbenannt wurden, sodass ein echtes kartesisches Produkt entsteht.

5.1.3 Select-Project-Join-Queries (SPJ-Queries)

- Joins sind äußert wichtig.
- Wurden Relationen aus Entwurfsgründen zerlegt, können diese während der Abfrage wieder kombiniert werden.
- Die meistgenutzten Operationen sind Selektion (S), Projektion (P) und Join (J).
- Sogenannten *SPJ-Queries* enthalten nur Operationen dieser Typen.
- Relationale DBMS optimieren hauptsächlich SPJ-Queries.

5.2 Relationenkalküle

Im folgenden wird das Relationentupelkalkül (RTK) und das Relationenwertebereichkalkül (RWK) betrachtet, welche Kalküle darstellen zur formalen Bau von Abfragen. Kalküle, also formale logische Sprachen, bestehen aus:

- **Bausteinen** (z.B. $\vee$, $\neg$, $\leq$, $\dots$), welche zu
- **wohlgeformten Ausdrücken** zusammengesetzt werden.
- **Ableitungs-/Schlussregeln**, um Ausdrücke in abgeleitete Formen zu transformieren (z.B. $(\emptyset \models \neg a) \implies (\models \neg a)$).
- **Axiome**, welche gegebene, nicht ableitbare, Ausdrücke darstellen.

5.3 Relationentupelkalkül (RTK)

Die allgemeine Form $\{t \mid \Psi(t)\}$ mit der Tupelvariablen t und einer Formel Ψ besteht aus:

- Atomen
  - Tupelvariable $r \in R$ aus einer Relation R.
  - Prädikate der Form $t(\alpha)\Theta r(\beta)$ mit $\Theta \in \{=, <, >, \leq, \geq, \cdots\}$.
  - Prädikate der Form $t(\alpha)\Theta c$ mit einer Konstanten c.
- Formeln
  - Atome sind Formeln.
  - $\neg\Psi$, $\Psi \vee \Phi$, $\Psi \wedge \Phi$ sind Formeln (mit Formeln Ψ, Φ).
  - $\forall r \in R(\Psi(r))$ und $\exists r \in R(\Psi(r))$ sind Formen (mit Formel Ψ).
    - $\forall$ heißt Allquantor.
    - $\exists$ heißt Existenzquantor.

5.4 Relationenwertebereichkalkül (RWK)

Im Relationenwertebereichkalkül (RWK) haben Ausdrücke die allgemeine Form $\{x_1 \cdots x_k \mid \Psi(x_1 \cdots x_k)\}$ und sind sehr ähnlich zu Ausdrücken im RTK, wobei das RWK spalten- und das RTK zeilenorientiert arbeitet. Die Formel Ψ besteht aus:

- Atomen
  - $R(x_1 \cdots x_k)$ mit Relation R und Wertebereichsvariable oder Konstante x_i.
  - Verhältnis $x \Theta y$ zwischen zwei Wertebereichsvariablen.
  - Verhältnis $x \Theta c$ zwischen einer Wertebereichsvariablen und einer Konstante.
- Formeln
  Analog zum RTK.

5.5 Relationenalgebra vs. Relationentupelkalkül

- **Relationenalgebra**
  Prozedurale Sprache; Es wird beschrieben, was getan wird.
- **Relationentupelkalkül**
  Deskriptive Sprache; Es wird beschrieben, wie das Ergebnis aussehen soll.
- Trotz der Unterschiede sind RA und RTK gleich stark in ihrer Aussagekraft (zu jedem RA-Ausdruck gibt es einen RTK-Ausdruck und anders herum).

5.6 RA, RTK, RWK vs. Anfragesprachen

- **SQL** ist eine Mischung aus RA und RTK.
- **QUEL** ist rein RTK-basierte.
- **QBE** ist rein RWK-basierte.

5.6.1 Mächtigkeit

Bisher können viele Anfragen nicht von RA und RTK abgedeckt werden, beispielsweise:

- Arithmetische Operationen
- Aggregatfunktionen
- Boolesche Anfragen
- Rekursive Anfragen
- Sortieroperationen
- Umgang mit Duplikaten

5.7 Änderungsoperationne

- Zustandsfolge der Datenbank $\sigma = (\mathrm{DB}_0, \cdots, \mathrm{DB}_\tau, \cdots)$
- Eine *Änderungsoperation* überführt die Datenbank vom Zustand DB_τ in einen Zustand $\mathrm{DB}_{\tau+1}$.
- Theoretisch muss jeder Zustand vollständig neu aufgebaut werden.
- In der Praxis ist dies zu Aufwendig, weshalb Änderungsoperationen $\mathrm{DB}_{\tau+1} = u(\mathrm{DB}_\tau)$ eingeführt werden.

5.7.1 Einfügen

Sei $\mathrm{DB}_\tau = (R_1, \cdots, R_i, \cdots, R_n)$ eine Datenbank zum Zeitpunkt τ mit den Relationen $R_1, \cdots, R_n$.
Die Grundoperation zum Einfügen eines Tupels t ist die Vereinigung $R_i \cup \{t\}$.

$$\mathrm{DB}_{\tau+1} = \begin{cases} (R_1, \cdots, R_i \cup \{t\}, \cdots, R_n) & \text{falls konsistent} \\ \mathrm{DB}_\tau & \text{sonst} \end{cases}$$

5.7.2 Löschen

Sei $\mathrm{DB}_\tau = (R_1, \cdots, R_i, \cdots, R_n)$ eine Datenbank zum Zeitpunkt τ mit den Relationen $R_1, \cdots, R_n$.
Die Grundoperation zum Löschen eines Tupels t ist die Differenz $R_i - \{t\}$.

$$\mathrm{DB}_{\tau+1} = \begin{cases} (R_1, \cdots, R_i - \{t\}, \cdots, R_n) & \text{falls konsistent} \\ \mathrm{DB}_\tau & \text{sonst} \end{cases}$$

5.7.3 Verändern

Sei $\mathrm{DB}_\tau = (R_1, \cdots, R_i, \cdots, R_n)$ eine Datenbank zum Zeitpunkt τ mit den Relationen $R_1, \cdots, R_n$.
Die Grundoperation zum Löschen eines Tupels $t \to t'$ ist das Löschen und Einfügen im gleichen Schritt $(R_i - \{t\}) \cup \{t'\}$

$$\mathrm{DB}_{\tau+1} = \begin{cases} (R_1, \cdots, (R_i - \{t\}) \cup \{t'\}, \cdots, R_n) & \text{falls konsistent} \\ \mathrm{DB}_\tau & \text{sonst} \end{cases}$$

6 Anfragesprachen

6.1 Structured Query Language (SQL)

- Anfragesprache für rationale Systeme.
- Standardisiert durch ANSI und ISO in unterschiedlichen Revisionen.
- SQL wird von fast alles freien und kommerziellen RDBMS unterstützt.

6.1.1 Datentypen

Datentyp	Bedeutung
`SMALLINT`	Ganzzahl mit kleinem Wertebereich
`INT` oder `INTEGER`	Ganzzahl mit „normalem“ Wertebereich
`REAL`	Gleitkommazahlen mit einfacher Genauigkeit
`DOUBLE PRECISION`	Gleitkommazahlen mit doppelter Genauigkeit
`FLOAT(p)`	Gleitkommazahlen mit Genauigkeit p
`DECIMAL(p, q)` oder `DEC(p, q)` oder `NUMERIC(p, q)`	Festkommazahlen mit Genauigkeit p und Nachkommastellen q
`CHARACTER(n)` oder `CHAR(n)`	Zeichenketter mit fester Länge n
`CHARACTER VARYING(n)` oder `VARCHAR(n)`	Zeichenkette variabler Länge (max n)
`BOOLEAN`	Wahrheitswert (Wahr, Falsch)
`BIT(n)`	Bitfolge fester Länge n
`BIT VARYING(n)`	Bitfolge variabler Länge (max n)
`DATE`	Datum
`TIME`	Uhrzeit
`TIMESTAMP`	Zeitstempel (Datum und Uhrzeit)
`INTERVAL`	Zeitintervall

Tabelle 6.1: Vordefinierte Datentypen in SQL

6.1.2 Wertebereiche

Eigene Wertebereiche lassen sich mittels `CREATE DOMAIN` anlegen.

