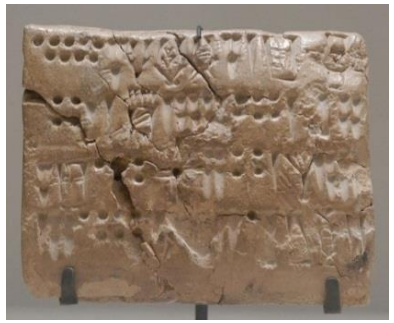
1. Einführung

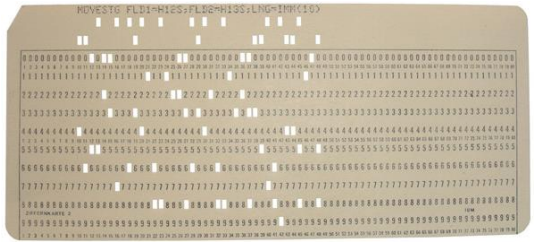
1. **Historische Aspekte und Grundbegriffe**
2. Abgrenzung zu Dateisystemen
3. Inhalte von Datenbanken
4. Architektur von Datenbanksystemen

Motivation von Datenbanken als Buchführungsinstrument

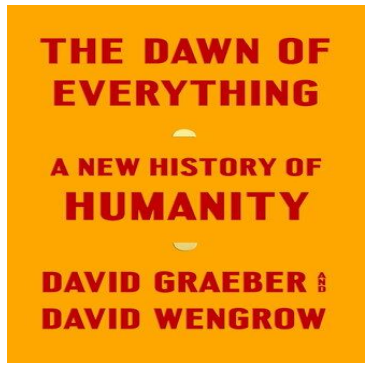
- Tontafeln von Uruk (3.200 v.Chr.)
- Druckerpresse (Gutenberg, ca. 1450)
- Lochkarten (1890)
- PC (1984)
- Verbreitetes Internet/WWW (1991)

exponentielle Tempoverschärfung





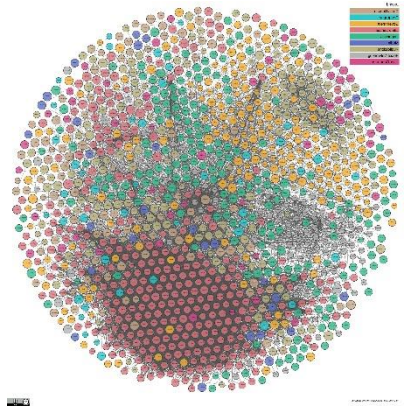
http://datentraeger-museum.de



Variety

Thema der Vorlesung

- Unstrukturierte Daten
  - Daten folgen keinem Format, Schema oder Grammatik
  - Text, Bitströme auf Speichern, rohe Video- & Bilddaten, …
- Semi-Strukturierte Daten
  - Daten folgen einem flexiblen Format, optionale Felder, graphbasierte Daten
  - Webseiten, XML documente, JSON, RDF, …
- Strukturierte Daten
  - Daten folgen einem striktem Schema
  - relationale Datenbanken, Tabellen, Sensordaten, …



Relation "Mitarbeiter"

Personal-nummer	Name	Gehalt
1427	Meier	3217,33
8219	Schmidt	1425,87
2624	Müller	2438,21
⋮	⋮	⋮

Relation "Mitarbeiter in Abteilung"

Personalnummer	Abteilung
1427	3-1
8219	2-2
2624	3-1
⋮	⋮

Einführung

- *Datenbanksysteme* sind zentrale Komponenten großer *Informationssysteme* in vielen Anwendungsbereichen:
  - administrativ-betriebswirtschaftlich: Abrechnung, Buchung, Planung usw.
  - technisch-wissenschaftlich: CAD/CAM/CIM, Geographie, Medizin, Biologie, usw

- Die primäre Aufgabe von Datenbanksystemen umfasst
  - die Beschreibung,
  - die Speicherung und Pflege sowie
  - die Wiedergewinnung

  von (umfangreichen) Datenmengen, die von verschiedenen Anwendungsprogrammen benutzt werden.

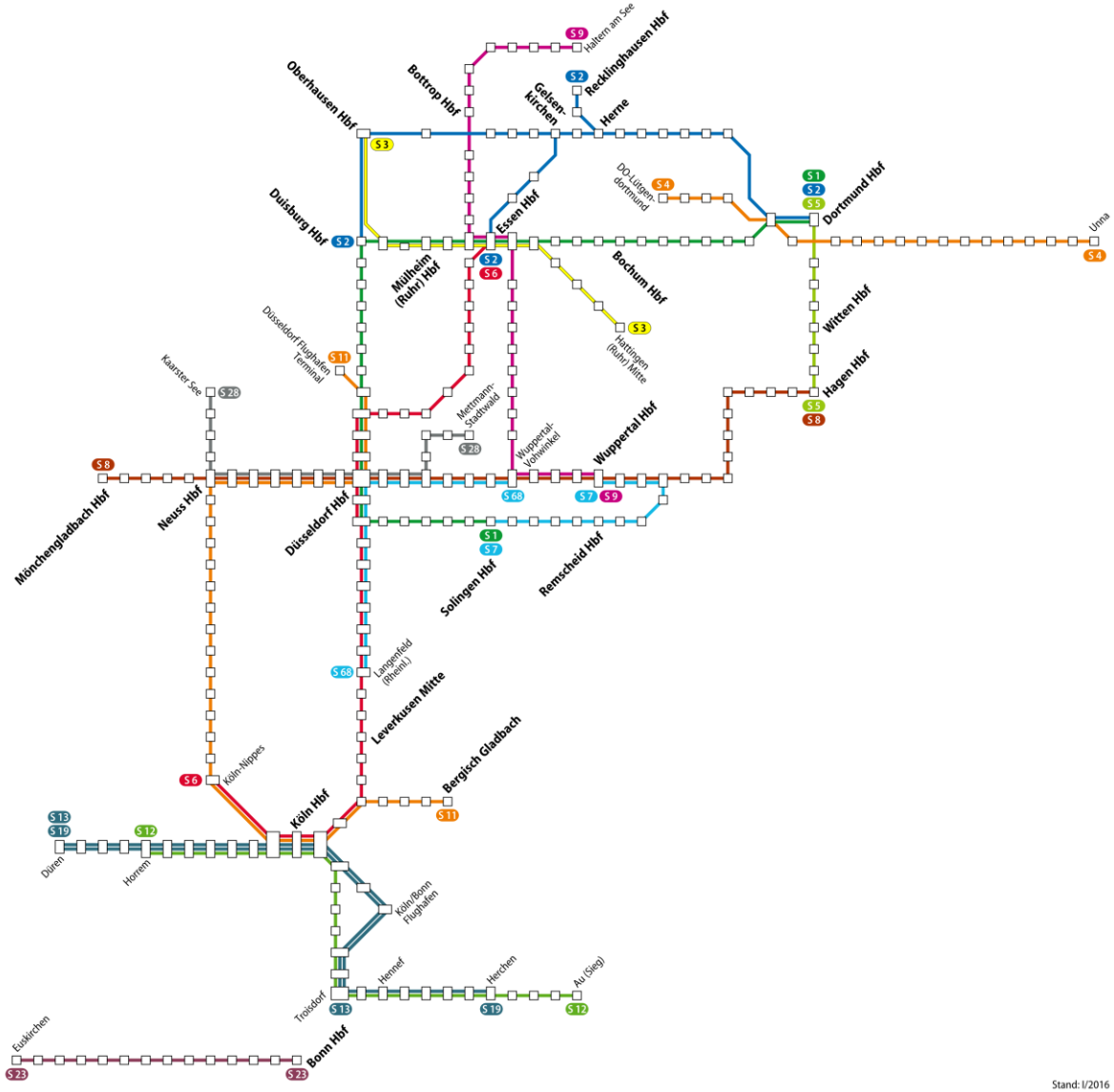
Ausrichtung dieser Vorlesung

- Datenbanken aus Benutzersicht
  - Beschreibung von Daten
    - Datenmodelle, Datenbankschema, Entwurfstheorie
  - Arbeiten mit Datenbanken
    - Anfragesprachen, Transaktionen

- Was ist charakteristisch für Datenbanken (im Gegensatz zu prozeduralen Programmen)?
  - Datenorientierte Programmierung
    - Beschreibe Daten, nicht Abläufe
    - Deklarativer Zugriff: Auf **was** möchte ich zugreifen (nicht: **wie**)?
  - Persistenz von Daten
    - Persistenz: Daten überleben Prozesse
    - „Daten sind schon da“-Programmierung
    - Dadurch Legacy-Problematik: Portierung ist schwieriger

Einschub: Abstraktion in der Informatik [1]

- **Abstraktion** (abstraction) – Das Ableiten oder Herausheben des unter einem bestimmten Gesichtspunkt Wesentlichen/Charakteristischen/Gesetzmäßigen aus einer Menge von Individuen (Dingen, Beobachtungen, ...)
- Abstraktionen sind ein zentrales Mittel für das Erstellen und Verstehen von Systemen und Modellen.



[1] Abschnitt basierend auf: Martin Glinz: Abstraktion. Vorlesung Informatik II: Modellierung, Kap. 12: Abstraktion. Universität Zürich https://files.ifi.uzh.ch/rerg/arvo/ftp/inf_II/inf_II_kapitel_12.pdf

Karte: NordNordWest, Lizenz: Creative Commons by-sa-3.0 de

Abstraktion beim Erstellen von Modellen

Abstraktion ist erforderlich:

- Zur Umsetzung der Pragmatik: Was ist unter welchem Gesichtspunkt wichtig / hervorzuheben?
- Beim Verkürzen: welche Eigenschaften des Originals werden nicht modelliert?

Abstraktion beim Verstehen von Modellen

- Abstraktion ist notwendig zur Beherrschung der Komplexität großer Modelle
- Große Modelle sind ohne einen klar strukturierten Aufbau nicht verstehbar.
- Abstraktion ermöglicht
  - das Sichtbarmachen großer Zusammenhänge unter Weglassung der Details
  - die Darstellung eines Details unter Weglassung/ starker Vergröberung des Rests
  - die Herstellung eines systematischen Zusammenhangs zwischen Übersichten und Detailsichten.

Abstraktionsarten

Die wichtigen Abstraktionsarten für Modelle sind:

- **Klassifizierung:** Charakterisierung der Gemeinsamkeiten von Individuen durch Typen oder Klassen
- **Komposition:** Zusammensetzen einer Menge zusammenhängender Individuen zu einem Ganzen
- **Generalisierung:** Verallgemeinerung der Merkmale einer Menge ähnlicher Typen
- **Benutzung:** Nutzung von Leistungen Dritter durch ein Individuum zwecks Erbringung eigener, höherwertiger Leistungen

- Alle vier Abstraktionsarten sind unabhängig voneinander

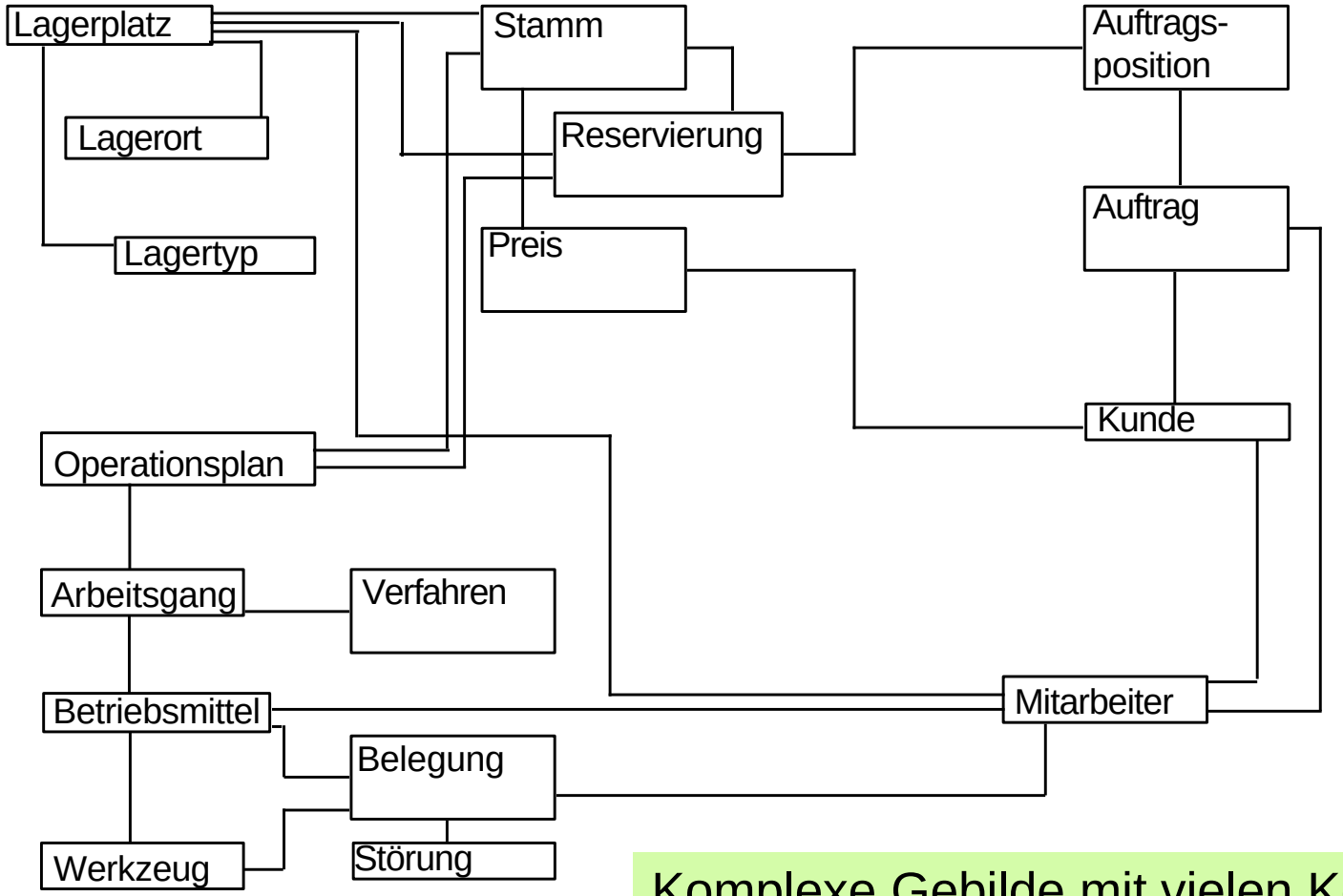
Klassifizierung

Der Typ bzw. die Klasse

- beschreibt die Attribute der Exemplare
- beschreibt den zulässigen Wertebereich zu jedem Attribut
- lässt alle Information zu individuellen Attributausprägungen der Exemplare weg (diese wird zu Wertebereichen abstrahiert)

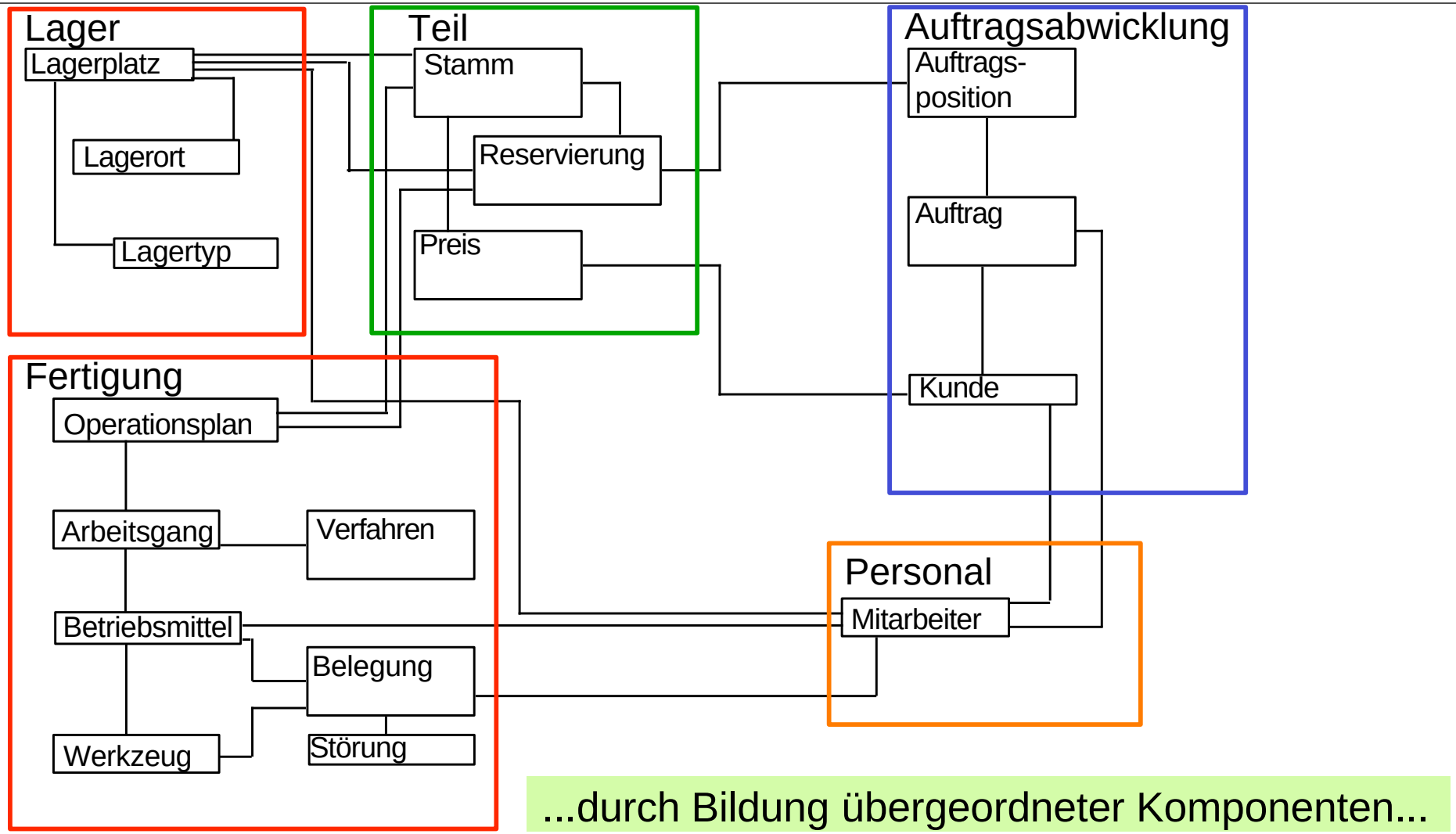
⇨ Zentrale Abstraktion bei jeder Modellbildung

Komposition



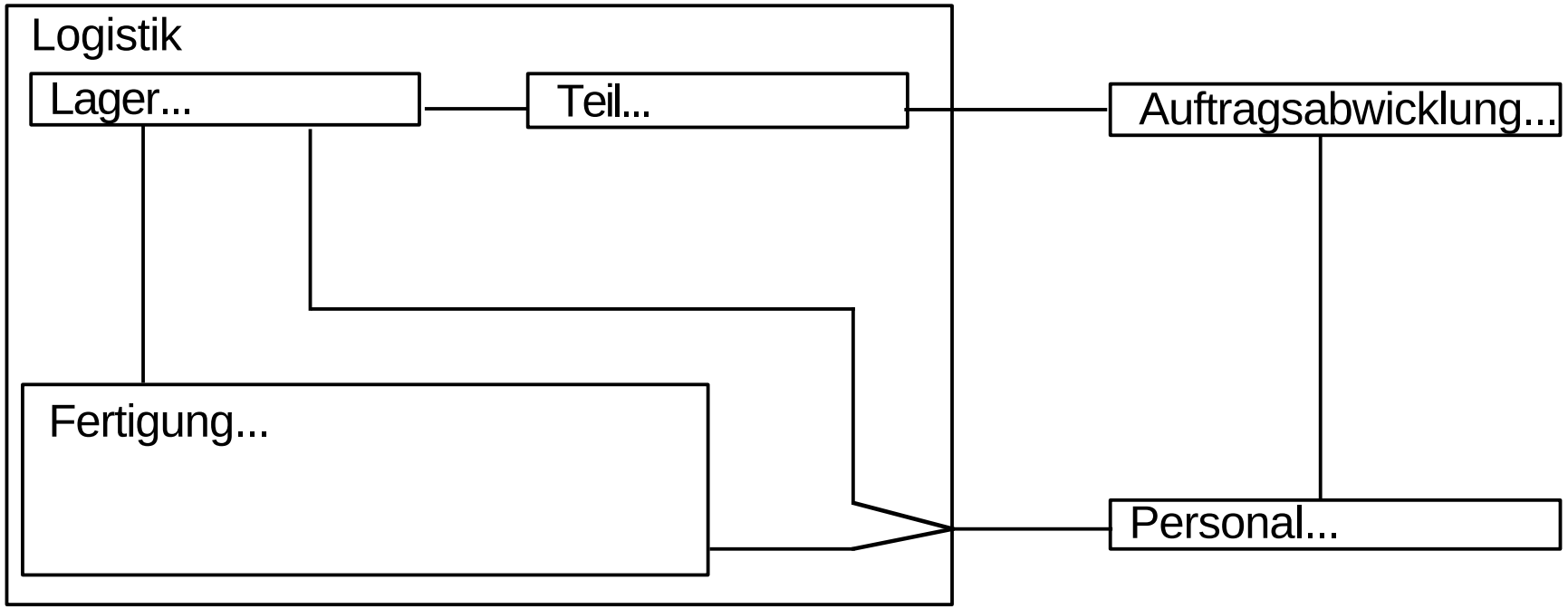
Komplexe Gebilde mit vielen Komponenten...

Komposition



...durch Bildung übergeordneter Komponenten...

Komposition-3



...abstrahieren und übersichtlicher machen

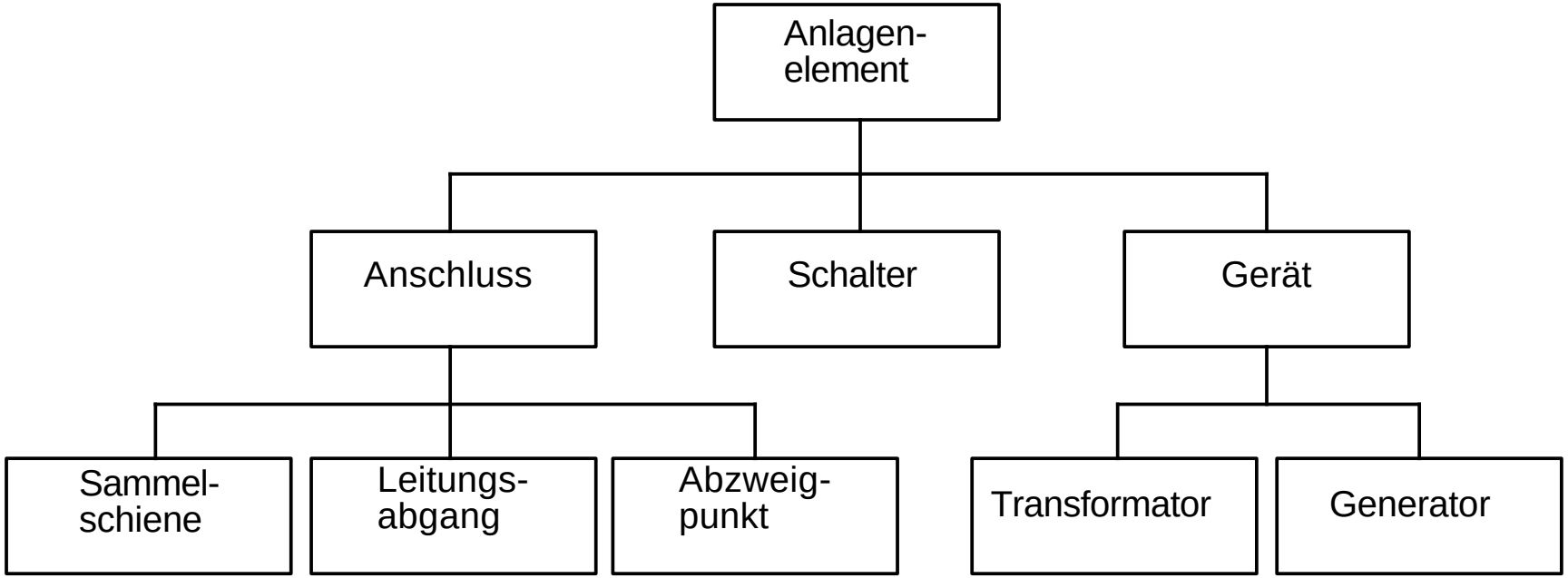
Komposition-4

- **Komposition** (composition) – Zusammenfassung einer Menge von Individuen mit teilweise gemeinsamen Merkmalen zu einem neuen Individuum mit neuen Merkmalen, welches die Gesamtheit der zusammengefassten Merkmale repräsentiert
- Die Komposition
  - fasst eine Menge von Einzelkomponenten (Teilen) unter Weglassung von Details zu einer übergeordneten Komponente (einem Ganzen) zusammen
  - fasst nicht irgendetwas irgendwie zusammen, sondern nur logisch zusammengehörende Komponenten
- Die übergeordnete Komponente (das Ganze) ist ein in sich möglichst geschlossenes Teilmodell

Komposition-5

- Zusammenfassung über mehrere Stufen
  ⇨Kompositionshierarchie (Teil-Ganzes-Hierarchie)
- Kompositionshierarchien bilden in umgekehrter Richtung eine hierarchische Zerlegung eines Modells
- Zentrales Mittel zur Beherrschung der Komplexität von Modellen mit vielen Einzelkomponenten

Generalisierung



Beispiel einer Generalisierungsabstraktion – Modellierung der Begriffswelt eines Problembereichs (Engineering von Unterstationen in Stromverteilungsnetzen)

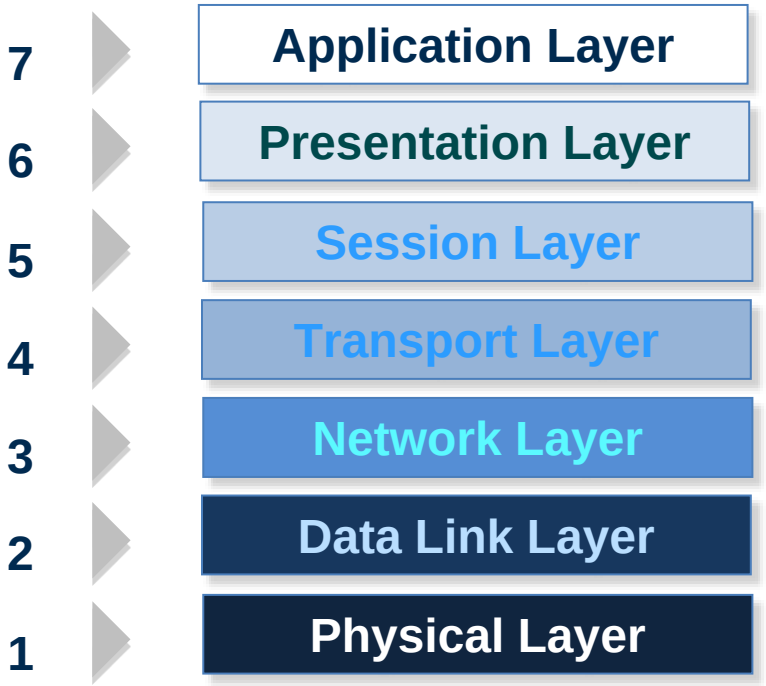
Generalisierung-2

- **Generalisierung** (generalization) – Zusammenfassung einer Menge von Individuen mit teilweise gemeinsamen Attributen durch ein übergeordnetes Individuum, welches nur die gemeinsamen Attribute aufweist.

- Die Bildung von Oberbegriffen zu Begriffen im menschlichen Denken ist eine Generalisierungsabstraktion
- Die Generalisierung erlaubt die systematische Modellierung ähnlicher Dinge (solcher mit teilweise gemeinsamen und teilweise unterschiedlichen Merkmalen)
- Die Bildung von Klassenhierarchien in Klassen- und Objektmodellen basiert auf der Generalisierung
- Generalisierung ist auch in Datenmodellen möglich

Benutzung und Schichtenbildung

Komplexe Gebilde durch Delegieren von Aufgaben und Anordnung der Aufgaben in Schichten übersichtlich und anschaulich modellieren

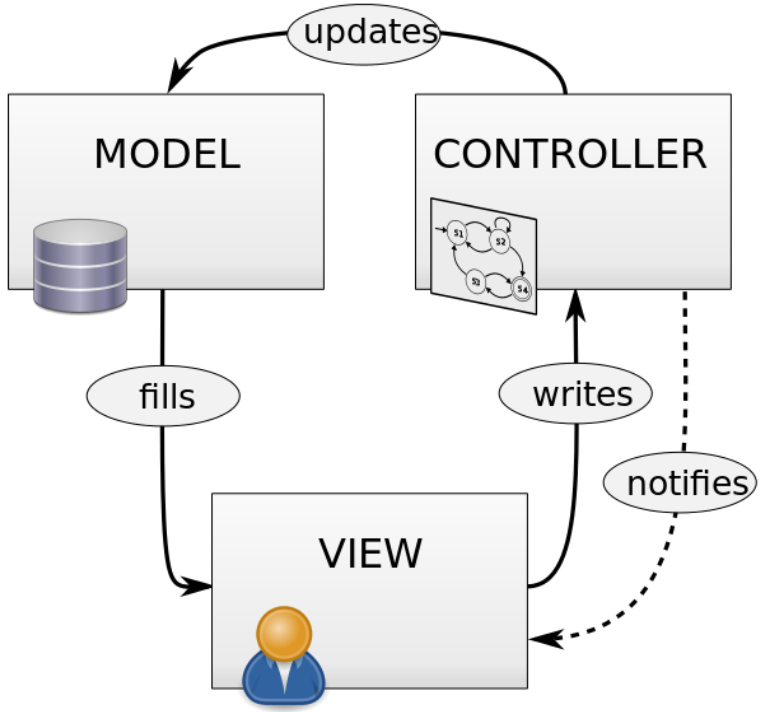


ISO/OSI Schichtenarchitektur

Präsentation

„Business“ Logik

Daten



https://commons.wikimedia.org/wiki/File:MVC_Diagram_%28Model-View-Controller%29.svg

Benutzung und Schichtenbildung-2

- **Benutzung** (delegation, usage) – Verwendung von Leistungen eines Individuums (oder einer Menge von Individuen) durch ein anderes Individuum zum Zweck der Erbringung eigener Leistungen.

- Bündelung von Leistungen zu höherwertigen, komplexen Leistungen; Verwender der komplexen Leistung sehen/kennen die dabei benutzten Leistungen nicht mehr
- Erlaubt die Bildung von Schichten, deren Komponenten
  - sich einerseits auf Leistungen tieferer Schichten abstützen,
  - andererseits Leistungen für höhere Schichten anbieten
  - Metapher der virtuellen Maschinen
  - In der Informatik von Dijkstra (1968) erstmals systematisch verwendet

Benutzung und Schichtenbildung-3

- Benutzer der Leistungen einer Schicht müssen nur wissen, was diese Leistungen sind, nicht aber, wie das Modell im Inneren der leistungserbringenden Schicht und in allen darunterliegenden Schichten aussieht
- Geheimnisprinzip (information hiding, Parnas 1972)
- Verstehen eines Modells, ohne alle Details zu kennen
- Trennung der Belange („Separation of concerns“). Entwickler separater Schichten können sich auf die bestmögliche Ausführung konzentrieren.

Abstraktion in der Informatik

Beispiele für die Anwendung von diesen Abstraktionsprinzipien in dieser Vorlesung:

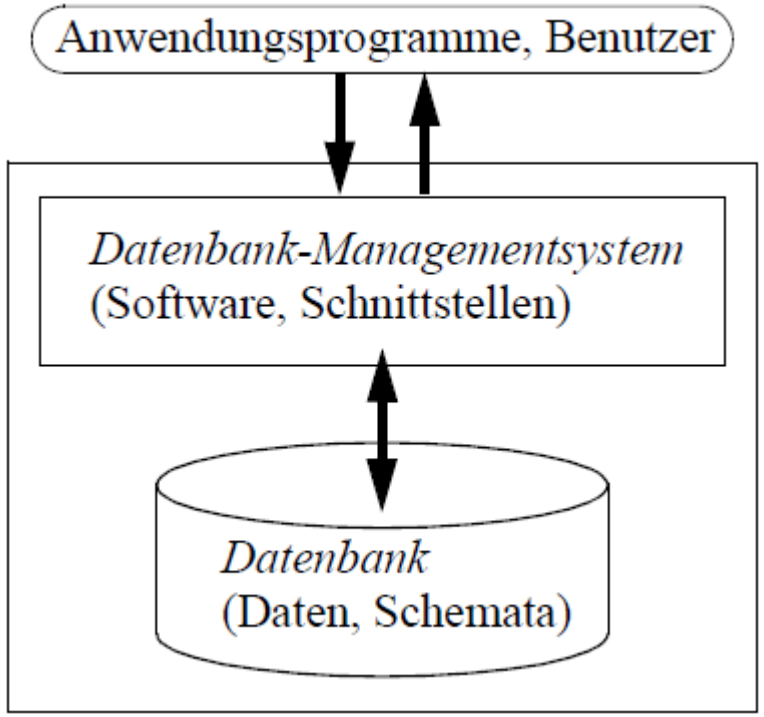
- Architektur von Datenbanksystemen
- Entity Relationship Modellierung
- Relationale Algebra

Literatur

- Dijkstra E.W. (1968). The Structure of the THE multiprogramming System. *Communications of the ACM* **11**, 5 (May 1968). 341-346.
- Ghezzi, C., M. Jazayeri, D. Mandrioli (1991). *Fundamentals of Software Engineering*. Englewood Cliffs: Prentice-Hall.
- Joos, S., S. Berner, M. Arnold, M. Glinz (1997). Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen. *Softwaretechnik-Trends* **17**, 1 (Feb 1997). 29-37.
- Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**, 12 (Dec. 1972). 1053-1058.