Die Wertebereiche können statt den vordefinierten Datentypen in Spaltendefinitionen verwendet werden.

```
1 CREATE DOMAIN money AS NUMERIC(7, 2);
2 CREATE DOMAIN shirt_size AS CHAR(2)
3         DEFAULT 'M'
4         CHECK (VALUE IN ('S', 'M', 'L', 'XL'));
```

6.1.3 NULL

Wird NULL als Wert in einem Feld gesetzt, kann dies verschiedene Bedeutungen haben:

- Attribut nicht zutreffend
- Wert unbekannt
- Wert existiert nicht
- Wert ist nicht definiert
- Wert ist ungültig
- Wert wurde nicht angegeben
- Eingefügte Werte bei Verbundoperation

NULL-Werte haben folgende Eigenschaften:

- Mit NULL kann (meist) nicht gerechnet werden.
- Vergleiche mit NULL ergeben immer FALSE.
- Attribute, die mit NOT NULL gekennzeichnet sind, können keine NULL-Werte zugewiesen werden.
- Attribute des PRIMARY KEY sind immer NOT NULL.
- Anstelle von NULL-Werten können auch DEFAULT-Werte definiert werden.

6.2 Datendefinitionssprache (DDL)

6.2.1 Relationen vs. Tabellen

- In der Relationenalgebra sind Relationen Mengen ohne Duplikate.
- In SQL sind Relationen Tabellen und können Duplikate enthalten.

6.2.2 Tabellen Erstellen

```
CREATE TABLE tabellenname (
    spaltenname_1 wertebereich_1 [DEFAULT wert] [NOT NULL],
        ...
    spaltenname_n wertebereich_n [DEFAULT wert] [NOT NULL]
    [, primary-key-definition ]
    [, primary-key-definition_1 ]
        ...
    [, primary-key-definition_k ]
);
```

6.2.3 Tabellen Löschen

```
DROP TABLE tabellenname [CASCADE | RESTRICT];
```

- Löscht die eigentliche Tabelle mit allen enthaltenen Daten.
- Löscht das dazugehörige Relationsschema.
- Löscht die angelegten Indexstrukturen.
- Löschverhalten:

  Cascade Abhängige Sichten und Integritätsbedingungen löschen.

  Restrict Verweigerung des Löschens, falls abhängige Sichten oder Integritätsbedingungen existieren.

6.2.4 Tabellen Ändern

```
ALTER TABLE tabellenname
        ADD [COLUMN] spaltendefinition
      | ALTER [COLUMN] spaltenname spaltenänderung
      | DROP [COLUMN] spaltenname [CASCADE | RESTRICT]
      | ADD CONSTRAINT ib_definition
      | DROP CONSTRAINT ib_name [CASCADE | RESTRICT]
```

6.2.5 Integritätsbedingungen

- Sichere Annahmen über Daten in der Datenbank.
- Integritätsbedingungen stellen sicher, dass die Datenbank konsistent ist.
- Integritätsbedingungen können sich beziehen auf:
  - einzelne Spalten (Spaltenbedingungen),
  - einzelne Tabellen (Tabellenbedingungen),
  - Datenbankschema (tabellenübergreifende Bedingungen) und
  - Wertebereiche
- Modellinhärente Integritätsbedingungen
  - `NOT NULL` bei bestimmten Attributen
  - `UNIQUE` - Eindeutigkeit, insbesondere von Schlüsseln
  - `PRIMARY KEY` und `FOREIGN KEY` - referentielle Integrität
- Anwendungsbezogene Integritätsbedingungen
  - `CHECK` - Überprüfung von Bedingungen auf einer Tabelle (z.B. `CHECK (Alter >= 0)`)
  - `ASSERTIONS` - Tabellenübergreifende Integritätsbedingungen (z.B. Kontostand muss der Summe der Buchungen entsprechen)

Eindeutigkeit (`UNIQUE`)

- Forderung der Eindeutigkeit über die gesamte Tabelle.

  $\approx$ Superschlüssel

```
ALTER TABLE buecher ADD CONSTRAINT UNIQUE(autor, titel, auflage);
```

Primärschlüssel

- Gewählter Schlüsselkandidat (minimal und eindeutig) $\rightarrow$ Primärschlüssel.
- Der Primärschlüssel ist immer `NOT NULL` und `UNIQUE`.
- Ein Primärschlüssel kann auch aus mehreren Attributen zusammengesetzt sein, diese werden dann mit Kommas getrennt.

```
CREATE TABLE buecher ( [···],
    PRIMARY KEY (isbn)
);

ALTER TABLE buecher ADD CONSTRAINT PRIMARY KEY (isbn);
```

Fremdschlüssel

- Die Fremdschlüssel (Foreign Keys) einer Tabelle ist eine Attributmenge, welche mit referentieller Integrität auf eine andere Tabelle verweist.
- Wie beim Primärschlüssel kann auch ein Fremdschlüssel zusammengesetzt sein.

```
[···]
FOREIGN KEY (referenzierendes_attribute)
    REFERENCES referenzierte_tabelle [(referenzierte_attribute)]
    [MATCH FULL | MATCH PARTIAL]
    [referenzielle_aktionen]
[···]
```

Mit folgenden referentiellen Aktionen (Triggern):

```
ON UPDATE | ON DELETE
CASCADE | SET DEFAULT | SET NULL | NO ACTION
```

```
CREATE TABLE buecher ( [···],
    FOREIGN KEY (autorId) REFERENCES autor(id)
);
```

Referentielle Integrität Fremdschlüssel (FK, K wie Schlüssel) verweisen auf die Attributmenge α einer anderen Tabelle. Es gilt referentielle Integrität:

- Die Attributmenge α der Zieltabelle muss `PRIMARY KEY` oder `UNIQUE` sein.
- Neue Datensätze können nur eingefügt werden, wenn die Werte des FK in der referenzierten Tabelle als Wert vorkommen.
- Änderungen oder Löschungen sind nur genau dann möglich, wenn keine abhängigen Datensätze existieren.

Bei `NULL` hängt „vorkommen“ oder „verweisen“ davon ab, ob `MATCH FULL` oder `MATCH PARTIAL` übergeben wurde:

- `MATCH NONE` (Standard): `NULL` im FK heißt „Zeile wird nicht geprüft“.
- `MATCH PARTIAL`: Ein Teil des FK kann `NULL` sein, der Rest wird geprüft.
- `MATCH FULL`: FK muss entweder ganz `NULL` oder ganz non-`NULL` sein.

Referentielle Aktionen (Trigger)

- `CASCADE`
  - Ändert sich ein referenziertes Attribut, so wird auch das referenzierende Attribut angepasst.
  - Bei Löschung wird die referenzierende Zeile ebenfalls gelöscht!
- `SET NULL` bzw. `SET DEFAULT`
  - Setzt den FK auf `NULL` oder einen Standardwert, wenn der bestehende Wert nicht länger in der referenzierten Tabelle existiert.
- `NO ACTION` (Standard)
  - Lässt eine Änderung nicht zu $\rightarrow$ Fehlermeldung

6.2.6 Tabellentypen

- Persistente Basistabellen
  - Persistente, also dauerhafte, sitzungsübergreifende Tabellendefinition und Ausprägung.
  - Definition mittels `CREATE TABLE`.
- Temporäre Basistabellen
  - Die Ausprägung der Tabelle besteht nichtsitzungsübergreifend und nicht persistent.
  - Gedacht für kurzzeitig benötigte Daten, z.B. Berechnungsergebnisse.
  - Daten werden bei Sitzungsende automatisch gelöscht, kein `DROP` nötig.
  - Verschiedene Untertypen.

6.3 Datenabfragesprachen (DQL)

- Abfrage der in der Datenbank gespeicherten Daten.
- Implementiert RA/RTK/RWK und weitere Operationen.