Grundbegriffe

- Ein **Datenbanksystem** (DBS) besteht aus:
  - **Datenbank-Managementsystem** (DBMS)
    - Programmsystem zur Verwaltung der Datenbank (Änderungen, Zugriffskontrolle)
  - **Datenbank** (DB)
    - Dauerhaft gespeicherte Sammlung aller Daten und der zugehörigen Beschreibungen



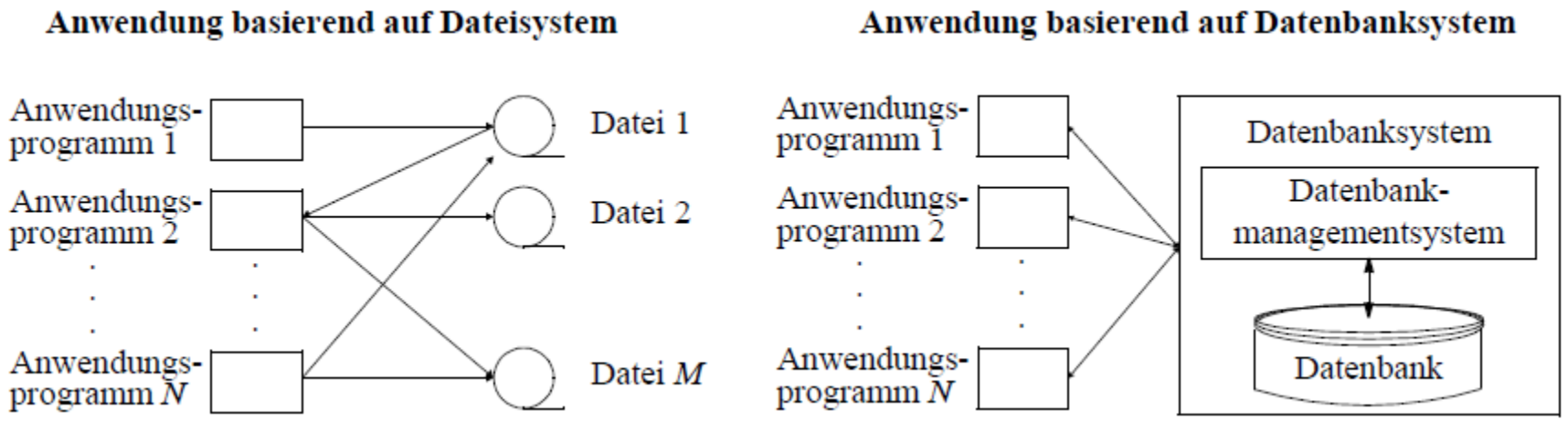
1. Einführung

1. Historische Aspekte und Grundbegriffe
2. **Abgrenzung zu Dateisystemen**
3. Inhalte von Datenbanken
4. Architektur von Datenbanksystemen

Abgrenzung zu Dateisystemen

- Datenbanksysteme ermöglichen die Verwaltung großer Datenbestände in einem einheitlichen System anstatt in einer Vielzahl konventionellen Dateien

- Einsatz von Dateiverwaltungssystemen und Datenbanksystemen:



Charakteristika von Datenbanken: Datenunabhängigkeit

- Datenbank-Managementsystem entkoppelt Daten von Speicher- und Zugriffsstrategien

- **Logische Datenunabhängigkeit**
  - Die logische Unabhängigkeit von Programmen und Daten bewirkt eine Immunität von Anwendungsprogrammen bzgl. Änderung der logischen Gesamtsicht der Daten, wie sie z.B. bei der Einführung neuer Attribute in Datensätzen und neuer Beziehungen zwischen Objekten auftreten.

- **Physische Datenunabhängigkeit**
  - Die physische Unabhängigkeit von Programmen und Daten führt zu einer Immunität von Anwendungsprogrammen bzgl. Änderung der Datendarstellung und Datenspeicherung, wie sie z.B. beim Einsatz effizienter Indexstrukturen auftreten.

Weitere Charakteristika von Datenbanken

- **Verminderte Redundanz**
  - Jedes Datum wird nur einmal in der Datenbank gespeichert. Kopien der Daten werden nur aus internen Sicherheits- oder Effizienzgründen gehalten (*kontrollierte Redundanz*). Dies spart Speicherplatz und erleichtert die Beachtung von *Integritätsbedingungen* in der Datenbank.

- **Einhaltung der Datenintegrität**
  - Die zentralisierte Kontrolle der Daten erlaubt eine einfachere Überprüfung der Daten auf Korrektheit und Vollständigkeit (*Datenintegrität*).

- **Verbesserter Schutz der Daten**
  - Der einheitliche und kontrollierte Zugang aller Anwender zur Datenbank erleichtert die Durchführung von Datenschutz- und Datensicherheitsmaßnahmen.

- **Erleichterung von Standardisierungen**
  - Ein klar organisierter Zugriff auf die Daten erleichtert die Durchsetzung von internationalen Standards bei der Datenmodellierung und bei den Anfragesprachen.

1. Einführung

1. Historische Aspekte und Grundbegriffe
2. Abgrenzung zu Dateisystemen
3. **Inhalte von Datenbanken**
4. Architektur von Datenbanksystemen

Inhalte von Datenbanken: Zwei Ebenen

- **Datenbankschema** (intensionale Ebene)
  - beschreibt *möglichen* Inhalt einer Datenbank
  - Struktur- und Typeninformation über die Daten (“Metadaten”)
  - Art der Beschreibung wird durch das *Datenmodell* vorgegeben
  - Änderungen sind typischerweise eher selten (“Schemaevolution”); einschneidende Schemaänderungen ziehen trotz logischer Datenunabhängigkeit aufwendige Änderungen des Datenbankinhalts nach sich (“Migration”)

- **Ausprägung einer Datenbank** (extensionale Ebene)
  - *tatsächlicher* Inhalt der Datenbank (Datenbankzustand)
  - Objektinformationen, Attributwerte
  - Struktur der Daten ist durch das Datenbankschema vorgegeben
  - Änderungen können sehr häufig auftreten (z.B. Flugbuchungssystem, Kontoführung, …)

- Eine Datenbank muss sowohl Informationen zum Datenbankzustand (Extension) als auch zum Schema (Metadaten) verwalten

Datenmodelle

- **Datenmodelle** sind Formalismen zur Beschreibung aller in der Datenbank enthaltenen Objekte und ihrer Beziehungen untereinander (Datenbankschema), wie sie auf der konzeptionellen und externen Ebene benötigt werden.

- Datenmodelle decken folgende Kernaspekte in unterschiedlicher Weise ab:

  - **Strukturen**
    - Objekttypen und Beziehungen zwischen Objekten

  - **Operationen**
    - Extraktion und Verknüpfung von Daten
    - Anfragesprachen: SQL, OQL, XQuery, SPARQL

  - **Integritätsbedingungen**
    - Modellinhärente: Schlüsseleigenschaften, Typsicherheit, etc.
    - Benutzerdefinierte: Beschränkte Wertebereiche, etc.

Relationales Daten(bank)modell

- Das **relationale Modell** basiert auf dem Strukturierungsprinzip “*Mengen*” (Tabellen, Relationen)
- Vorgeschlagen von Edgar F. Codd (1970)
- IBM System R: erste kommerzielle Implementierung (1980)
- Sehr hohe praktische Bedeutung, sehr viele Implementierungen, Industriestandard
- Steht in dieser Vorlesung im Vordergrund

Relation “Mitarbeiter”

Personal-nummer	Name	Gehalt
1427	Meier	3217,33
8219	Schmidt	1425,87
2624	Müller	2438,21
⋮	⋮	⋮

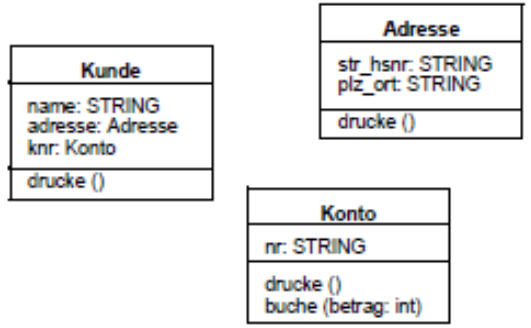
Relation “Mitarbeiter in Abteilung”

Personalnummer	Abteilung
1427	3-1
8219	2-2
2624	3-1
⋮	⋮

Objektorientiertes Daten(bank)modell

- Im **objektorientierten Modell** wird das Konzept “allgemeiner Graph” aufgegriffen
- Zu den zentralen Konzepten *Objekte*, *Attribute* und *Beziehungen* gibt es im objektorientierten Modell noch *Methoden* zur Beschreibung des Verhaltens von Objekten:
  - *Objekttypen* vollständige Beschreibungen gleichartiger Objekte
  - *Objektidentität* dient zur Unterscheidung einzelner Objekte (vs. Werteidentität)
  - *Attribute* speichern den Zustand eines Objektes (statischer Aspekt)
  - *Methoden* beschreiben das Verhalten von Objekten (dynamischer Aspekt)
  - *Beziehungen* werden im OO-Modell weiter unterschieden

- Gute Anbindung an die objektorientierte Programmierung
  - Modellierung durch UML-Klassendiagramme
- Hat sich dennoch in der Praxis nicht stark durchgesetzt
  - Schwierige Optimierbarkeit wg. prozedural geprägter Elemente



Weitere Daten(bank)modelle

- **Objektrelationales Datenmodell**
  - Erweiterung des relationalen Datenmodells um objektorientierte Konzepte
  - *Vorteil*: Investitionsschutz für bestehende relationale Systeme (bei Hersteller und Kunden)
  - *Nachteil*: Komplexität, zum Teil noch Performanzprobleme

- **eXtended Markup Language (XML)**
  - Ursprünglich als reines Daten- und Dokumentaustauschformat zwischen Anwendungen gedacht
  - Für Datenbanken um Anfragesprachen erweitert (XPath, XQuery, XSLT)

- **NoSQL oder Graph-Datenmodelle**
  - Keine festen Schemainformationen
  - Key-Value-Stores, Graphdatenbanken
  - Beispiele: JSON, CouchDB, MongoDB, TripleStores
  - Vgl. Vorlesung „Semantic Web“ (Decker)

Datenmodelle: Allgemeine Unterscheidung

- **Strukturierte Datenmodelle**
  - Relationales Modell: starre Tabellenstruktur mit Zeilen (= Tupel) und Spalten (=Attribute)
  - Objektorientiertes Modell: Objekte mit Attributen und Methoden

- **Semistrukturierte Datenmodelle**
  - XML, etc.: Strukturelemente (Tags, z.T. mit Attributen) in zum Teil beliebig freier Zusammensetzung, auch unstrukturierte Inhalte (zwischen den Tags)
  - Resource Description Framework (RDF): gelabelte Knoten und Kanten. Dabei sind Labels entweder URIs oder Literale (Text).

- **Unstrukturierte Datenmodelle**
  - BLOBs (binary large objects) in relationalen und objektrelationalen Datenbanken:
    - interne Struktur für Datenbanksystem nicht erkennbar
    - definiert durch Anwendungen (Bsp. JPEG)
  - Key-Value-Stores, NoSQL: Beliebig zusammengesetzte Datenelemente

1. Einführung

1. Historische Aspekte und Grundbegriffe
2. Abgrenzung zu Dateisystemen
3. Inhalte von Datenbanken
4. Architektur von Datenbanksystemen

1. Einführung

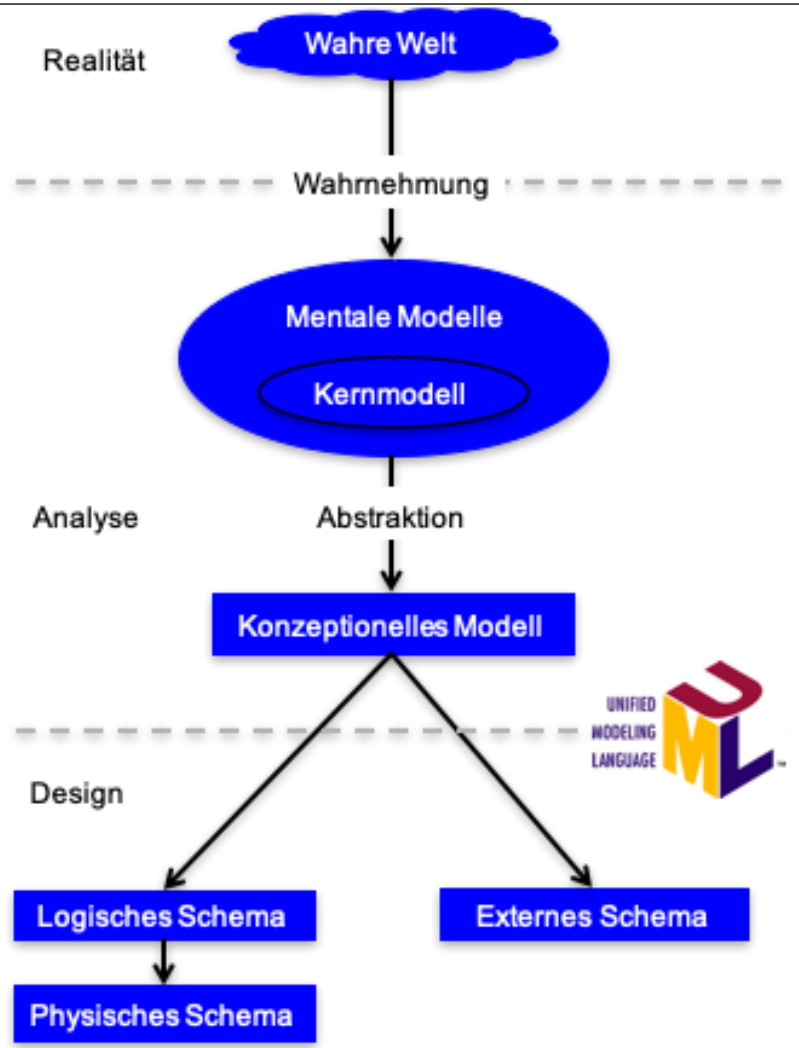
1. Historische Aspekte und Grundbegriffe
2. Abgrenzung zu Dateisystemen
3. Inhalte von Datenbanken
4. **Architektur von Datenbanksystemen**

Von der Realität zu Design-Schemata

Die Erstellung eines konzeptionellen Modells erfordert einen Einigungsprozess der Beteiligten Stakeholder auf Basis einer gemeinsamen Vorstellung der Realität

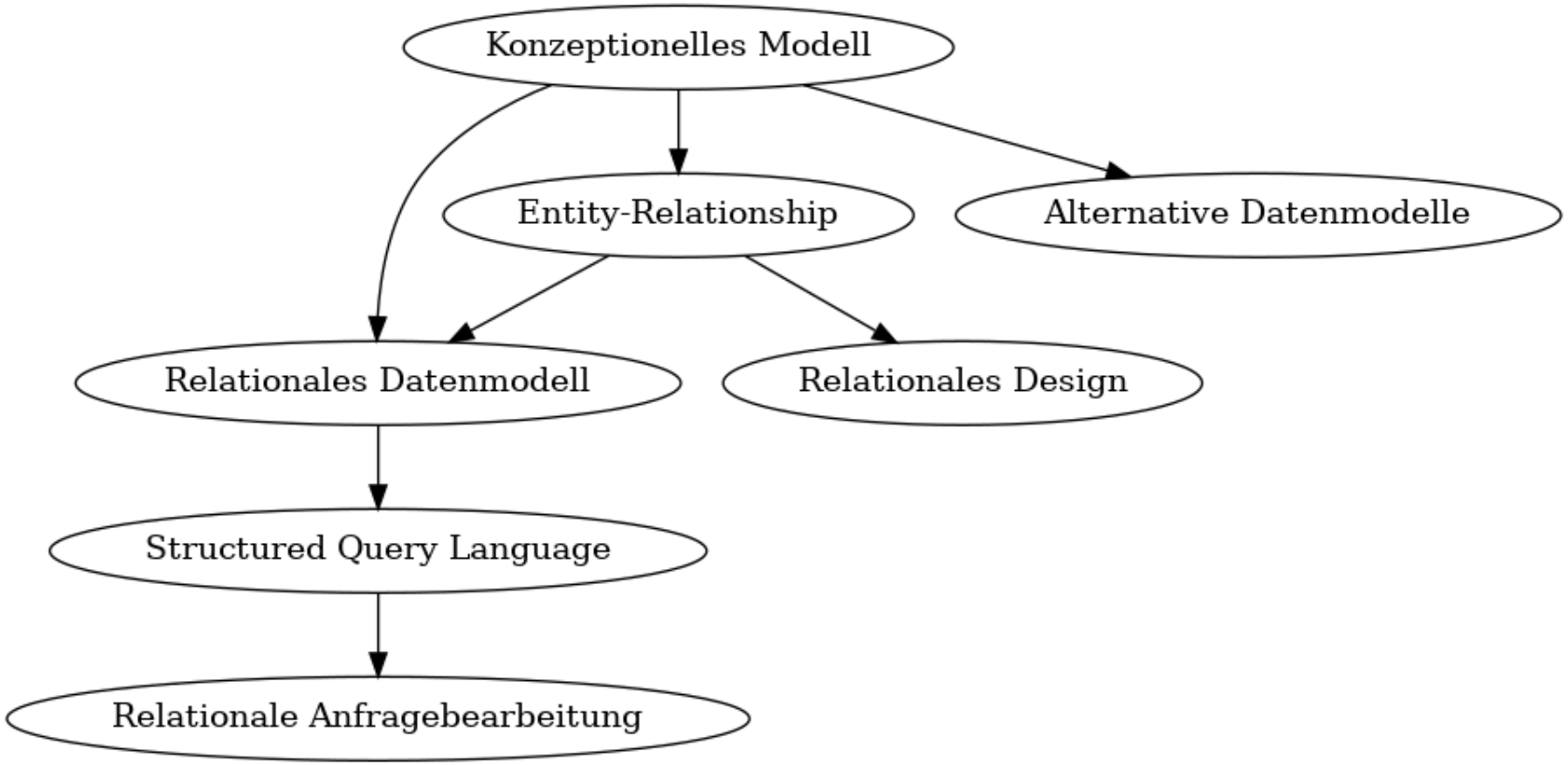
Die Gestaltung der Schemata erfolgt auf Basis funktionaler und nicht-funktionaler Anforderungen sowie von Randbedingungen.