6.3.1 SELECT

```
SELECT { * | spalte_1, [···] spalte_n }
FROM   tabelle_1, [···] tabelle_n
[WHERE bedingung];
```

- RA-Ausdruck: $\pi_{spalte_1,\cdots,spalte_n}(\sigma_{bedingung}(tabelle_1 \times \cdots \times tabelle_n))$

6.3.2 Projektion

```
SELECT attributsliste
FROM   tabelle;
```

- RA-Ausdruck: $\pi_{attributsliste}(tabelle)$

6.3.3 Selektion

```
SELECT *
FROM   tabelle
WHERE  bedingung;
```

- RA-Ausdruck: $\sigma_{tabelle}(tabelle)$

6.3.4 Selektionsbedingung

- Analog zu RA/RTK:
  - **Konstantenselektion:** $attribut \circ konstante$
  - **Attributselektion:** $attribut_1 \circ attribut_2$
  - **Logische Verknüpfungen:** f_1 `AND` f_2, f_1 `OR` f_2, `NOT` f_1
  - Mit Vergleichsoperation $\circ$, abhängig vom Wertebereich:
    - Beispiel Ganzzahlen: $\circ \in \{$`=`, `<`, `>`, `<=`, `>=`, `<>`$\}$
- Zusätzlich:
  - **Nullselektion:** $attribut$ `IS NULL`
  - **Intervallbedingung:** $attribut$ `BETWEEN` $konst_1$ `AND` $konst_2$
  - **Verbundbedingung:** $tabelle_1.attribut_1$ `=` $tabelle_2.attribut_2$
  - **Ähnlichkeitsselektion:** $attribut$ `LIKE` $muster$

Ähnlichkeitsbedingung

- Finden von ähnlichen Zeichenketten:

  attribut `[NOT] LIKE` *muster* `[ESCAPE` *zeichen* `]`

- Das Muster kann Wildcards enthalten:
  - Platzhalter für ein beliebiges Zeichen: **_**
  - Platzhalter für mehrere (oder keine) beliebige Zeichen: **%**
- Um nach Wildcards zu suchen, kann ein Escape-Zeichen definiert werden.

6.3.5 Mengenoperationen

Vereinigung, Schnitt, Differenz:

```
SELECT [···]
{ UNION | INTERSECT | EXCEPT } [ALL | DISTINCT]
SELECT [···]
```

- RA-Ausdrücke: $R \cup S$, $R \cap S$, $R - S$
- Das Ergebnis ist eine Multimenge (`ALL`) oder eine Menge (`DISTINCT`, Standard).

6.3.6 Duplikate

```
SELECT DISTINCT attribute
FROM            tabellen;
```

- Das Ergebnis wird von einer Multimenge zu einer Menge.
- Das Ergebnis enthält somit keine Duplikate mehr.

6.3.7 Sortierung

```
SELECT attribute
FROM   tabelle
ORDER BY attribut [ ASC | DESC ], [···];
```

- Sortiert die Ergebnisse in der gegebenen Spaltenreihenfolge.
- Die Sortierrichtung wird durch `ASC` (Ascending, Aufsteigend) und `DESC` (Descending, Absteigend) angegeben.

```
SELECT *
FROM   buecher
ORDER BY titel ASC, jahr DESC;
```

6.3.8 Bedingungen über Tabellen

Folgende Operatoren können auf ganze Tabellen T angewendet werden, sie ergeben dabei TRUE oder FALSE:

- `EXISTS T` Prüft, ob T nicht leer ist.
- `s IN T` Prüft, ob ein s in T existiert.
- `s > ALL T` Prüft, ob s größer ist als alle Werte in T.
- `s > ANY T` Prüft, ob s größer ist als mindestens ein Wert in T.
- Sämtliche Bedingungen können mit NOT negiert werden.
- Anstelle von „Größer“ sind natürlich auch andere Vergleichsoperatoren möglich.

6.3.9 Verschaltete Abfragen

- Anstelle von Bedingungen über Tabellen kann auch eine verschachtelte Anfrage genutzt werden.

```
SELECT ···
FROM   ···
WHERE  ··· IN (
    SELECT ···
    FROM   ···
    [···]
)
```

6.3.10 Tupelvariablen und Produkt

- Werden mehrere Tabellen in der FROM-Klausel angegeben, so werden diese kartesisch multipliziert.
- Dann müssen in der SELECT-Klausel auch die Tabellen angegeben werden, von denen selektiert wird.

6.3.11 SQL und Relationentupelkalkül

SQL realisiert in großen Teilen das RTK, wobei die Attribute hinter dem SELECT die Tupelvariablen und die WHERE-Klausel die beschreibende Formel realisieren.

6.3.12 Verbund (Join)

Durch das Selektieren von mehreren Tabellen wurde bereits ein Verbund erreicht, mit Einführung einer `WHERE`-Bedingung kann ein allgemeiner Theta-Join gebaut werden:

```
SELECT attribute
FROM   tabelle_1, tabelle_2
WHERE  tabelle_1.spalte_1 = WHERE tabelle_2.spalte_2 [AND ···]
```

entspricht dem RA-Ausdruck $tabelle_1 \bowtie_{spalte_1=spalte_2} tabelle_2 = \sigma_{spalte_1=spalte_2}(tabelle_1 \times tabelle_2)$.

In SQL gibt, um klar zwischen Join und Selektion zu unterscheiden, einen `JOIN`-Operator, der noch deutlich mehr Arten von Joins unterstützt:

- `r CROSS JOIN s` Kartesisches Produkt $R \times S$
- `r UNION JOIN s` Vereinigungsmenge $R \cup S$
  Keine Schemakompatibilität erforderlich (nicht vereinbares wird `NULL` gesetzt).
- `r NATURAL JOIN s` Natürlicher Verbund $R \bowtie S$
- `r [INNER] JOIN s ON r.a` Θ `s.b` Theta-Join $R \bowtie_{a \Theta b} S$
- `r { LEFT | RIGHT | FULL } [OUTER] JOIN s ON r.a` Θ `r.b`

6.3.13 Spaltenumbenennung

- Spalten können mit `AS` umbenannt werden.
- Damit kann der Zugriff aus Programmiersprachen und die Lesbarkeit der Ergebnisse erhöht werden.

6.3.14 (Aggregat-) Funktionen

- In SQL gibt es zahlreiche Funktionen, um die Ergebnisse zu manipulieren.
- Beispiele:
  - `POSITION(s IN t)` Position von `s` in `t` (beginnend bei 1).
  - `SUBSTRING(s FROM p TO l)` Teilstring aus `s` mit Länge `l`, beginnend bei `p`.
- Zur Zusammenfassung von den Werten aller Zeilen in eine Spalte gibt es noch viele Aggregatfunktionen:
  - `COUNT(` *ausdruck* `)`: Anzahl der Werte inklusive `NULL`.
  - `MAX(` *ausdruck* `)`: Maximaler Wert ohne `NULL`.
  - `MIN(` *ausdruck* `)`: Minimaler Wert ohne `NULL`.
  - `SUM(` *ausdruck* `)`: Summe der Werte ohne `NULL`.
  - `AVG(` *ausdruck* `)`: Durchschnitt der Werte ohne `NULL`.
- Durch die verschiedene Behandlung von `NULL` durch `COUNT`, `SUM` und `AVG` gilt `SUM / COUNT = AVG` im Allgemeinen nicht!

6.3.15 Gruppierung

- Aggregierte Werte können gruppiert werden, indem die GROUP BY-Klausel verwendet wird.
- Attribute, welche nicht Gruppiert wurden, dürfen dann nur in aggregierter Form vorkommen.
- Die HAVING-Klausel darf ausschließlich Gruppierungsattribute und aggregierte Attribute verwenden und entspricht einem WHERE auf den Gruppierungen.

```
SELECT   grupierungsattribut, aggregierte nicht-gruppierungsattribute
FROM     tabelle
[WHERE   bedingungen]
GROUP BY gruppierungsattribut
[HAVING   grupierungsbedingung];
```

6.4 Datenmanipulationssprache (DML)

- Mit der DML können die Daten in der Datenbank verändert werden.
  - Einfügen von Tupeln: INSERT
  - Aktualisieren von Tupeln: UPDATE
  - Löschen von Tupeln: DELETE
- Veränderungen können dabei einzelne oder auch mehrere Tupel betreffen.
- Werden Integritätsbedingungen verletzt, wird die Änderung nicht akzeptiert.

6.4.1 Daten Löschen

```
DELETE FROM tabelle
[WHERE bedingung];
```

- Die Bedingung funktioniert genau so wie bei einem SELECT.
- Eine Löschung bezieht sich immer auf genau eine Tabelle, auch wenn andere Tabellen in der Bedingung vorkommen dürfen.