Der Lebenszyklus dieser Schemata ist oft langlebig und nicht an einzelne Softwareentwicklungsprojekte gebunden. Standardisierungsbemühungen können sehr aufwändig sein und hohe Kosten verursachen - Beispiel: ERP Systeme (SAP).



Vom Konzeptuellem Modell zum Relationalem Modell

Verweise auf Kapitel dieser Vorlesung



Datenmodelle, Datenbankmodelle, Datenstrukturen

- **Konzeptuelle Modelle**

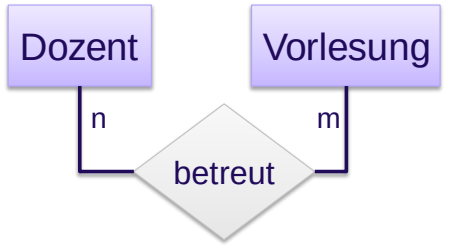
  Für Entwurfsphase
  - Entity-Relationship-Modell
  - Unified Modeling Language (UML)
  - „Ontologien“ (Wissensrepräsentation)

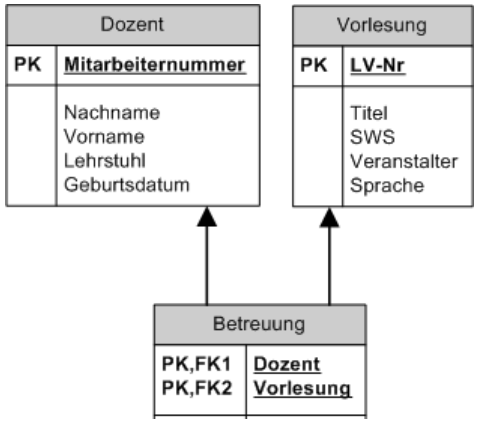
- **Logische Datenbankschema**

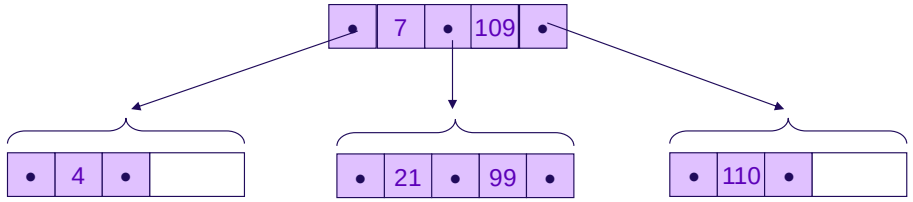
  Für Gesamtsicht und Benutzersichten
  - Relationales Datenbankmodell
  - Objektorientiertes Datenbankmodell
  - Semi-strukturiertes XML-Datenmodell

- **Physische Datenstrukturen**

  Für interne Datenspeicherung
  - B*-Bäume, Hashing, verkettete Listen, …



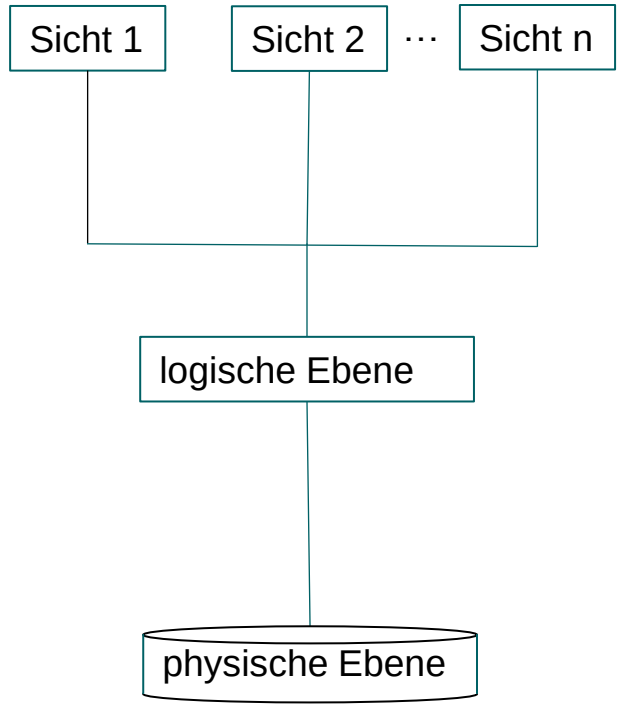




Abstraktionsebenen eines Datenbanksystems: 3-Ebenen Modell nach ANSI/SPARC (1975)

Zur Realisierung der physischen und logischen Datenunabhängigkeit sind Datenbanksysteme typischerweise in drei verschiedenen Abstraktionsebenen aufgebaut

- **Externe Ebene, Benutzersichten**
  - Unterschiedliche Sichten von Benutzern / Benutzergruppen auf den Datenbestand
  - Betrachtung einer Teilmenge der Daten, die für eine Anwendung benötigt wird (z.B. alle Studierenden eines Studiengangs)
- **Konzeptionelle Ebene, Logische Ebene**
  - Logische Gesamtsicht aller Daten; Abstraktion zwischen externer und interner Ebene
  - Logische Organisation der Daten, z.B. in Tabellen
- **Interne Ebene, Physische Ebene**
  - Festlegungen zur physischen Datenorganisation; Zugriffsalgorithmen
  - Ablegen, Verwalten und Finden der Daten auf einem Speichermedium (meist: Festplatte)



Konzeptuelle Ebene funktioniert als Abstraktion zwischen externer und interner Ebene

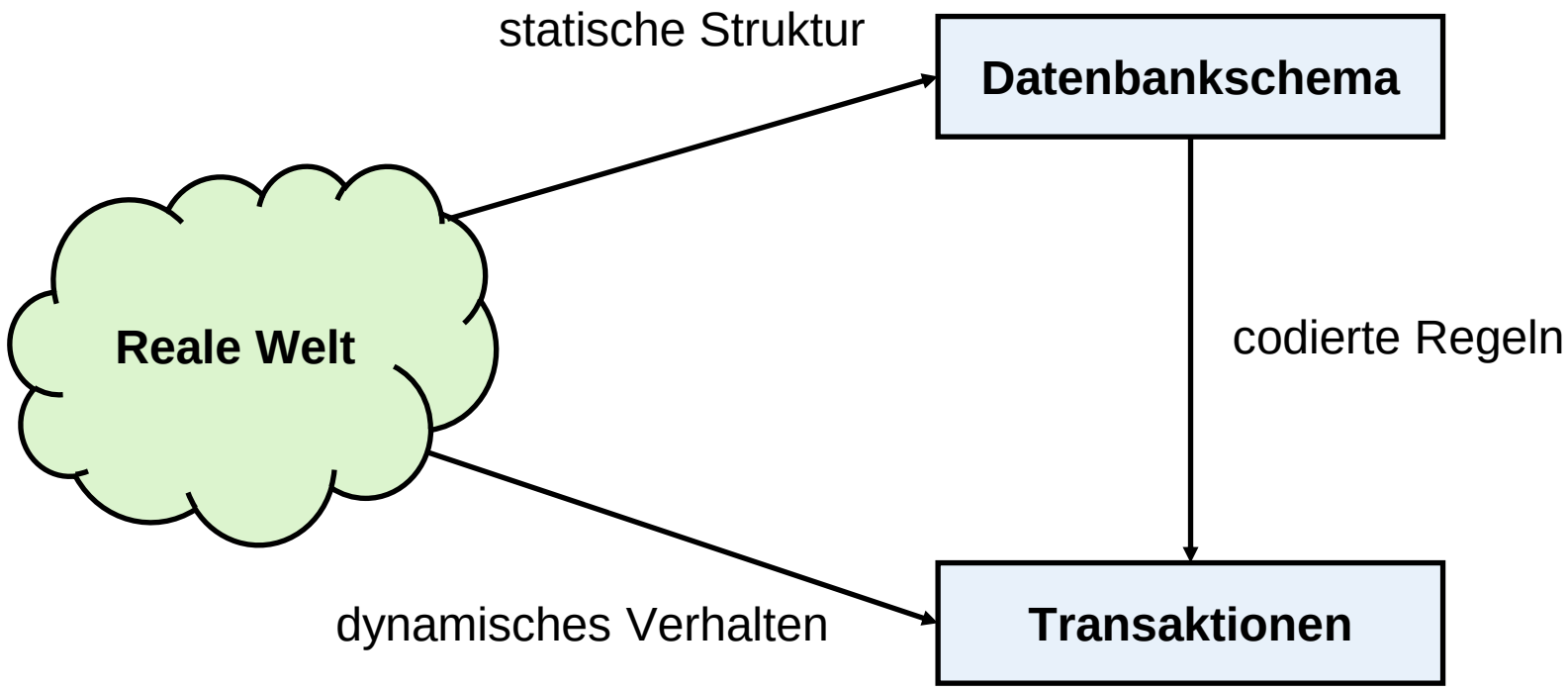
2. Das Entity-Relationship-Modell

1. Der Datenbankentwurf
2. Entity-Relationship-Modell
3. Konzeptueller Entwurf

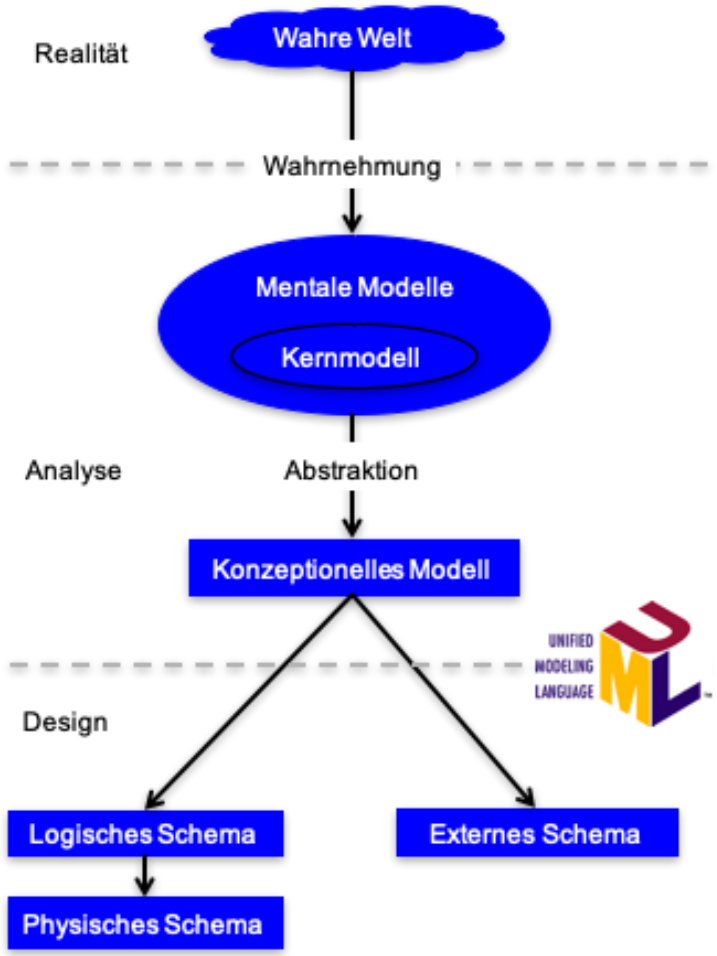
2. Das Entity-Relationship-Modell

1. **Der Datenbankentwurf**
2. Entity-Relationship-Modell
3. Konzeptueller Entwurf

Modellierung von Datenbanken



Modellierung von Datenbanken

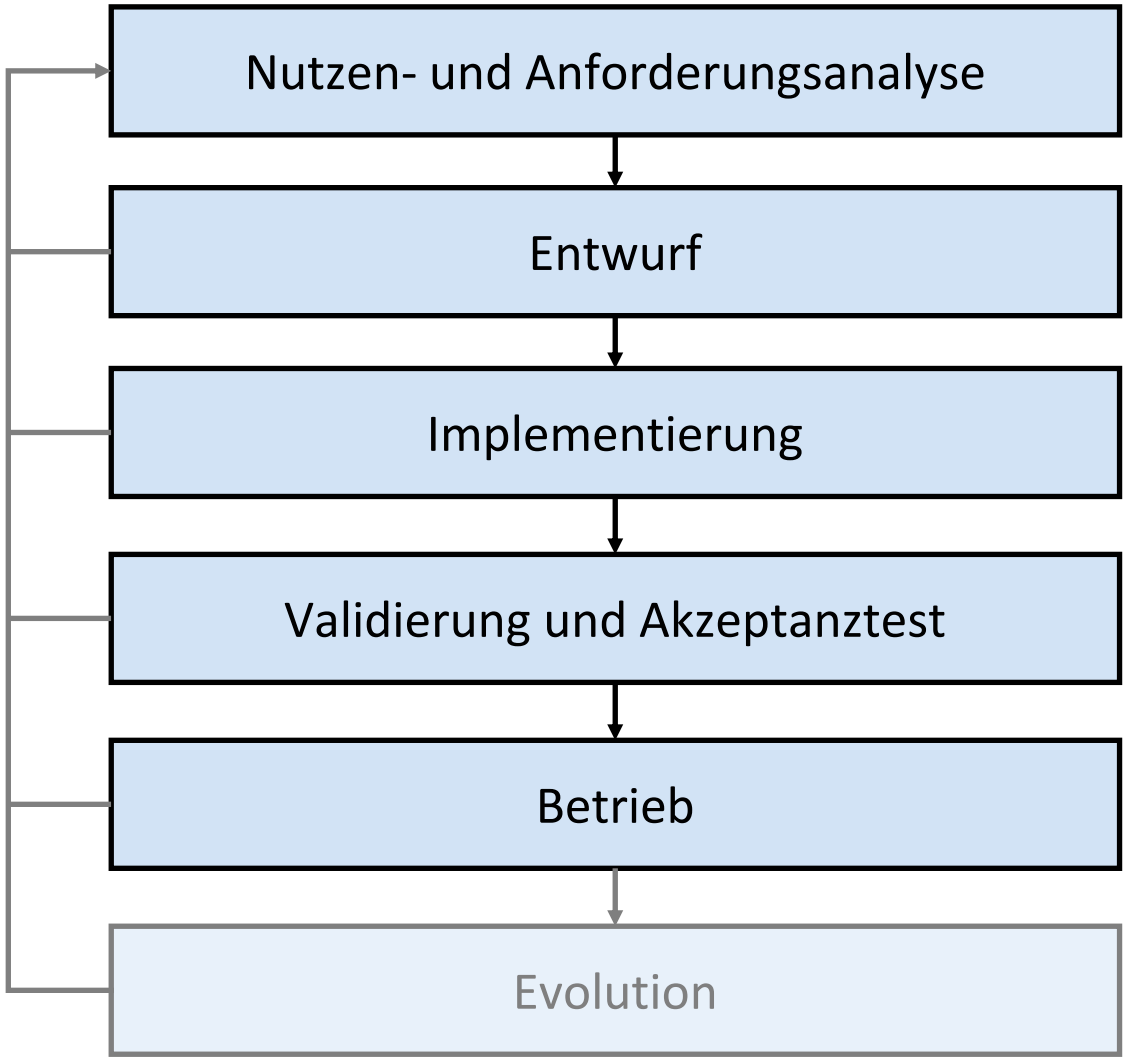


Datenbankentwurf

- **Definition: *Datenbankentwurf*** *[Vossen]*
  - „Die Aufgabe des Datenbankentwurfs ist der Entwurf der *logischen* (*konzeptuellen*) und *physischen Struktur* einer Datenbank so, dass die *Informationsbedürfnisse* der Benutzer in einer Organisation für *bestimmte Anwendungen adäquat* befriedigt werden können.“

- **Datenbankentwurf**
  1. Entwurf der logischen (konzeptuellen) Struktur der Datenbank.
  2. Entwurf der physischen Struktur der Datenbank.
  3. Angemessen für bestimmte, wohl umrissene Anwendungen der Datenbank

Informationssystem-Lebenszyklus



Datenbankentwurf

- **Nutzenanalyse**
  - Ermittlung potentieller Anwendungsgebiete des Systems
  - Kosten-Nutzen-Rechnungen für die Anwendungen
  - Es werden Prioritäten für die Anwendungsbereiche gesetzt