Alle Daten Löschen

```
TRUNCATE TABLE tabelle;
```

- Löscht alle Daten der Tabelle, logisch äquivalent zu DELETE FROM *tabellenname*.
- In der Praxis aber meist schneller, da ausschließlich der „Füllgrad“ der Tabelle auf 0 gesetzt wird und neue Daten somit die alten überschreiben und diese nicht explizit gelöscht werden müssen.

6.4.2 Daten Ändern

```
UPDATE tabelle
SET spalte_1 = ausdruck_1, [···] spalte_n = ausdruck_n
[WHERE bedingung];
```

- Wurde keine `WHERE`-Klausel angegeben, so bezieht sich die Änderung auf die gesamte Tabelle.
- Der Ausdruck kann ein konstanter oder berechneter Wert sein.

6.4.3 Daten Einfügen

```
INSERT INTO tabelle [(spalte_1, [···] spalte_n)]
VALUES (wert_{1,1}, [···] wert_{1,n}), [···] (wert_{k,1}, [···] wert_{k,n});

INSERT INTO tabelle [(spalte_1, [···] spalte_n)]
VALUES (SELECT [···]);
```

- Die Liste der Spaltennamen ist optional. Wurden keine angegeben, so müssen die Werte genau zu dem Schema der Tabelle passen.
- Nicht in der Spaltenliste angegebene Attribute werden mit `NULL` oder dem Default-Wert belegt.
- Auch kann ein verschachtelte Anfrage angegeben werden, um bspw. Daten zu kopieren.

6.5 Weiterführende Themen

6.5.1 Sichten (Views)

```
CREATE VIEW sichtname AS
    anfrage
[ WITH [CASCADE | LOCAL] CHECK OPTION ];

DROP VIEW sichtname;
```

- Mit Sichten (Views) könne virtuelle Tabellen angelegt werden, auf denen Anfragen ausgeführt werden können.
- Theoretisch kann man Views auch vollständig durch geschachtelte Anfragen ersetzen, dies kann aber sehr unübersichtlich werden.
- Die Abfrage werden aber meist nicht gespeichert, sondern jedes Mal neu erstellt! $\Longrightarrow$ Vorsicht bei großen Anfragen.
- Mit `CHECK OPTION` wird verhindert, dass für die Sicht nicht sichtbare Daten geändert werden.

Vorteile

- Vereinfachung von Anfragen
- Zuschnitt auf die Bedürfnisse des Nutzers/der Anwendung
- Logische Datenunabhängigkeit (bei Änderungen der darunterliegenden Struktur kann eine Sicht dennoch die gleichen Ergebnisse liefern)
- Zugriffskontrolle

Probleme Durch Anwendung von Änderungsoperationen auf Sichten könne viele Probleme entstehen:

- **Effektkonformität:** Aktualisierung der Sicht sollte den gleichen Effekt haben wie Aktualisierungen an den Tabellen.
- **Minimalität:** Die Datenbank sollte so wenig wie möglich geändert werden, um den gewünschten Effekt zu erzielen.
- **Konsistenzerhaltung:** Integritätsbedingungen dürfen nicht verletzt werden.
- **Zugriffskontrolle:** Aktualisierungen dürfen nur die von der Sicht eingeschlossenen Daten betreffen.

7 SQL und Programmiersprachen

7.1 Datenbankanfragen

Anfragen an die Datenbank, bzw. das DBMS, können auf zwei grundlegend verschiedene Arten gestellt werden:

- Ad-Hoc von einem Nutzer über die Kommandozeile oder Query-Tool.
- Durch eine Anwendung, welche die Daten mglw. weiter verarbeitet.

7.2 Cursor

- Die meisten Programmiersprachen haben keine Mengen, sondern erwarten einzelne Tupel.
- Dieses Tupel (das Ergebnistupel), eine Array-artige Struktur, wird nacheinander mit Daten befüllt.
- Dies wird mit Hilfe eines Cursors umgesetzt.

```
DECLARE cursorname CURSOR FOR query;

OPEN cursorname;

-- Laden des ersten Ergebnistupels.
FETCH cursorname;
-- Laden des zweiten Ergebnistupels.
FETCH cursorname;
...
-- Laden des letzten Ergebnistupels.
FETCH cursorname;
-- Versuch, noch eines zu laden.
-- Dieses ist leer --> letztes Element erreicht.
FETCH cursorname;

CLOSE cursorname;
```

7.3 Zugriff aus Java

```
void insertBook() throws SQLException {
    try (Connection con =
        DriverManager.getConnection("jdbc:postgresql://user:password@localhost:3306/library");
            Statement stmt = con.createStatement()) {
        stmt.execute("INSERT INTO books (title, author, price) VALUES ('Java ist auch eine Insel',
            'Christian Ullenboom', 4990)");
    }
}
```

- Zum Zugriff auf eine Datenbank aus Java muss ein JDBC-Treiber geladen werden.
- Für PostgreSQL gibt es bspw. einen Treiber unter https://jdbc.postgresql.org/download.html.

7.3.1 Abfragen

```
void listBooks(Connection con) throws SQLException {
    val stmt = con.createStatement();

    // Declare cursor.
    ResultSet result = stmt.executeQuery(SELECT title, author FROM books)

    // Fetch tupel.
    while (result.next()) {
        System.out.println(result.getString(1) + " von " + result.getString(2));
    }

    // Close cursor.
    result.close();

    stmt.close();
}
```

7.3.2 Wiederkehrende Anfragen

Da Anfragen, werden sie mit `executeQuery(···)` ausgeführt, erst vorbereitet werden müssen, können `PreparedStatements` verwendet werden, damit der Query nur einmal vorbereitet werden muss und mehrmals ausgeführt werden kann:

```
void doThings(Connection con) throws SQLException {
    // Prepare the query.
    val pstmt = conn.prepareStatement("SELECT author, title FROM books");

    // Execute the query multiple times without preparing it ever again.

    val resultSet = pstmt.executeQuery();
    ···
    resultSet.close();

    ···

    val resultSet = pstmt.executeQuery();
    ···
    resultSet.close();

    pstmt.close();
}
```

7.3.3 Parametrisierte Anfragen

Werden dynamisch Queries einfach durch Konkatenation aufgebaut, ist dieser Code anfällig für SQL Injection.

Besser werden auch hier `PreparedStatements` genutzt, welche die Parameter korrekt Escapen:

```
// Condition: titles.length == authors.length == prices.length
void insertBooks(String[] titles, String[] authors, int[] prices) {
    val pstmt = conn.prepareStatement("INSERT INTO books (title, author, price), (?, ?, ?)");

    for (int i = 0; i < titles.length; i++) {
        String title = titles[i];
        String authors = authors[i];
        int price = prices[i];

        pstmt.setString(1, title);
        pstmt.setAuthor(2, author);
        pstmt.setInt(3, price);
        pstmt.execute();
    }
```

```
15
16      pstmt.close();
17  }
```

7.4 SQL Injection

Wird ein Query wie

```
1  stmt.executeQuery("SELECT hash FROM users WHERE username = '" + login + "'");
```

aufgebaut, dann kann ein Nutzer durch die Eingabe von `Fabian'; DROP DATABASE library;` das System zur Ausführung von folgendem Query verleiten:

```
1  SELECT hash FROM users WHERE username = 'Fabian'; DROP DATABASE library;'
```

wodurch die gesamte Datenbank gelöscht wird.

7.5 Objektrelationale Abbildung (ORM)

- Durch ORM können Tabellen direkt und automatisch auf Objekte abgebildet werden.
- Damit entfällt das manuelle Lesen der Daten und das Ausdenken eines Schemas.
- Somit ist ein höherer Abstraktionsgrad möglich.
- Das bekannteste Framework für ORMs ist Hibernate (https://hibernate.org/).
- Für weitere Informationen siehe unter anderem https://vimeo.com/28885655.

8 Transaktionen und Mehrbenutzerbetrieb

8.1 Transaktionen

- Eine *Transaktion* ist ein atomarer Prozess, der eine Datenbank von einem konsistenten in einen anderen konsistenten Zustand überführt.
- Eine Transaktion ist die minimale Prozesseinheit in einem Datenbanksystem.
- Transaktionen werden gestartet (BOT, Begin of Transaction) und gestoppt (EOT, End of Transaction).
- Zwischen BOT und EOT stehen:
  - SQL-Anfragen zur Manipulation, Abfrage, etc. von Daten
  - Programmteile einer anderen Programmiersprache, die nicht an das DBMS übergeben werden.

8.1.1 Terminierung

Eine Transaktion muss immer terminieren.

- Normale Terminierung: Commit
  - Die Änderungen werden permanent in die DB geschrieben.
  - Ein Zurückrollen ist nicht mehr möglich.
  - Der Auto-Commit-Modus nach jedem Befehl ist für den interaktiven Betrieb (Command Line, ...) sinnvoll.
- Abnormale/Vorzeitige Terminierung: Abort
  - Der Zustand der Datenbank wird zu dem Zustand vor BOT zurück gerollt.
  - Ist SQL ist dies in der Regel als `ROLLBACK` definiert.
- Ein Abort kann unterschiedlich eingeleitet werden, bspw. durch:
  - den Benutzer,
  - das Anwendungsprogramm oder
  - das DBMS, wenn lange Zeit kein Commit erfolgt ist (Timeout).
- Abgebrochene Transaktionen können:
  - Neu gestartet werden, wenn sie durch einen Hard- oder Systemfehler aufgetreten sind (z.B. Deadlocks).
  - Entfernt werden, insbesondere bei fehlerhaften Transaktionen (z.B. bei Division durch Null).

8.1.2 Konsistenz

- Datenbankkonsistenz
  - Eine Datenbank ist konsistent, wenn alle Konsistenzregeln erfüllt wurden (Fremdschlüssel, Primärschlüssel, `NOT NULL`).
- Transaktionskonsistenz
  - Transaktionen bilden einen konsistenten auf einen anderen konsistenten Zustand ab.
  - Die Datenbank muss vor und nach der Ausführung konsistent bleiben!
  - Nebenläufige Transaktionen dürfen sich nicht behindern und produzieren keine inkonsistenten Zustände.

8.1.3 ACID-Eigenschaften

Transaktionen sollten die ACID-Eigenschaften einhalten:

- **A**tomicity - Atomarität
  Transaktionen werden entweder vollständig oder gar nicht ausgeführt.
- **C**onsistency - Konsistenz
  Transaktionen produzieren nur konsistente Zustände.
- **I**solation - Isolation
  Transaktionen beeinflussen sich nicht gegenseitig. Veränderungen werden erst nach einem Commit sichtbar.
- **D**urability - Dauerhaftigkeit
  Veränderungen sind permanent.

8.2 Anomalien im Mehrbenutzerbetrieb

Wenn mehrere Transaktionen parallel ablaufen und diese auf die gleichen Daten zugreifen, kann es zu Anomalien kommen. Die wichtigsten Anomalien werden im folgenden behandelt und sind:

- Inkonsistentes Lesen
- Verlorene Updates
- Schreib-Lese-Konflikt
- Phantomproblem

8.2.1 Inkonsistentes Lesen

t	Transaktion TX1	Transaktion TX2	X	
1	READ(X)		10	
2		READ(X)	10	
3		X := X + 1	10	
4		WRITE(X)	11	
5		COMMIT	11	
6	READ(X)		11	↯

- Die gleiche Leseoperation in TX1 liefert unterschiedliche Ergebnisse.

8.2.2 Verlorene Updates

t	Transaktion TX1	Transaktion TX2	X	
1	READ(X)		10	
2		READ(X)	10	
3	X := X + 10		10	
4		X := X + 1	10	
5	WRITE(X)		20	
6		WRITE(X)	11	↯

- Zeitliche Überschneidung von zwei Transaktionen.
- Durch konkurrierendes Lesen/Schreiben geht die Änderung von TX1 vollständig verloren.

8.2.3 Schreib-Lese Konflikt

t	Transaktion TX1	Transaktion TX2	X	
1	READ(X)		10	
2	X := X + 10		10	
3	WRITE(X)		20	
4		READ(X)	20	
5		X := X + 1	20	
6		WRITE(X)	21	
7		COMMIT	21	
8	ABORT		?	↯

- Aktualisierung von X durch TX1, die anschließend fehlschlägt.
- TX2 nutzt den falschen Wert, nach $t = 8$ sollte eigentlich $X = 11$ gelten.

8.2.4 Phantomproblem

t	Transaktion TX1	Transaktion TX2	#	
1	`READ(all rows)`		10	
2		`WRITE(insert a row)`	11	
3		`COMMIT`	11	
4	`READ(all rows)`		11	↯
5	`COMMIT`			

- Ähnlich wie das inkonsistente Lesen, nur verändern sich hier nicht die Elemente, sondern die Anzahl dieser.
- Das von TX2 hinzugefügte Tupel heißt *Phantom*.
- Dies kann nicht durch das Sperren eines Tupels verändert werden → Tabellensperre.

8.3 Serialisierbarkeit

8.3.1 Serielle vs. Verschränkte Ausführung

- Serielle Ausführung
  - Es kann nur eine Transaktion zeitgleich ausgeführt werden → langsam.
  - Per Definition korrekt (keine Anomalien).
  - Bei n Transaktionen gibt es $n!$ Ausführungspläne.
- Verschränkte Ausführung
  - Mehrere Transaktionen werden parallel ausgeführt → schneller.
  - Jede Transaktion behält ihre Reihenfolge bei.
  - Es kann zu den vier vorgestellten Anomalien kommen.
  - Durch *Serialisierbarkeit* können Anomalien vermieden werden.

8.3.2 Serialisierbarkeit

- Serielle Ausführung sind frei von Anomalien.
- Es wird versucht, verschränkte Ausführungen zu finden, sodass diese den gleichen Effekt wie eine serielle Ausführung hat.
- Eine solche Ausführung (Schedule) heißt *serialisierbar*.
- Zwei Schedules, die den gleichen Effekt haben, sind äquivalent.

$$S, S' \in \text{Schedule}, \qquad S \sim S' \iff \forall \text{DB} : S(DB) = S'(DB)$$

- Dies wird bspw. durch *Sperren* erreicht.

8.3.3 Sperren

- Lesesperre LOCK-S (Shared Lock)
  - Ein Datenobjekt darf von einer Transaktion gelesen, aber nicht geschrieben werden.
  - Mehrere Transaktionen können zeitgleich eine Lesesperre auf das Objekt haben.
- Schreibsperre LOCK-X (Exclusive Lock)
  - Ein Datenobjekt darf von einer Transaktion gelesen und geschrieben werden.
  - Eine Transaktion hat exklusiven Zugriff auf die Daten.
- In den meisten Fällen können Sperren auf einzelne Zeilen (Zeilensperre) oder ganze Tabellen (Tabellensperre) gesetzt werden.
- Ein Zugriff auf eine gesperrte Zeile/Tabelle muss warten, bis die Zeile/Tabelle wieder entsperrt ist $\rightarrow$ die Transaktion wird pausiert.

Problematiken

- Die Transaktionen müssen länger warten, bevor diese ausgeführt werden $\rightarrow$ längere Ausführungszeiten.
- **Verklemmung** (Deadlock): Mehrere Transaktionen hindern sich gegenseitig an der Ausführung $\rightarrow$ sie warten ewig wechselseitig aufeinander.
- Somit sollte nur gesperrt werden, wenn die wirklich nötig ist.

8.3.4 Verklemmung (Deadlock)

t	Transaktion TX1	Transaktion TX2	Sperren
1	`X := 1`		LOCK-X(TX1, X)
2		`Y := 2`	LOCK-X(TX1, X); LOCK-X(TX2, Y)
3	`READ(Y)`		LOCK-X(TX1, X); LOCK-X(TX2, Y)
4		`READ(X)`	LOCK-X(TX1, X); LOCK-X(TX2, Y)

- TX1 wartet ab $t = 3$ auf TX2.
- TX2 wartet ab $t = 4$ auf TX1.

  $\rightarrow$ Verklemmung $\implies$ keine Terminierung

Detektion

- Deadlocks können durch *aktives* Prüfen von Zyklen in einem Wait-For-Graph (WFG) erkannt werden.
- Transaktionen sind dabei Knoten und Anforderungen an gesperrte Ressourcen sind Kanten.