- **Anforderungsanalyse**
  - Wichtigste Phase
    - Beeinflusst alle anderen Phasen
    - läuft parallel zur Nutzenanalyse
  - Im Allgemeinen in *Zusammenarbeit mit potentiellen Benutzern*, um Probleme und Anwendungen umfassend verstehen zu können
  - Anforderungen der gewählten Anwendungen werden ermittelt
    - Informationsanforderungen
    - Bearbeitungsanforderungen
      - Möglichkeiten (quantitativ, qualitativ)
    - Benutzeranforderungen
      - Gute Schnittstelle, Ähnlichkeit zu anderen bekannten (Papier-)Formaten, mögliche/unterstützte Operationen, Dokumentation, Erklärungs-/Hilfskomponenten

Datenbankentwurf

- **Entwurf**
  - Entwurf der Anwendungen, die die Datenbank verwenden sollen
  - Entwurf der Datenbank als zentrales Informations-Reservoir
  - Berücksichtigung von physikalischen Randbedingungen wie Latenz, Bandbreite
  - Konzeptueller Entwurf
    - Unabhängig von Zieldatenmodell
  - Logischer Entwurf
    - Übersetze Konzepte in konkretes Datenbankschema

- **Implementierung**
  - Implementierung anhand des Entwurfs
  - Transformierung der Modelle in physische Strukturen
    - Datenstrukturen, Zugriffsstrukturen

Datenbankentwurf

- **Validierung und Akzeptanztest**
  - Schlüsselkriterium für den Erfolg
  - Validierung der Implementierung anhand der Anforderungsanalyse
    - Das System muss die Benutzer-Anforderungen erfüllen (funktionale Eigenschaften)
    - Das System muss gegebene Performance-Vorgaben erfüllen (nicht-funktionale Eigenschaften)

- **Betrieb**
  - Das System kann in Betrieb genommen werden
  - Eventuelle Konvertierung alter Anwendungen auf das neue System
  - Überwachung und Verbesserung der System-Performance

- **Evolution**
  - Fortentwicklung des Systems
  - Rückgriffe auf frühere Phasen können notwendig werden

Qualitätskriterien des Datenbankentwurf

- **Korrektheit**
  - Korrekte Verwendung der Konzepte des Datenmodells
    - Sprache zur Schemabeschreibung wird *syntaktisch* korrekt verwendet
    - Beim Entity-Relationship-Modell kann zwischen *syntaktischer* und *semantischer* Korrektheit unterschieden werden

- **Vollständigkeit**
  - Ein Datenbankenwurf bzw. ein konzeptionelles Datenbankschema soll vollständig sein
    - Alle relevanten Aspekte und Eigenschaften des Anwendungsbereichs sind erfasst

- **Minimalität**
  - Jeder Aspekt der Anforderungen sollte lediglich einmal im Schema vorkommen
  - Redundanzvermeidung
    - Gewünschte und unvermeidbare Redundanzen werden dokumentiert

Qualitätskriterien des Datenbankentwurf

- **Lesbarkeit**
  - Datenbankschema sollte selbsterklärend sein, d.h. wichtige Elemente und Konzepte sollten durch das Schema verstanden werden können
  - Zusätzlich sollte es eine umfassende Dokumentation geben

- **Modifizierbarkeit/Anpassbarkeit**
  - Anpassung an neue/andere Anforderungen
  - Modifikation der Datenbank ist vergleichsweise leicht wenn die Datenbank modular aufgebaut ist und gut dokumentiert wurde

- **Normalisierung**
  - Datenbankschema sollte gewissen Normen genügen
  - Erstellung einer „übersichtlichen“ Struktur und Vermeidung von Redundanzen

2. Das Entity-Relationship-Modell

1. Der Datenbankentwurf
2. **Entity-Relationship-Modell**
3. Konzeptueller Entwurf

Entity-Relationship-Modell

- Das Entity-Relationship-Modell dient dazu, für einen zu modellierenden Ausschnitt der realen Welt ein konzeptionelles Schema zu erstellen und auf hohem Abstraktionsniveau graphisch darzustellen.

- Das Ergebnis ist ein *Entity-Relationship-Diagramm (ER Diagramm oder ER-Modell)*.
  - Das ER-Modell ist ein abstraktes, maschinenfernes Datenmodell
  - Überlegungen zur Effizienz und zur physischen Umsetzung spielen noch keine Rolle.

- Für die konkrete Implementierung eines Datenbankschemas muss das ER-Diagramm auf eines der maschinennäheren Modelle abgebildet werden (z.B. hierarchisches Modell, Netzwerkmodell, relationales Modell).
  - Für eine rudimentäre Transformation können einfache Regeln angegeben werden
  - Die Gewinnung eines *effizienten* Schemas erfordert ein tieferes Verständnis des Zielmodells

Schemaentwurf

- Die generelle Aufgabe des Schemaentwurfs besteht darin, für den zu modellierenden Teil der “realen Welt” eine formale Beschreibung zu entwickeln. Dabei gibt es verschiedene Zwischenstufen:
  - Beschreibung durch natürliche Sprache *(z.B. Pflichtenheft, …)*
  - Beschreibung durch abstrakte graphische Darstellungen *(z.B. ER-Diagramm, …)*
  - Beschreibung durch konkrete, implementierbare Sprache *(z.B. im Relationalen Modell)*

- *Beispielanwendung:* Lehrbetrieb an der Fachgruppe Informatik
- Folgende Objekte spielen eine Rolle:
  - *Personen (genauer: Studierende, Professor/Professorinnen, Mitarbeiter/Mitarbeiterinnen, ...), Räume, Zeiten, Lehrveranstaltungen, Bücher, usw.*

- Diese Objekte der realen Welt sind nie voneinander isoliert, vielmehr stehen sie in verschiedenen Beziehungen zueinander, z.B.:
  - *veranstaltet, nimmt teil an, findet statt in, wohnt in*

Entity-Relationship-Diagramme

- Das Entity-Relationship-Modell (ER-Modell) beschreibt die Welt durch:
  - **Objekttypen** (*Entity-Typen*) mit
  - **Eigenschaften** (*Attributes*) und
  - **Beziehungen** (*Relationships*) zueinander.

- Beim Schemaentwurf entstehen sogenannte *ER-Diagramme*, d.h. graphische Darstellungen des betrachteten Realitätsausschnittes.

- Eine entscheidende Aufgabe beim Entwurf eines Datenbankschemas besteht darin, *geeignete* Objekttypen, Attribute und Beziehungen zu bestimmen.

- Ein *ER-Diagramm* ist die graphische Darstellung eines konkreten Datenbankentwurfs im ER-Modell.
  - Innerhalb von Rechtecke, Ellipsen und Rauten werden im ER-Diagramm die Namen der jeweiligen Entity-Typen, Attribute bzw. Beziehungen notiert.

Elemente des ER-Modells (Entity-Typen)

- **Entities:** Als *Entity* (= “Seiendes”, Objekt) bezeichnet man etwas, das existiert und unterscheidbar ist von anderen Entities. Beispiele für Entities sind *Person, Auto, Kunde, Buch, etc*.
  - Einzelne Entities des gleichen Typs sind unterscheidbar, d.h. sie haben eine *Objektidentität*.

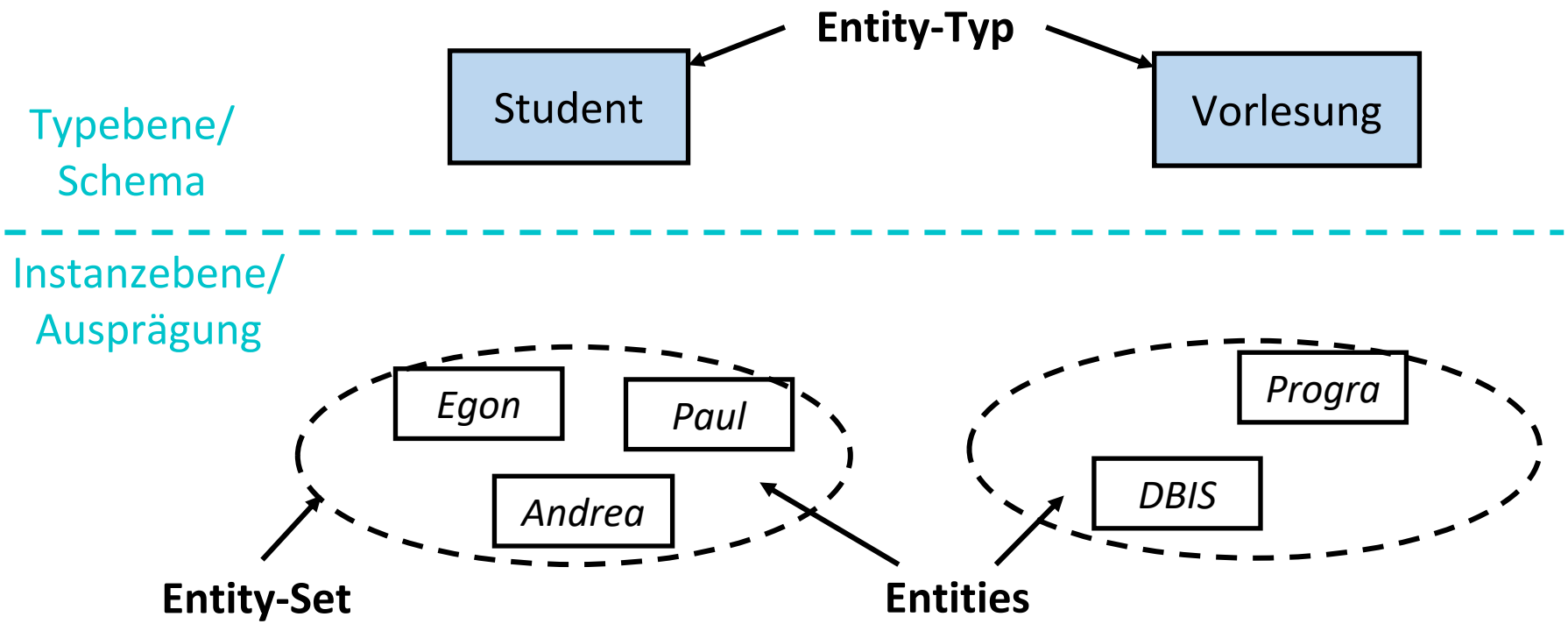
- Diese Eigenschaften zeigen, dass das ER-Modell eine Urform des objektorientierten Modellierens ist.

- **Entity-Typ** (Objekttyp) ist eine Menge von Entities mit den selben *Attributen z.B: Personen, Autos, Kunden, etc*.
  - Entity-Typen werden im ER-Diagramm durch *Rechtecke* dargestellt: 

- **Entity-Set** (Objektmenge) ist eine Menge von Entities eines bestimmten Entity-Typ zu einem Zeitpunkt. Ein Entity-Set wird normalerweise mit dem gleichen Namen wie der Entity-Typ referenziert.

Elemente des ER-Modells (Entity-Typen)

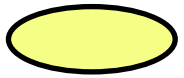
Beispiel:



Elemente des ER-Modells (Attribute und Schlüssel)

- **Attribute**
- Alle Mitglieder eines Entity-Typs werden durch eine Menge charakterisierender Eigenschaften (*Attribute*) beschrieben. Beispiele dafür sind “Farbe”, “Gewicht” und “Preis” beim Entity-Typ <Teile>. Die Werte eines Attributes stammen normalerweise aus Wertebereichen wie `INTEGER`, `REAL`, `STRING`, etc. Aber auch strukturierte Werte wie Listen, Bäume, usw. sind vorstellbar.

- Attribute werden im ER-Diagramm durch *Ellipsen* dargestellt:



  - Attribute sind über ungerichtete Kanten mit dem zugehörigen Entity-Typ verbunden.
  - Schlüssel-Attribute werden (in der Regel) unterstrichen.
  - Entity-Typen, an die nur ein Attribut gebunden ist, werden oft nur über die Attribut-Ellipse repräsentiert und erhalten den Namen des Attributes

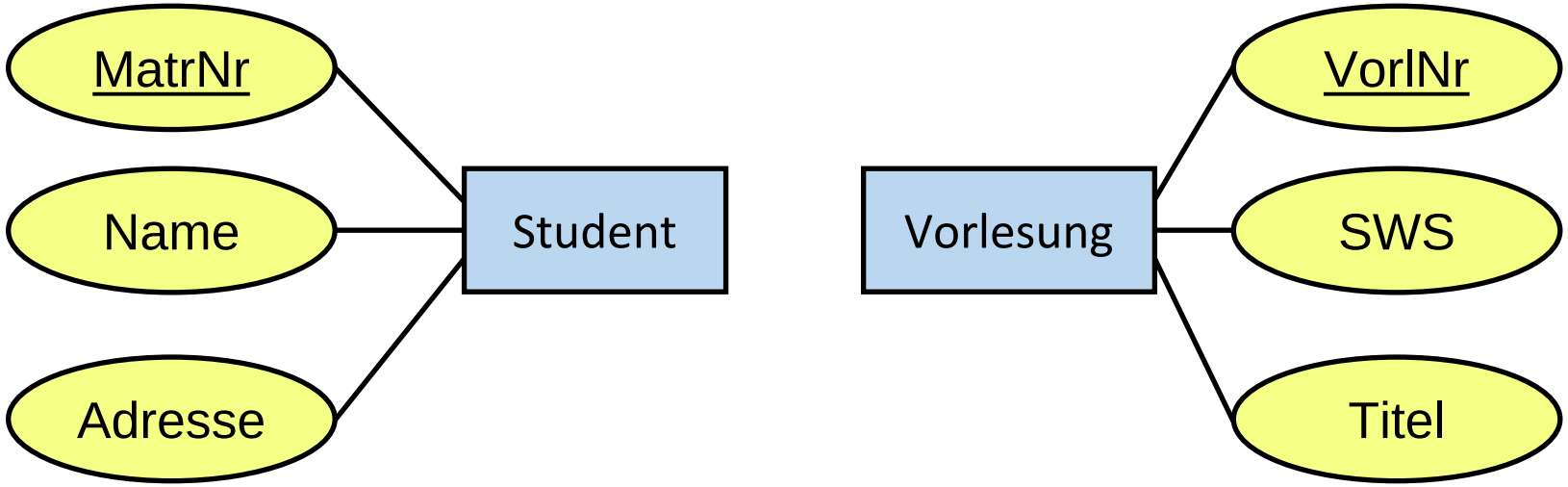
Elemente des ER-Modells (Attribute und Schlüssel)

- **Schlüssel**
- Eine minimale Menge von Attributen, deren Werte das zugeordnete Entity eindeutig innerhalb aller Entities seines Typs identifizieren, nennt man *Schlüssel*.
  - Gibt es verschiedene minimale und identifizierende Attributmengen als Schlüsselkandidaten, dann wählt man einen dieser Kandidaten-Schlüssel als *Primärschlüssel* aus.
  - Häufig wird einem Typ ein zusätzliches Attribute als Schlüssel hinzugefügt (*Kunstschlüssel*), z.B. Personalnummer, Vorlesungsnummer, etc.
  - Attribute, die den Primärschlüssel bilden, werden durch Unterstreichung gekennzeichnet.

- *Beispiele:*
- Typ `Student` mit Attributen (`MatrNr, Name, Adresse`)
- Typ `Vorlesung` mit Attributen (`VorlNr, SWS, Titel`)
- Typ `Leistungsnachweis` mit Attributen (`MatrNr, VorlNr, Note`)

Elemente des ER-Modells (Attribute und Schlüssel)

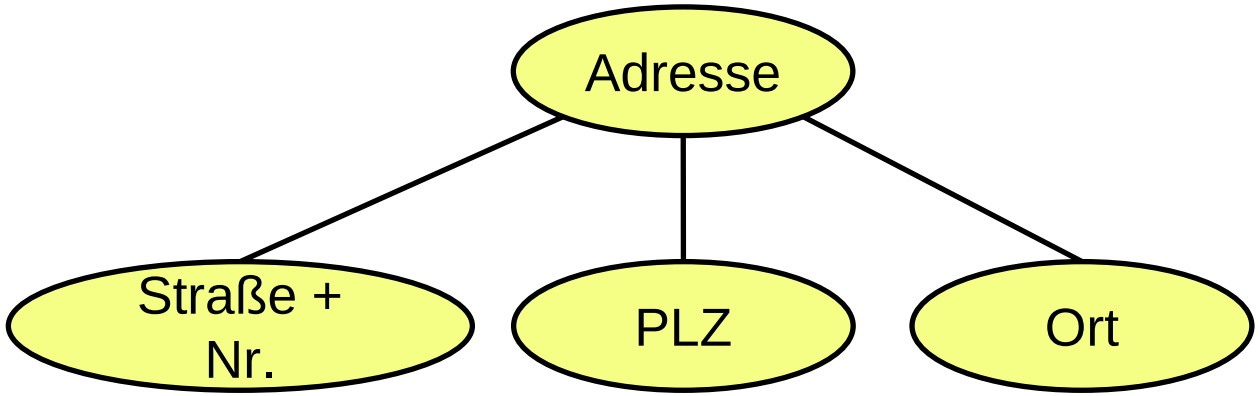
Beispiele:



- Entity-Typ *Student* mit Attributen MartNr (Schlüsselattribut), Name und Adresse
- Entity-Typ *Vorlesung* mit Attributen VorlNr (Schlüsselattribut), SWS und Titel

Elemente des ER-Modells (Zusammengesetzte Attribute)

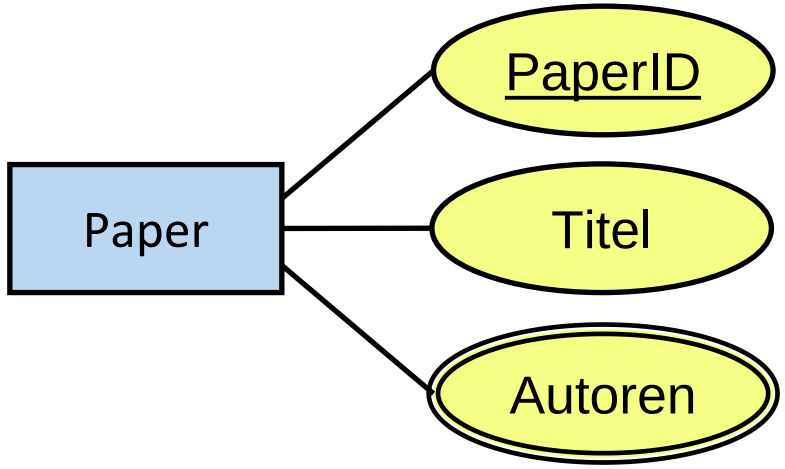
- Ein Attribut kann aus anderen Attributen bestehen
  - Diese Unterattribute werden mit Linien zu dem Attribut verknüpft



- Das Attribut *Adresse* besteht aus den Unterattributen *Straße + Nr.*, *PLZ* und *Ort*

Elemente des ER-Modells (Mehrwertige Attribute)

- Ein (mehrwertiges) Attribut kann eine Menge von Werten enthalten
  - Die bisherigen Attribute waren einwertig
  - Ein mehrwertiges Attribut wird als *Ellipse* mit *doppelter Umrandung* dargestellt



- Ein *Paper* hat eine *PaperID* (Schlüsselattribut) und einen *Titel,* aber kann mehrere *Autoren* haben

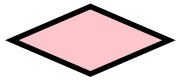
Elemente des ER-Modells (Relationship)

- **Relationships (Beziehungen)**
- Über *Relationships* (Beziehungen) lassen sich Zusammenhänge zwischen Entity-Typen darstellen.

- *Beispiele*:
  - <Vorlesung> *setzt voraus* < Vorlesung >
  - <Vorlesung > *findet statt in* <Raum>

Elemente des ER-Modells (Relationship)

- Beziehungen werden im ER-Diagramm durch *Rauten* dargestellt:



  - Beziehungen werden mit den entsprechenden Entity-Typen durch Kanten verbunden.
  - Eine Beziehung kann auch Attribute haben

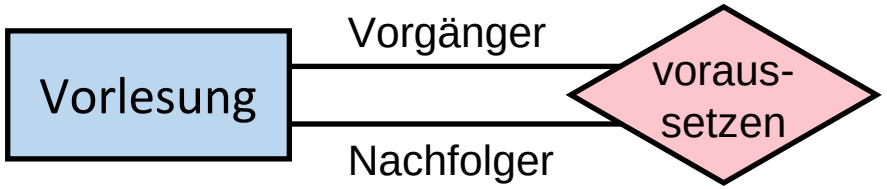
Beispiel:

- *Student* (Entity-Typ) *hört* (Beziehung) *Vorlesung* (Entity-Typ)

Elemente des ER-Modells (Rekursive Beziehung)

- Eine *Rekursive Beziehung* verbindet einen gleichen Entity-Typen mehrfach
  - Tritt ein Entity-Typ in einer Beziehung mehrfach auf, so hat es dort unterschiedliche *Rollen*

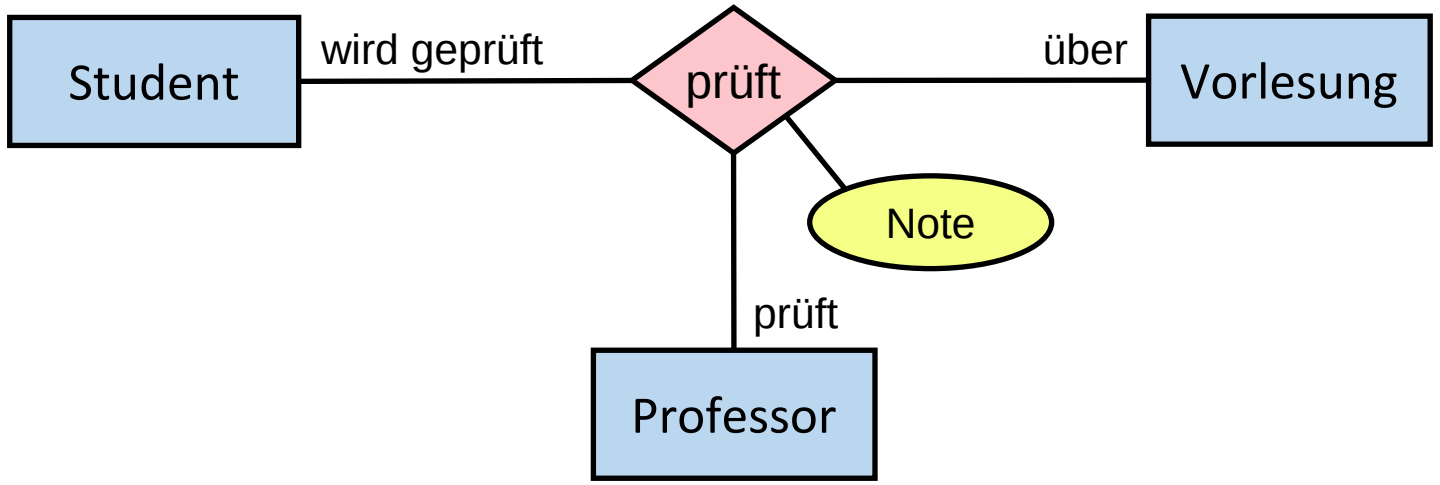
- *Beispiel*: <Vorlesung> *setzt voraus* <Vorlesung>
  - Der Objekttyp *Vorlesung* tritt in den beiden Rollen ‘Vorgänger’ und ‘Nachfolger’ auf.



Elemente des ER-Modells (n-stellige Beziehung)

- Zusätzlich zu zweistelligen Beziehungen gibt es auch mehrstellige Beziehungen
  - Der Spezialfall der mehrstelligen Beziehung, die drei Entity-Typen verbindet nennt man auch *ternäre* Beziehung

- *Beispiel (ternäre Beziehung)*:
- <Student> wird von <Professor> über <Vorlesung> *geprüft* und erhält *Note*



Elemente des ER-Modells (Relationship)

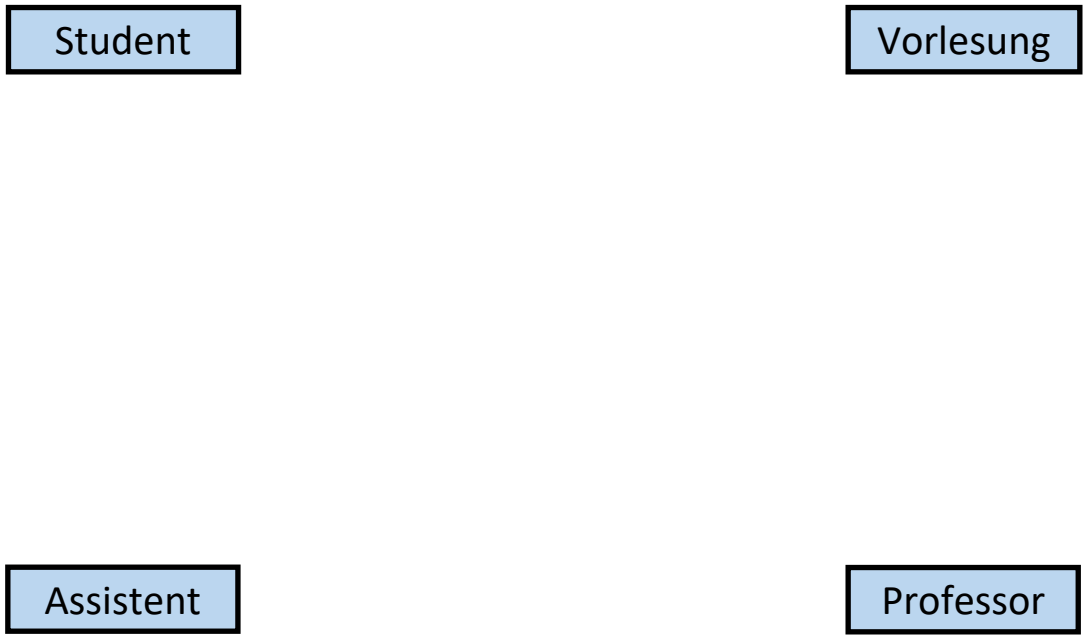
- *Formal:*
  - Eine Beziehung $R(E_1, E_2, \dots, E_n)$ für $n \geq 1$ Entity-Typ E_i kann als Relation im mathematischen Sinne aufgefasst werden, d.h. $R \subseteq E_1 \times E_2 \times \cdots \times E_n$. Ein bestimmter Entity-Typ darf mehrfach vorkommen, es sind zwei- und mehrstellige Beziehungen möglich.
  - Eine Ausprägung einer Beziehung ist eine endliche Menge von n-Tupeln $(e_1, \dots, e_n) \in R$,d.h. $e_i \in E_i$ für $i = 1, \dots, n$.

Elemente des ER-Modells (Ausprägung einer ternären Beziehung)

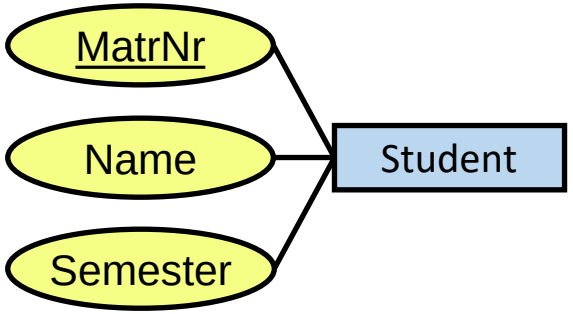
- Beispiel Beziehung:
  <Produkt> *wird geliefert von* <Lieferant> an <Filiale>
- Ternäre Beziehung zwischen <Produkt>, <Lieferant>, <Filiale>

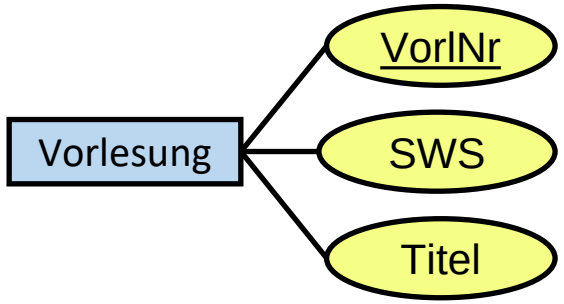
- Ausprägung:
  - *wird_geliefert_von_an* (Milch, Huber, Lehel)
  - *wird_geliefert_von_an* (Milch, Meier, Lehel)
  - *wird_geliefert_von_an* (Milch, Meier, Altstadt)

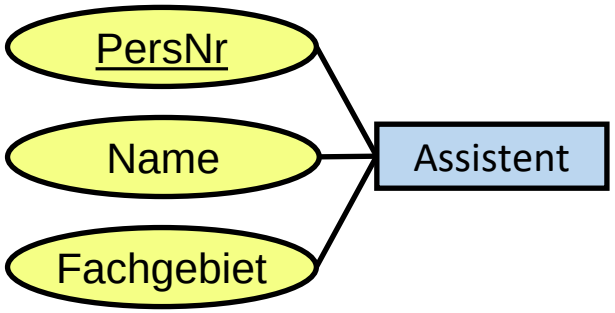
Beispiel: ER-Diagramm (Universität)
Entity Typen

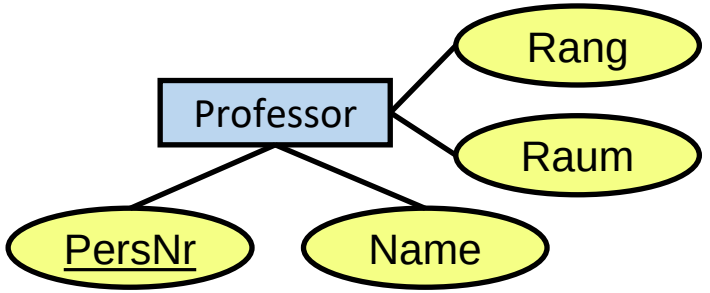


Beispiel: ER-Diagramm (Universität)
Entity Typen + Attribute

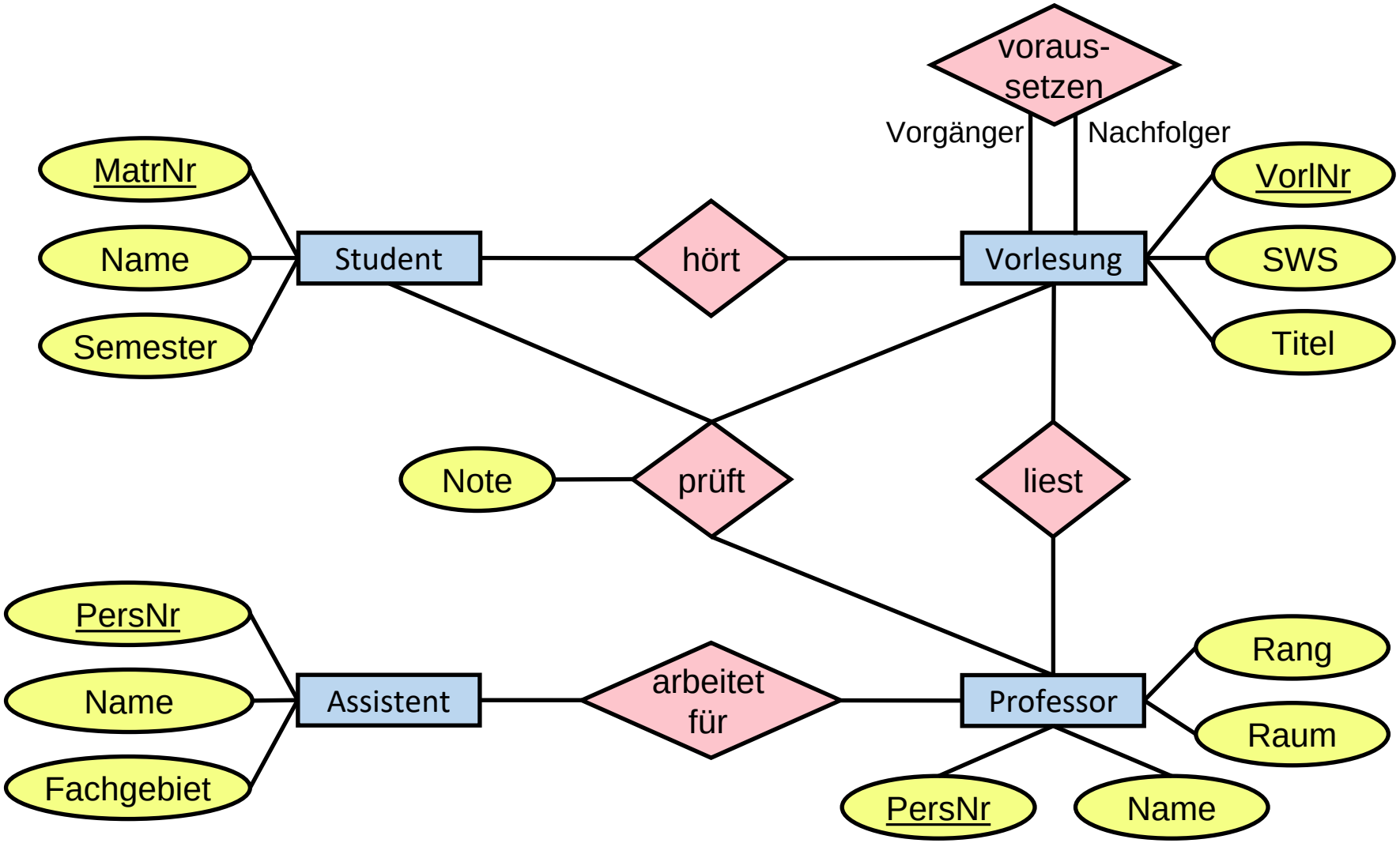








Beispiel: ER-Diagramm (Universität)
Entity Typen + Attribute + Beziehungen



Elemente des ER-Modells (Kardinalitäten)

- Fragestellung: “Wie viele Entitäten können über eine Beziehung einer Entität zugeordnet werden?“

- Beispiele:
  - Wie viele Assistenten-Entities können über die *arbeitet für* Beziehung mit einer Professor-Entity in Beziehung stehen?
  - Wie viele Professoren-Entities kann die *liest* Beziehung einer einzelnen Vorlesungs-Entity zuordnen?

- Die Antwort hängt davon ab, was modelliert wird.
- Am einfachsten mit binären Beziehungen zu modellieren.