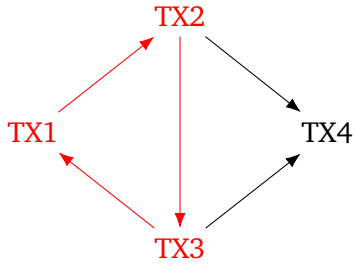


Abbildung 8.1: Beispiel: Wait-For-Graph mit Zyklen

- Tritt, wie in Abb. 8.1, ein Zyklus auf, dann existiert eine Verklemmung.
- Alternativ können Deadlocks *passiv* über Timeouts bestimmt werden.
- Wurde ein Deadlock erkannt, so müssen die Transaktionen abgebrochen (Abort) werden, um die Ressourcen wieder freizugeben.

8.4 Sperrprotokolle

8.4.1 Zwei-Phasen Sperrprotokoll (2PL)

Phase 1 Es werden nur gesperrte Daten gelesen, eingefügt, gelöscht oder verändert.

Phase 2 Nach einer Fragegabe einer Sperre kann keine neue Sperre mehr erstellt werden.

- Aus Phase 2 folgt somit, dass es *Sperrphasen* und *Freigabephasen* geben muss.
- 2PL garantiert dabei serialisierbare Abläufe.
- **Aber:** 2PL ist nicht verklemmungsfrei!
  - Der Grund ist die dynamische Verwaltung von Sperren, siehe Beispiel in Abschnitt 8.3.4.

8.4.2 Konservative Sperrprotokolle

Ein konservatives 2PL ist ähnlich dem 2PL, mit folgenden Modifikationen:

- Alle nötigen Sperren werden direkt nach dem BOT gesetzt (Preclaiming).
- Es gibt keine eigentliche Sperrphase.

Damit ist das konservative 2PL verklemmungsfrei. Allerdings weiß eine Transaktion vor der Ausführung noch nicht genau, welche Sperren gesetzt werden müssen, weshalb meistens mehr Sperren als nötig gesetzt werden $\Longrightarrow$ langsamere Ausführung.

Daher wird das konservative 2PL in der Praxis kaum genutzt.

8.4.3 Strikte Sperrprotokolle

Bei striktem 2PL werden:

- Die Sperren ausschließlich am EOT freigegeben, also bei einem Commit oder Abort.
- Es gibt keine eigentliche Freigabephase.
- EOT ist der einzig sichere Zeitpunkt, ab dem keine Befehle mehr ausgeführt werden.
- Somit wird kaskadierendes Zurücksetzen verhindert.
- Bei konservativem striktem 2PL (KS2PL) werden alle Sperren beim BOT gesetzt und erst beim EOT freigegeben.
  → Gar keine Sperr-/Freigabephase.

8.5 Isolationsgrade

- Mit dem Isolationsgrad kann in SQL bestimmt werden, in welchem Ausmaß Transaktionen konkurrierenden Transaktionen ausgesetzt sind.
- Damit kann der Benutzer zwischen höherer Performanz und höherer Ausführungssicherheit wählen.
- In SQL wird der Isolationsgrad wie folgt gesetzt:

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SERIALIZABLE };
```

8.5.1 `SERIALIZABLE`

- *Eigene Schreibsperren* werden angefordert uns bis EOT gehalten.
- *Eigene Lesesperren* werden angefordert und bis EOT gehalten.
- *Fremde Schreibsperren* werden beachtet (es werden nur Daten gelesen, die committed wurden).
- Entspricht dem strikten 2PL.
- Langsamere Ausführung, aber frei von Anomalien.

8.5.2 `REPEATABLE READ`

- *Eigene Schreibsperren* werden angefordert und bis EOT gehalten.
- *Eigene Lesesperren* werden angefordert uns bis EOT gehalten.
- *Fremde Schreibsperren* werden beachtet (es werden nur Daten gelesen, die commited wurden).
- Gleiche Sperrstrategie wie `SERIALIZABLE`, setzt aber *keine Indexsperren*.
- Es können Phantomprobleme auftreten.

8.5.3 `READ COMMITTED`

- *Eigene Schreibsperren* werden angefordert und bis EOT gehalten.
- *Eigene Lesesperren* werden angefordert und *sofort freigegeben*.
- *Fremde Schreibsperren* werden beachtet (es werden nur Daten gelesen, die commited wurde).
- Reduzierte Wartezeit beim Lesen.
- Es kann zu inkonsistentem Lesen kommen.

8.5.4 `READ UNCOMMITTED`

- *Eigene Schreibsperren* sind nicht vorhanden, `READ UNCOMMITTED` ist *read only*.
- *Eigene Lesesperren* werden angefordert und *sofort freigegeben*.
- *Fremde Schreibsperren* werden *nicht beachtet* (es werden auch Daten gelesen, die noch nicht commited wurden).
- Eignet sich für statistische Prozesse, bei denen ein Überblick ausreicht.
- Effiziente Ausführung.

8.5.5 Übersicht

	Eigene Indexsperren	Eigene Lesesperren	Eigene Schreibsperren	Fremde Schreibsperren
`SERIALIZABLE`	bis EOT	bis EOT	bis EOT	beachtet
`REPEATABLE READ`	N/A	bis EOT	bis EOT	beachtet
`READ COMMITTED`	N/A	nur kurzzeitig	bis EOT	beachtet
`READ UNCOMMITTED`	N/A	nur kurzzeitig	read only	nicht beachtet

Tabelle 8.1: Isolationsgrade und Sperren

	Phantom Read	Non-repeatable Read	Lost Update	Dirty Read
`SERIALIZABLE`	verhindert	verhindert	verhindert	verhindert
`REPEATABLE READ`	möglich	verhindert	verhindert	verhindert
`READ COMMITTED`	möglich	möglich	möglich	verhindert
`READ UNCOMMITTED`	möglich	möglich	möglich	möglich

Tabelle 8.2: Isolationsgrade und Anomalien

9 Indexstrukturen und Performanz

- Bei immer größeren Datenmengen dauert die Ausführung von Anfragen immer länger.
- Mögliche Lösungen:
  - Bessere Hardware.
  - Alternative DBMS einsetzen, siehe Kapitel 10.
  - Optimierung der DBMS-Einstellungen.
  - Gezielter Einsatz von Indexstrukturen, dieses Kapitel.
  - ...

9.1 Datenbankindizierung

- Szenario: Eine Tabelle enthält 1 Mio. Zeilen, in denen genau ein Datum gesucht wird.
- Im Worst Case müssen somit 1.000.000 Zeilen überprüft werden (die gesuchten Daten existieren nicht oder stehen in der letzten Zeile).
- Zur Beschleunigung könnten die Daten sortiert werden, bzw. von vornherein sortiert eingefügt werden.

9.1.1 Binäre Suche

- Wurden die Daten irgendwie sortiert, so kann binäre Suche angewandt werden.
- Mit dieser müssen nicht alle Zeilen angeschaut werden und das Ergebnis ist schneller findbar.

9.1.2 Sortierte Tabellen

- Es ist eine gute Idee, Tabellen direkt bei der Erstellung und Einfügung von neuen Daten zu sortieren/-sortiert zu halten.
- Allerdings kann jede Tabelle zur einmal sortiert werden (z.B. exklusiv nach Telefonnummer oder Name).
- Außerdem müssen beim Einfügen einer neuen Zeile viele Daten verschoben werden $\implies$ das Einfügen neuer Daten kann mitunter lange dauern.
- Alternativ kann eine separate Datenstruktur genutzt werden, ein Index.

9.1.3 Index

- Ein *Index* enthält redundante Informationen, welche auf die Tabelle verweisen.
- Er muss bei jeder Änderungsoperation auf der eigentlichen Tabelle ebenfalls geändert werden.
  - Dies wird vom DBMS selbstständig erledigt, kann aber viel Zeit kosten.
- Zu den Aufgaben eines Index gehört es,
  - Datenbankabfragen zu beschleunigen mit geeigneter Wahl der Datenstrukturen (z.B. Baumstrukturen, Hashtabellen, ...) und
  - Datenintegrität zu gewährleisten (Eindeutigkeit, Referenzielle Integrität).

Indexverwaltung in SQL

- Anlegen

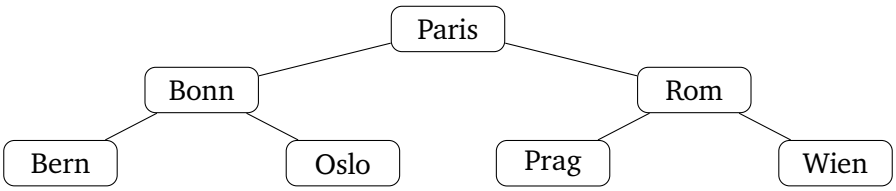
```
CREATE [UNIQUE] INDEX indexame
    ON tabellenname (spalte_1, [···] spalte_n);
```

- Löschen