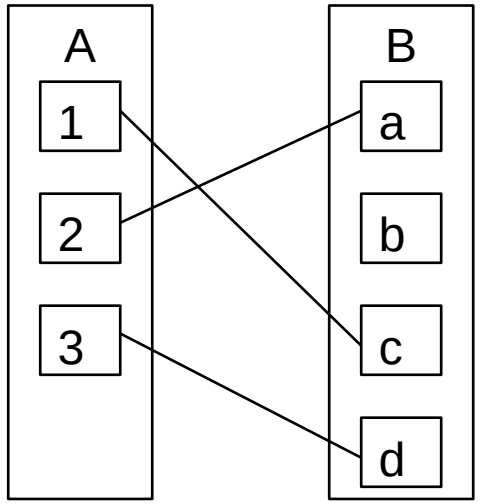
Elemente des ER-Modells (Kardinalitäten)

Gegeben:

- Entity-Typen A and B
- Binäre Beziehung R zwischen A und B

1:1-Beziehung (one-to-one)

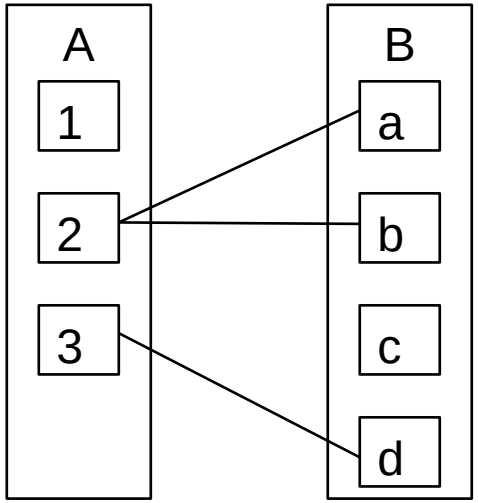
- Jede Entity in A ist höchstens mit einer Entity aus B in Beziehung
- Jede Entity in B ist höchstens mit einer Entity aus A in Beziehung



Elemente des ER-Modells (Kardinalitäten)

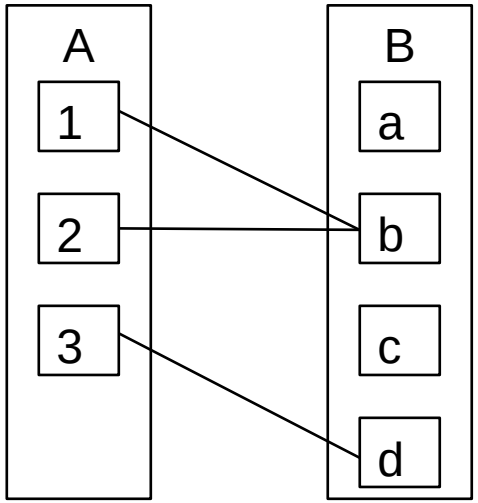
1:m-Beziehung (one-to-many)

- Jede Entity aus A steht mit null oder mehr Entities aus B in Beziehung.
- Jede Entity aus B steht mit höchstens einer Entity aus A in Beziehung



m:1-Beziehung (many-to-one)

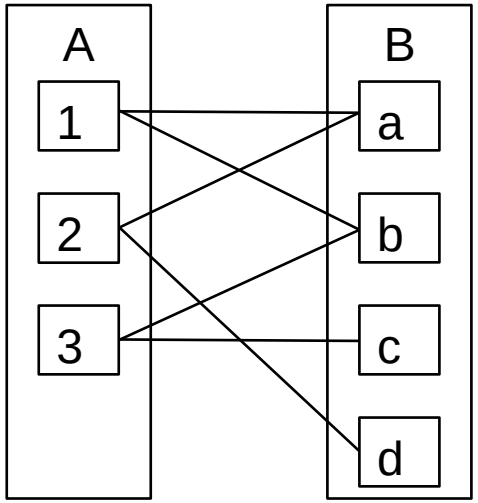
- Analog zu 1:m
- Jede Entity aus A steht mit höchstens einer Entity aus B in Beziehung
- Jede Entity aus B steht mit null oder mehr Entities aus A in Beziehung.



Elemente des ER-Modells (Kardinalitäten)

m:n-Beziehung (many-to-many)

- Jede Entity aus A steht mit null oder mehr Entities aus B in Beziehung
- Jede Entity aus B steht mit null oder mehr Entities aus A in Beziehung



Elemente des ER-Modells (Kardinalitäten)

Erweiterung auf n-stellige Beziehungen:

- **m:1-Beziehung (many-to-one)**
  - Binäre Beziehungen: Eine Beziehung $R(E_1, E_2)$ “von E_1 nach E_2”, bei der jedes Entity aus E_1 zu höchstens einem Entity aus E_2 in Beziehung steht (nicht notwendig umgekehrt), heißt *m:1-Beziehung*.

  - N-stellige Beziehungen: Eine Beziehung $R(E_1, \dots, E_j, \dots, E_n)$ heißt “*m:1* von $E_1, \dots, E_{j-1}, E_{j+1}, \dots, E_n$ nach E_j”, falls jede Auswahl von Entities aus $E_1, \dots, E_{j-1}, E_{j+1}, \dots, E_n$ höchstens ein Entity in E_j bestimmt.

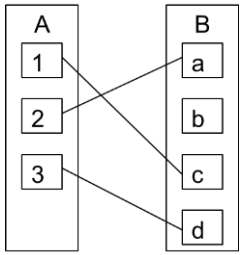
- Es besteht eine funktionale Abhängigkeit.
  Deswegen wird die Kardinalität auch als Funktionalität bezeichnet.

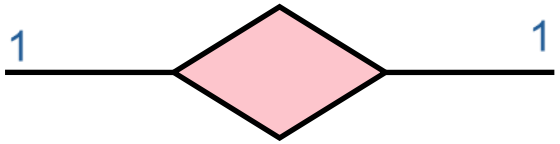
Elemente des ER-Modells (Kardinalitäten)

- Welche Kardinalität ist für eine Beziehung am besten geeignet?
  - Die Antwort hängt davon ab, was modelliert werden soll!
  - Man könnte einfach überall m:n Beziehungen verwenden, aber das wäre nicht klug.
- Ziel:
  - Die Kardinalität sollte reflektieren, was zulässig sein sollte.
  - Die Datenbank kann diese Einschränkungen automatisch erzwingen.
  - Ein gutes Datenbankdesign reduziert oder eliminiert die Möglichkeit, falsche Daten zu speichern.

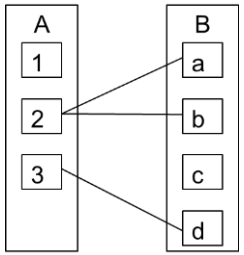
Elemente des ER-Modells (Kardinalitäten)
m:n Notation

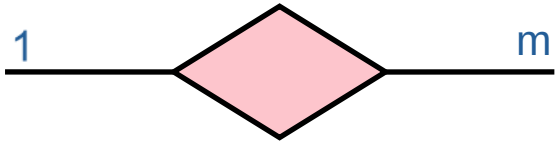
1:1-Beziehung (one-to-one)



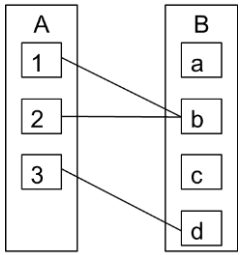


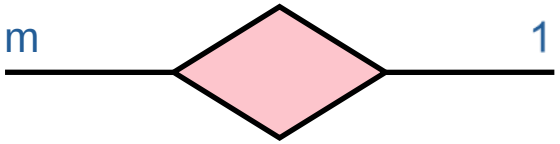
1:m-Beziehung (one-to-many)



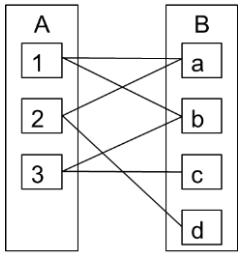


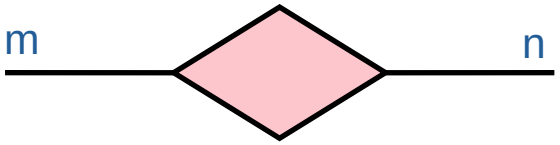
m:1-Beziehung (many-to-one)





m:n-Beziehung (many-to-many)

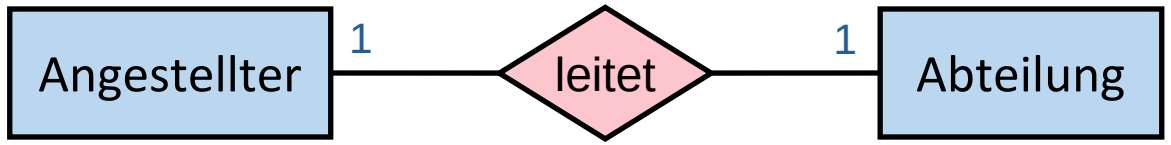




Elemente des ER-Modells (Kardinalitäten)

Beispiele

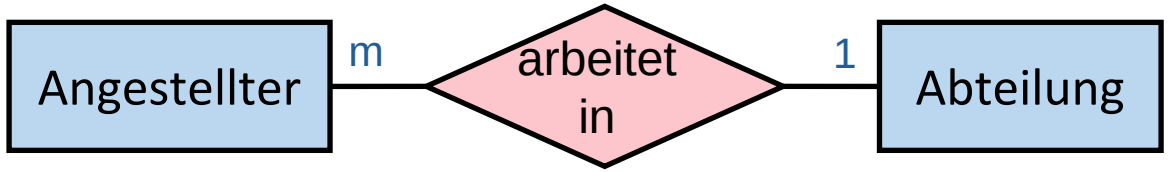
- <Abteilung> *wird geleitet von* <Angestellter>
  - 1:1-Beziehung unter der Annahme, dass jede Abteilung genau einen Leiter hat und kein Angestellter mehr als eine Abteilung leitet.



Elemente des ER-Modells (Kardinalitäten)

Beispiele:

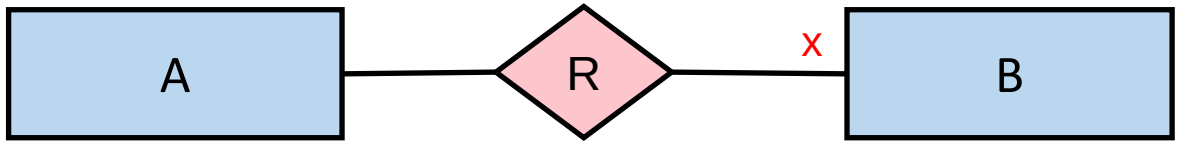
- <Angestellter> *arbeitet in* <Abteilung>
  - m:1-Beziehung unter der Annahme, dass jeder Angestellte in genau einer Abteilung arbeitet.



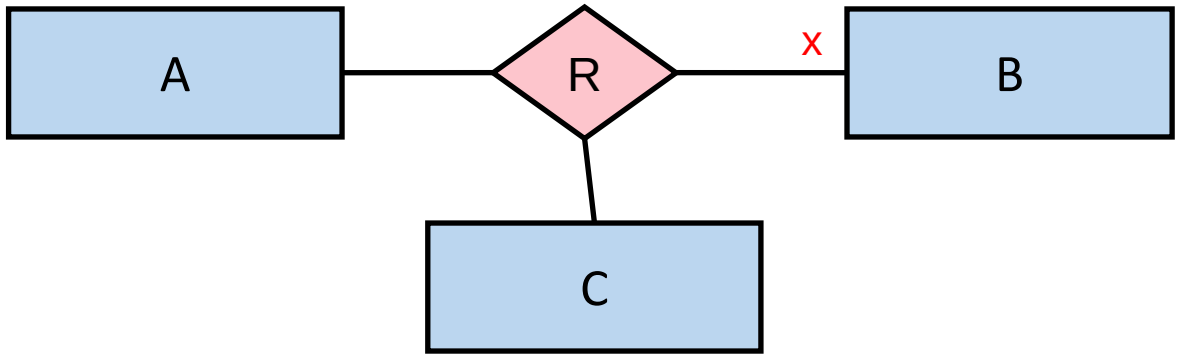
- <Student> *hört* <Vorlesung>
  - m:n-Beziehung, weil Studierende im allgemeinen mehrere Vorlesungen hören und Vorlesungen in der Regel von mehreren Studierenden besucht werden.

Elemente des ER-Modells (Kardinalitäten)

Kardinalitäten für zwei- und mehrstellige Beziehungen: **1:n-Notation**



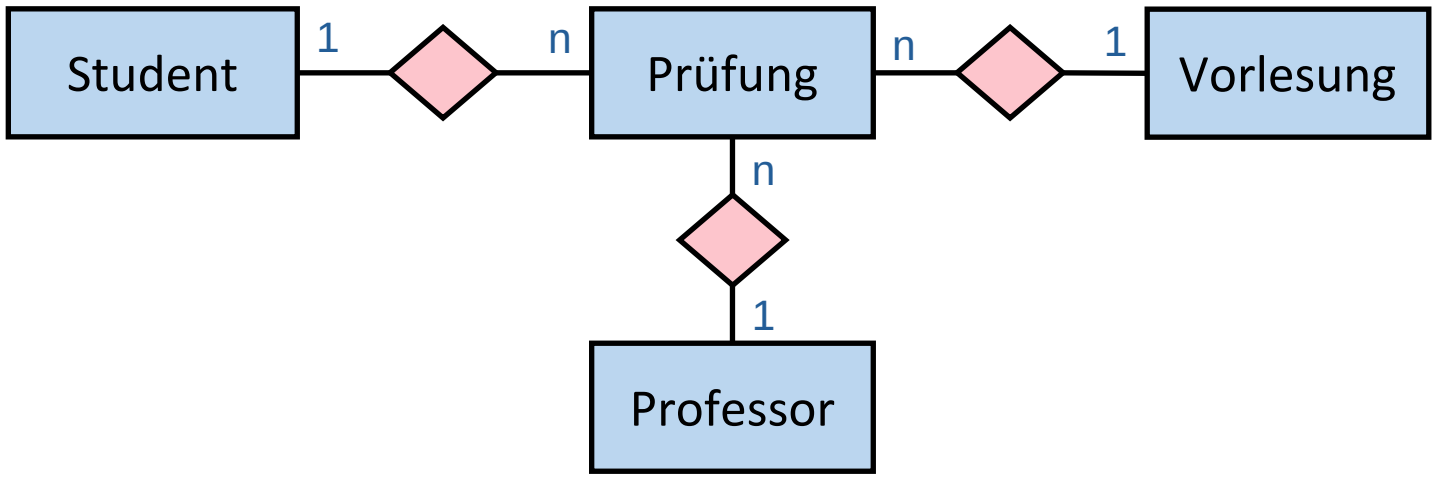
- x: Wie viele Entities des Entity Typ B stehen in Beziehung zu einer Entity des Entity Typ A?



- x: Wie viele Entities des Entity Typ B stehen in Beziehung zu einem (a,c)-Paar mit a∈A und c∈C?

Elemente des ER-Modells (Kardinalitäten)

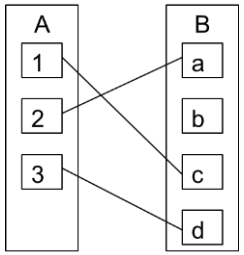
- Eine ternäre Beziehung kann als binäre Beziehungen dargestellt werden
  - Damit die Semantik weitestgehend erhalten bleibt ist ein neuer Entity-Typ notwendig

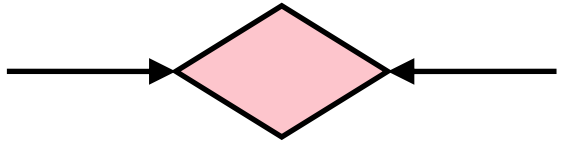


- Aber: Die Einschränkung, dass Studenten eine Vorlesung nur bei einem Professor prüfen lassen können, ist nicht darstellbar

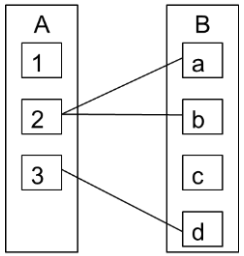
Elemente des ER-Modells (Kardinalitäten)
Pfeilnotation

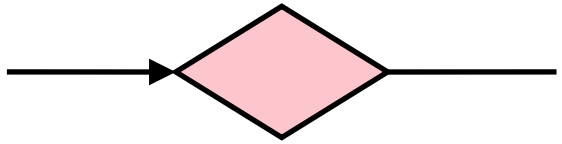
1:1-Beziehung (one-to-one)



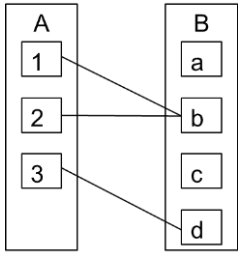


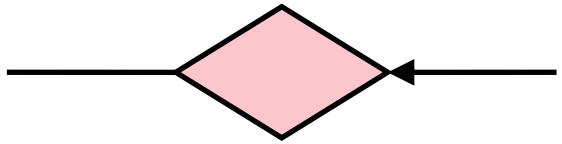
1:m-Beziehung (one-to-many)



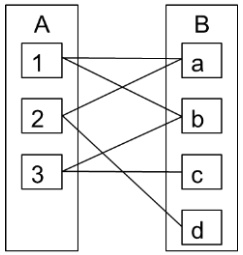


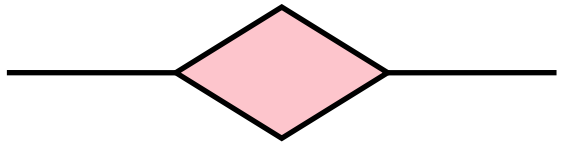
m:1-Beziehung (many-to-one)