```
DROP INDEX indexname;
```

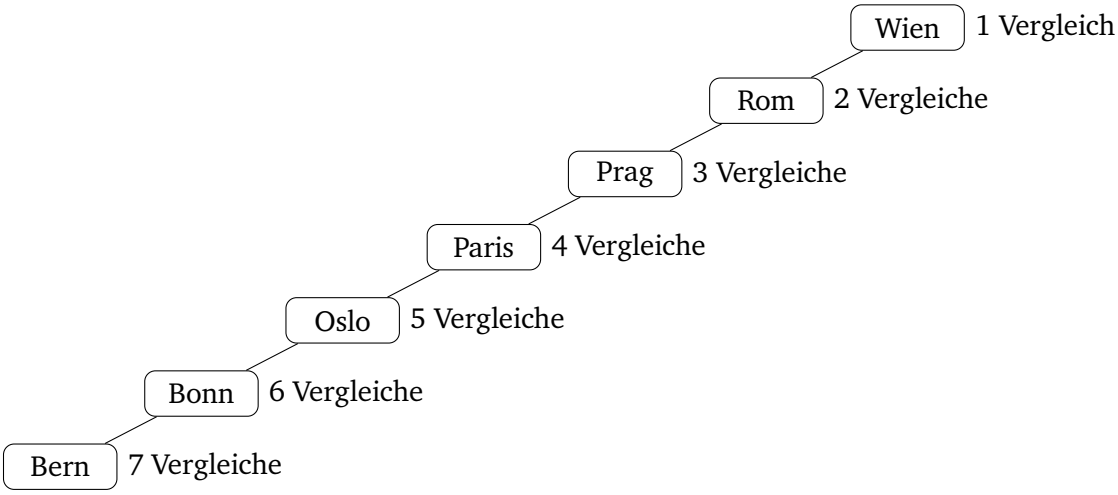
9.2 Indexanatomie

9.2.1 Binärer Suchbaum



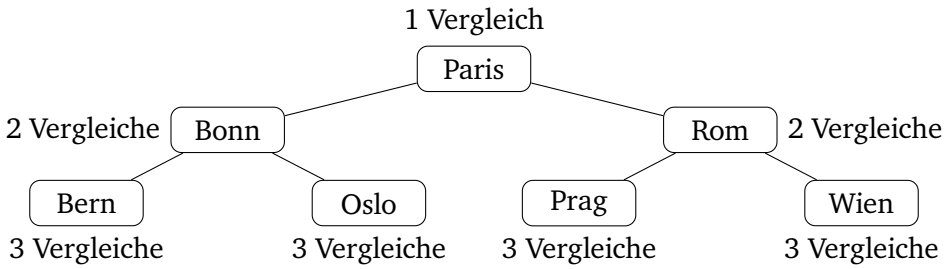
Entarteter Binärer Suchbaum

Aufgabe: Finde die Stadt „Bern“



Balancierter Binärer Suchbaum

Aufgabe: Finde die Stadt „Bern“.



9.2.2 B-Baum

Siehe https://de.wikipedia.org/wiki/B-Baum.

9.2.3 B^+-Baum Index

Siehe https://de.wikipedia.org/wiki/B+-Baum.

9.2.4 Zusammengesetzter Index

Ein zusammengesetzter Index ist ein Index über mehrere Spalten, wobei nur schnell nach Elementen gesucht werden kann, wenn wirklich alle Spalten in der Anfrage spezifiziert werden.

9.3 Ausführungspläne

- Beliebige Datenbankanfragen können bei gleichbleibendem Ergebnis unterschiedlich ausgeführt werden, zum Beispiel durch unterschiedliche Reihenfolge der Operationen.

- Beispiel:

$$\pi_{\text{Hersteller,Modell}}(\sigma_{\text{Land}=\text{„BRD“}}(\text{PKW} \bowtie \text{Standort}))$$
$$=\pi_{\text{Hersteller,Modell}}(\text{PKW} \bowtie \sigma_{\text{Land}=\text{„BRD“}}(\text{Standort}))$$

  Es könnte z.B. einen Index über das Land im Standort geben, dann ist die zweite Anfrage deutlich schneller.

- Der *Ausführungsplan* legt fest, wie die einzelnen Schritte der Abfrage ausgeführt werden.
- In SQL kann der Ausführungsplan mit „`EXPLAIN` *anfrage*“ angezeigt werden.

9.3.1 Optimizer

- Der *Optimizer* erstellt den Ausführungsplan für eine Anfrage; Dies wird häufig als *compiling* oder *parsing* bezeichnet.
- Regelbasierter Optimizer
  - Arbeitet nach einem fest vorgegebenen Regelwerk.
  - Wird heute kaum noch genutzt.
- Kostenbasierter Optimizer
  - Generiert mögliche Ausführungspläne und schätzt die Kosten ab.
  - Die Kosten kombinieren die Anzahl und Art der Operationen und die erwartete Zeilenanzahl.
  - Der beste Ausführungsplan hat die geringsten Kosten.
- Ein kostenbasierter Optimizer benötigt Informationen zu:
  - Anzahl der unterschiedlichen Werte einer Spalte
  - Kleinster und größter Wert einer Spalte
  - `NULL`-Einträge und Wertverteilung einer Spalte
  - Größe einer Tabelle (in Zeilen und Blöcken)
  - Baumtiefe eines Indexes
  - u.v.m.

  um die Kosten abzuschätzen.
- Das DBMS führt dazu Statistiken, die regelmäßig aktualisiert werden müssen.
- In PostgreSQL geht dies mit „`ANALYSE` *tabellenname*“.

9.3.2 Durchsuchen von Tabellen

- Full Table Scan, Table Access Full, SEQ Scan (Indexfrei)
  - Die gesamte Tabelle muss durchsucht werden.
  - Bei einer Tabelle mit 1 Mio. Einträgen und einer üblichen Festplatte[1] dauert dies $1 \cdot 10\text{ms} + 1.000.000 \cdot 0{,}01\text{ms} \approx 10\text{s}$.

[1] Mit 10ms zur Lese-/Schreibkopfpositionierung und $0{,}01$ms für sequentielle Zugriffe.

- Indexzugriff, Index Unique Scan, Index Scan
  - Direkter Zugriff auf einen Index.
  - Da maximal ein Indexeintrag existiert, ist kein Verfolgen des Index nötig.
- Indexscan, Index Range Scan
  - Anfrage an einen Bereich an Indexknoten (bspw. alle Mitarbeiter von `A` bis `D`).
  - Da nur indizierte Spalten verwendet werden, können die Indexknoten verfolgt werden.
- Tabellenzugriffe, Table Access by RowID, RID Lookup
  - Nachladen von nicht im Index erhaltenen Daten.
  - Beispiel: Alle Rufnummern der Mitarbeiter von `A` bis `D`.

9.3.3 Probleme

Viele Tabellenzugriffe

- Bei Anfragen wie „alle Mitarbeiter von `A` bis `D` mit der Vorwahl `06171`“ gibt es sehr viele zufällige Tabellenzugriffe.
- Angenommen es gibt 500.000 Mitarbeiter von `A` bis `D`, dann werden 500.000 wahlfreie Zugriffe produziert:

$$\approx \underbrace{500.000 \cdot 0{,}01\text{ms}}_{\text{Indexscan}} + \underbrace{500.000 \cdot 10\text{ms}}_{\text{Tabellenzugriffe}} = 83\text{min}$$

- Vergleich: Ein Full Table Scan über eine Tabelle mit 1 Mio. Zeilen nutzt sequentielle Zugriffe:

$$\approx 10\text{ms} + 1.000.000 \cdot 0{,}01\text{ms} = 10\text{s}$$

- $\Longrightarrow$ Indexstrukturen sind also nicht in jedem Falle sinnvoll!

Sortierfolge

- Außerdem kann der Index nicht effektiv genutzt werden, wenn unerwartete Anfragen ausgeführt werden. Eine Lösung ist hier, einen zweiten Index anzulegen.

9.4 Indexdesign

- Proaktives Indexdesign
  - Die Indexstrukturen werden schon während des Entwurfs geplant.
  - Mögliche Abfragen werden geplant und gezielt indiziert.
- Reaktives Indexdesign
  - Langsame Anfragen werden im laufenden Betrieb analysiert.
  - Diese werden dann durch geeignete Indexwahl beschleunigt.
- Assistentenbasiertes Indexdesign
  - Es werden verschiedene Indexstrukturen automatisiert durchprobiert.
  - Indexwahl und -aufbau wird optimiert.

9.4.1 1-Sterne Index

- Alle Spalten aus einer `WHERE`-Bedingung werden in beliebiger Reihenfolge indiziert.
- Damit können Selektionen vollständig über den Index abgewickelt werden.

9.4.2 2-Sterne Index

- Die Spalten aus der `ORDER BY`-Klausel werden an den Index angefügt (unter Beachtung der Reihenfolge).
- Die gefilterten Zeilen müssen somit nicht mehr sortiert werden.

9.4.3 3-Sterne Index

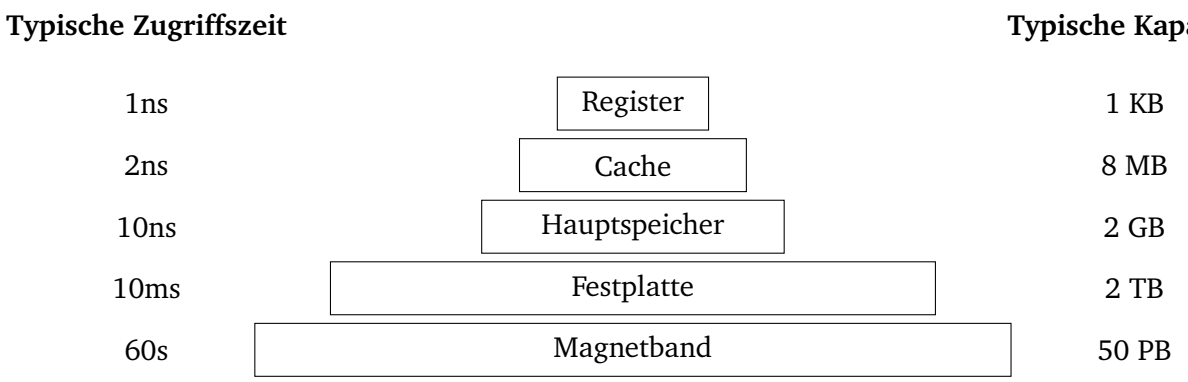
- Alle anderen referenzierten Spalten werden an den Index angefügt (Fat Index oder Covering Index).
- Somit ist gar kein Tabellenzugriff mehr nötig, da alle Spalten im Index sind.

10 Neuere Datenbankkonzepte (Ausblick)

10.1 In-Memory Datenbanken

- Massenspeicherbasierte Datenbank
  - Die Antwortzeiten sind wesentlich durch das Medium bestimmt.
  - Bei wahlfreiem Zugriff $\approx$ 10ms (Festplatte)
- Hauptspeicher Datenbank
  - Die Daten werden im Hauptspeicher gespeichert.
  - Bei wahlfreiem Zugriff $\approx$ 10ns (Unterschied von $\approx 10^6$)
  - Die Daten sind flüchtig; hybride Lösungen speichern die Daten im Hintergrund auf einen Massenspeicher.
    Die Recovery (Wiederaufbau) ist meistens langsam.
  - Der bestimmende Faktor ist die Speichergröße.

10.1.1 Zeitvergleich



Typische Zugriffzeit	Ungefähre Entsprechung
1ns	Raumwechsel im Gebäude (2 Minuten)
2ns	Gebäudewechsel in der Uni (10 Minuten)
10ns	Fahrt nach Köln (2 Stunden)
10ms	Reise zum Pluto (2 Jahre)
60s	Reise zum Andromedanebel (2000 Jahre)

10.1.2 Kompression

- Da der Speicherplatz im Hauptspeicher sehr kostbar ist, müssen die Daten komprimiert werden.
- Bspw. durch Dictionary-Ansätze, d.h. jedem duplizierten Wort wird eine ID zugewiesen und es wird nur die ID gespeichert → nur wenig Speicherplatzverbrauch durch Deduplikation.

10.2 Spaltenorientierte Datenbanken

- Relationen sind immer Zweidimensional (Spalten × Zeilen), die Speicherung muss aber linear erfolgen.
- Ein klassischem RDBMS speichert die Daten Zeilenorientiert.
- Es gibt aber auch Spalteorientierte DBMS.
- Vorteil: Aggregatfunktionen, z.B. `SUM`, sind deutlich schneller.
- Vor allem relevant für Data Warehouses.

10.2.1 Partitionierung

- Einzelne Spaltenteile können auf verschiedenen Server liegen.
- Z.B. kleine werte auf Server 1, große auf Server 2, ...
- Dadurch können Join-Operationen optimiert werden.

10.3 NoSQL

- NoSQL („Not only SQL“ oder „no SQL“) ist ein Sammelbegriff für nicht-relationale Datenbanken.
- Optimiert für große Datenmengen und verteilte Systeme.
- Gut skalierbar und zum Teil noch verfügbar.
- Meistens wird auf ein umfangreiches Transaktionssystem und komplexe Integritätsbedingungen verzichtet.
- Wesentliche Konzepte:
  - Spaltenorientierte DB
  - Key-Value DB
  - Graph-DB
  - Dokumentenorientierte DB

10.3.1 Key-Value Datenbanken

- Grundoperationen
  - `INSERT(k, v)`
    Legt einen Wert `v` unter dem Schlüssel `k` ab.
  - `LOOKUP(k)`
    Holt die Daten mit dem Schlüssel `k`.
  - `DELETE(K)`
    Löscht den Schlüssel `k`.
  - Komplexere Operationen müssen die Anwendungen selbst implementieren.
- Das Format des Wertes ist nicht vorgegeben, es können beliebige Strukturen verwendet werden (Binär, JSON, XML, ...).
- Software: Cassandra, Dynamo, Berkeley DB, BigTable, LevelDB, Redis, ...

10.3.2 Tripel Speicher

- Optimiert für Tripel der Form $(\text{Subjekt}, \text{Prädikat}, \text{Objekt})$.
- Beispiel: $(\text{TU\_Darmstadt}, \text{ist\_eine}, \text{Universität})$
- Wird vor allem im *Semantic Web* eingesetzt, erlaubt Abfragen mit SPARQL.
- Software: Virtuoso, Apache Jena, ...; Umsetzungen: DBpedia, ...

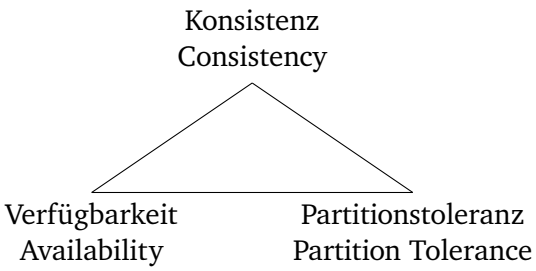
10.3.3 Graphdatenbanken

- Ähnlich zu Tripel Speicher, aber besonderer Fokus auf Graphenalgorithmen, bspw.:
  - Kürzeste Wege
  - Breitensuche
  - Tiefensuche
  - Identifikation von Cliquen
  - ...
- Software: Titan, k-infinity, Neo4J, ...

10.3.4 Dokumentenorientierte Datenbanken

- Optimiert für große Textmengen.
- Besonders relevant ist die Volltextsuche in Texten.
- Software: CouchDB, MongoDB, OrientDB

10.4 CAP-Theorem, Brewers Theorem



Konsistenz Auf allen Knoten werden die gleichen gültigen Daten gespeichert.

Verfügbarkeit Akzeptable Antwortzeiten.

Partitionstoleranz Das System kann Ausfälle einzelner Knoten verkraften.

> Brewer: Es können nicht alle drei Kriterien zeitgleich optimiert werden.