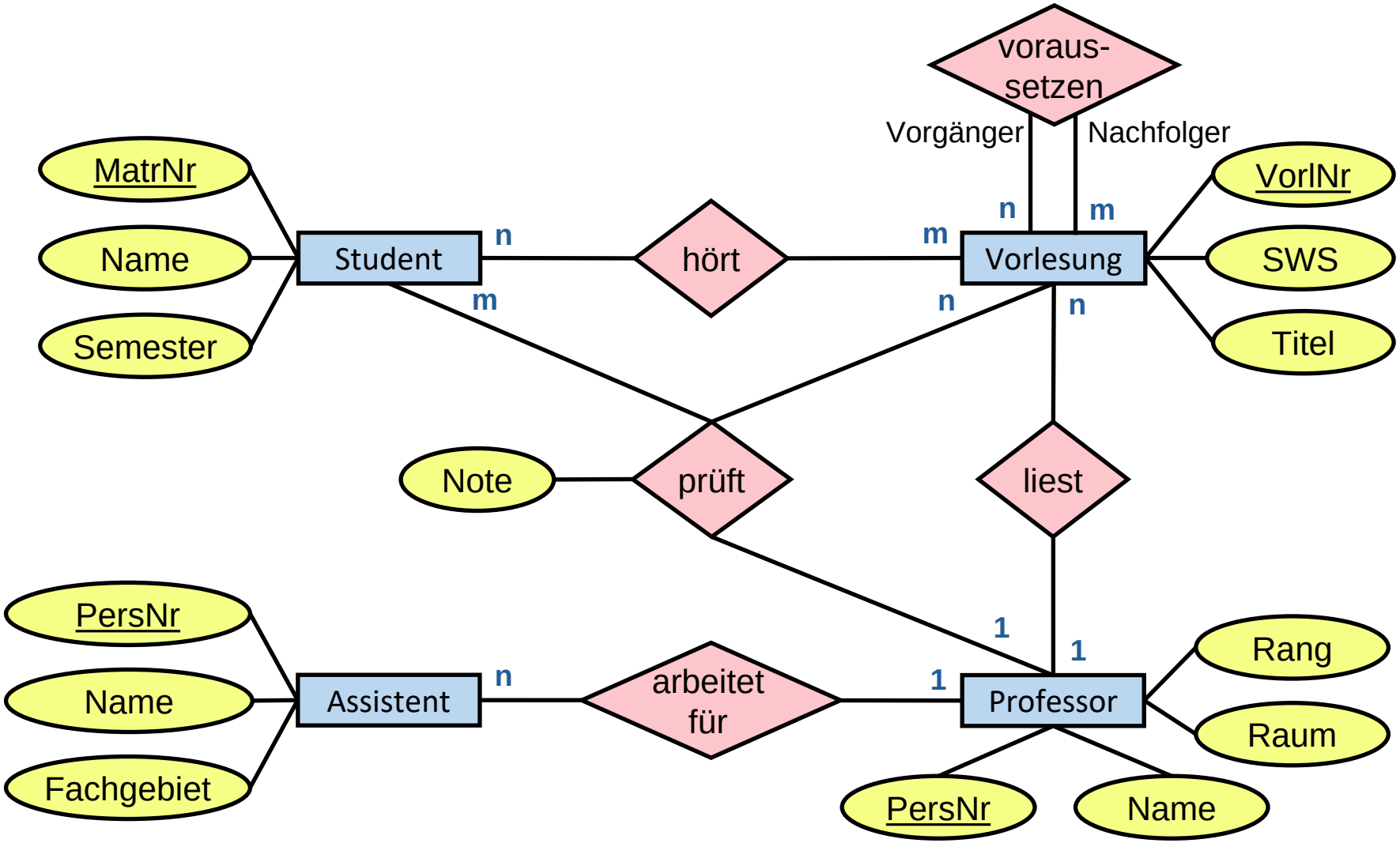


m:n-Beziehung (many-to-many)





Beispiel: ER-Diagramm (Universität)
Entity Typen + Attribute + Beziehungen + Kardinalitäten



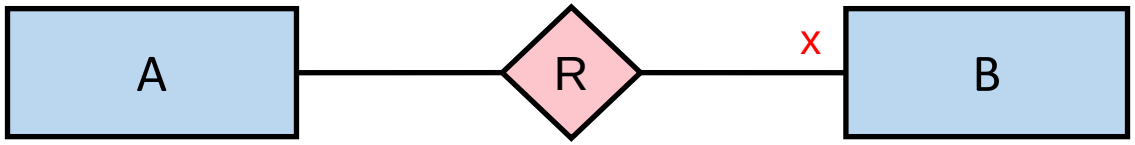
Elemente des ER-Modells (Kardinalitäten)

- Alternative Notation für Kardinalitäten: (min,max)-Notation
  - Erlaubt eine Beschränkung der minimalen Anzahl korrespondierender Entities
  - (min,max)-Notation und 1:n-Notation haben unterschiedliche Ausdruckstärke

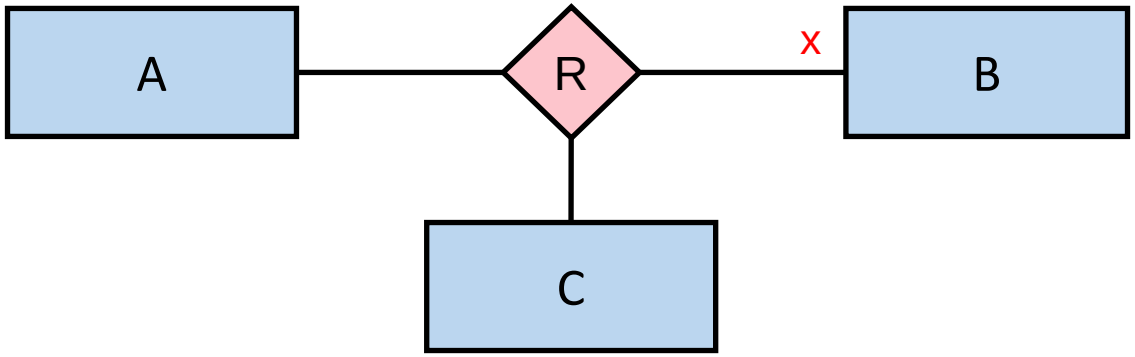
- Achtung: Positionsvertauschung in (min,max)-Notation

Elemente des ER-Modells (Kardinalitäten)

Kardinalitäten für zwei- und mehrstellige Beziehungen: **(min,max)-Notation**



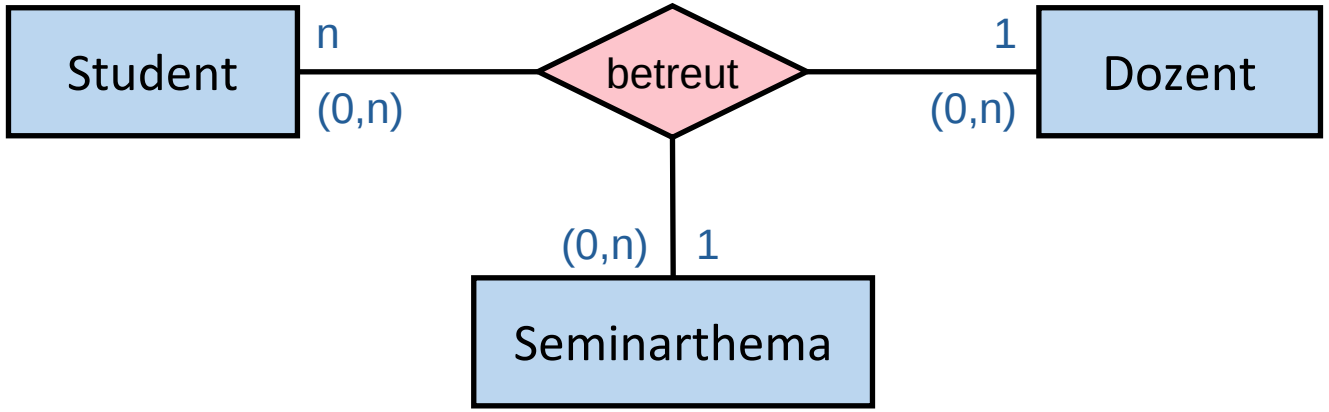
- x: An wie vielen R-Ausprägungen ist ein Element des Entity Typs B beteiligt?
  = Wie viele Elemente des Entity Typs A stehen in Beziehung zu einem Element des Entity Typs B?



- x: An wie vielen R-Ausprägungen ist ein Element des Entity Typs B beteiligt?
  = Mit wievielen Paare (*a*,*c*) mit *a*∈A und *c*∈C steht ein Element des Entity Typs B in Beziehung?

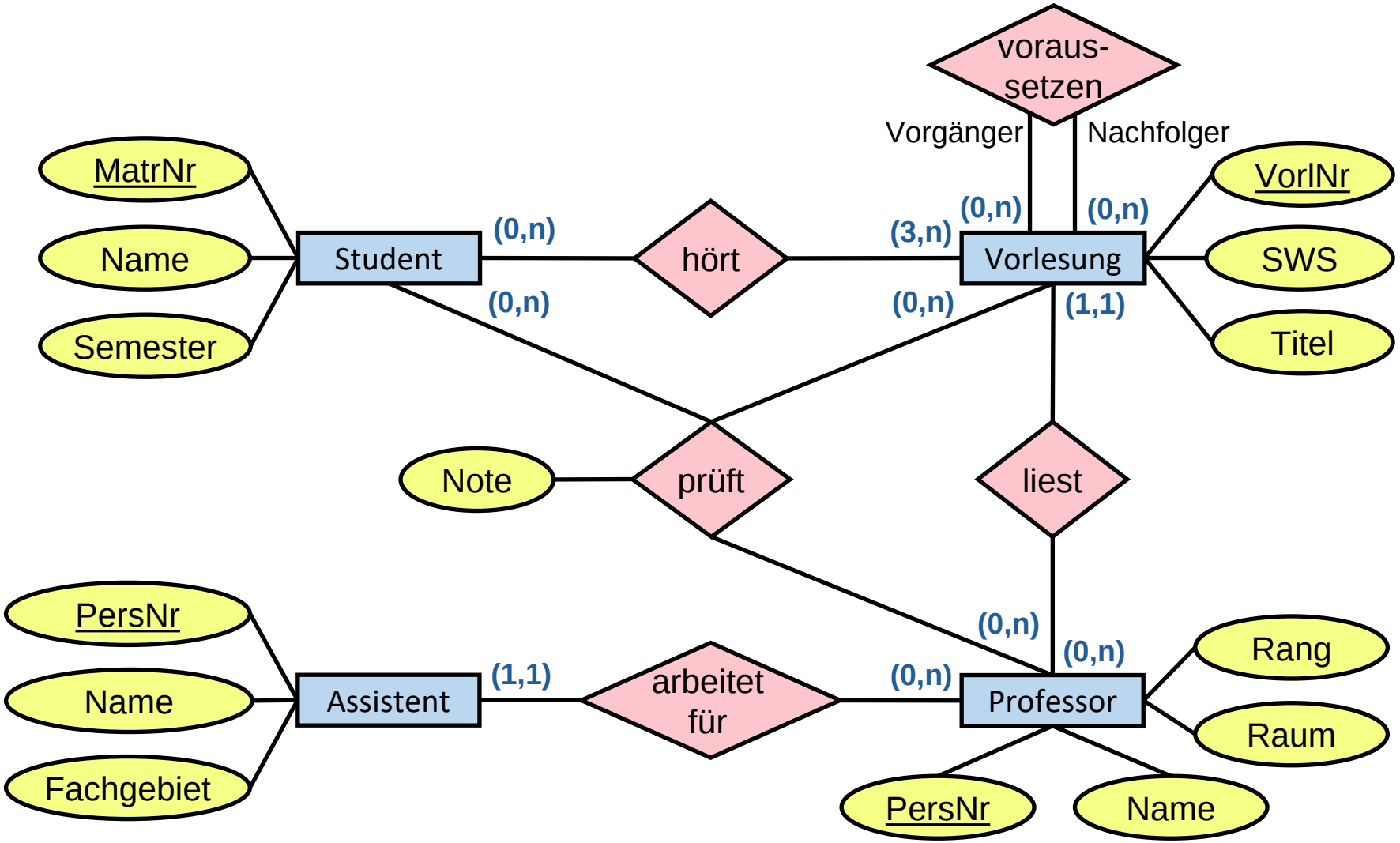
Elemente des ER-Modells (Kardinalitäten)

Beispiel:



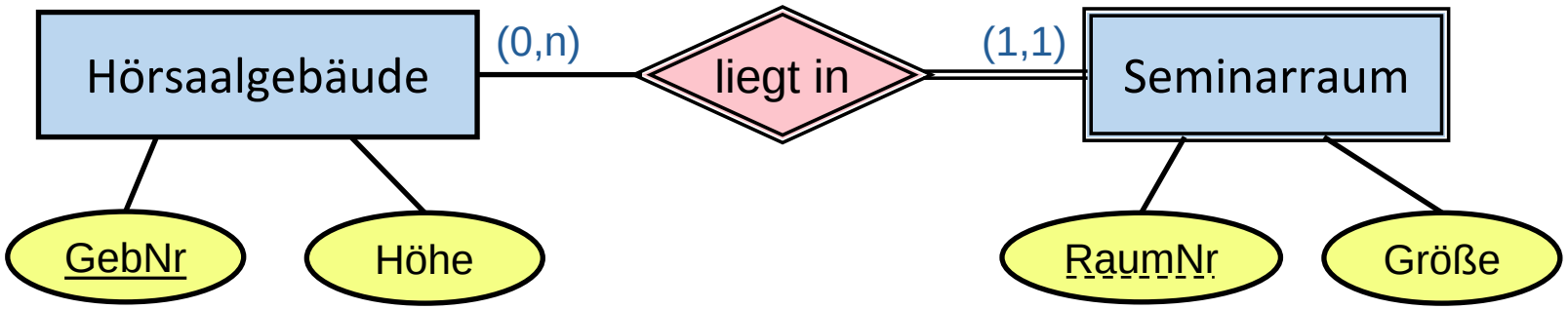
- Annahmen über die Domäne
  - Studenten dürfen pro Dozent nur ein Seminarthema belegen
  - Studenten dürfen Seminarthema nur einmal belegen
  - Dozenten können Seminarthemen wiederverwenden
  - Das gleiche Thema kann von verschiedenen Dozenten vergeben werden

Beispiel: ER-Diagramm (Universität)
Entity Typen + Attribute + Beziehungen + Kardinalitäten (min,max)



Elemente des ER-Modells (Schwache Entity-Typen)

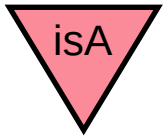
- Ein schwacher Entity-Typ ist von der Existenz des übergeordneten Entity-Typs abhängig.
- *Beispiel*: Ohne Hörsaalgebäude kann es keine Seminarräume geben
  - Kardinalitätsrestriktion zu übergeordneten Entity-Typ in (min,max): (1,1)
  - Der schwache Entity-Typ (*Seminarraum*) und seine Relation (*liegt in*) zum übergeordneten Entity (*Hörsaalgebäude*) werden durch doppelte Umrandung des Entity-Typ, der Verbindung zur Beziehung und der Beziehung selbst dargestellt



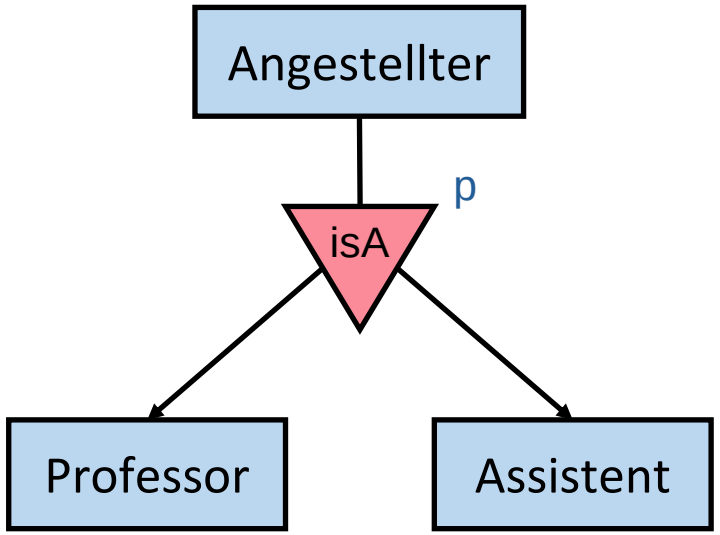
Schlüssel von schwachen Entity-Typen: gestrichelt unterstrichen

Elemente des ER-Modells (Generalisierung/Spezialisierung)

- Vererbungsbeziehung (*isA*) zwischen Entity-Typ und spezialisierten Entity-Typ
  - Spezialisierter Entity-Typ erbt von dem allgemeinen Entity-Typ
  - Eine *isA* Beziehung wird durch ein umgedrehtes Dreieck dargestellt
    Ein Unterscheidungsmerkmal bestimmt die Zugehörigkeit
    und kann explizit mitmodelliert werden.



Beispiel:



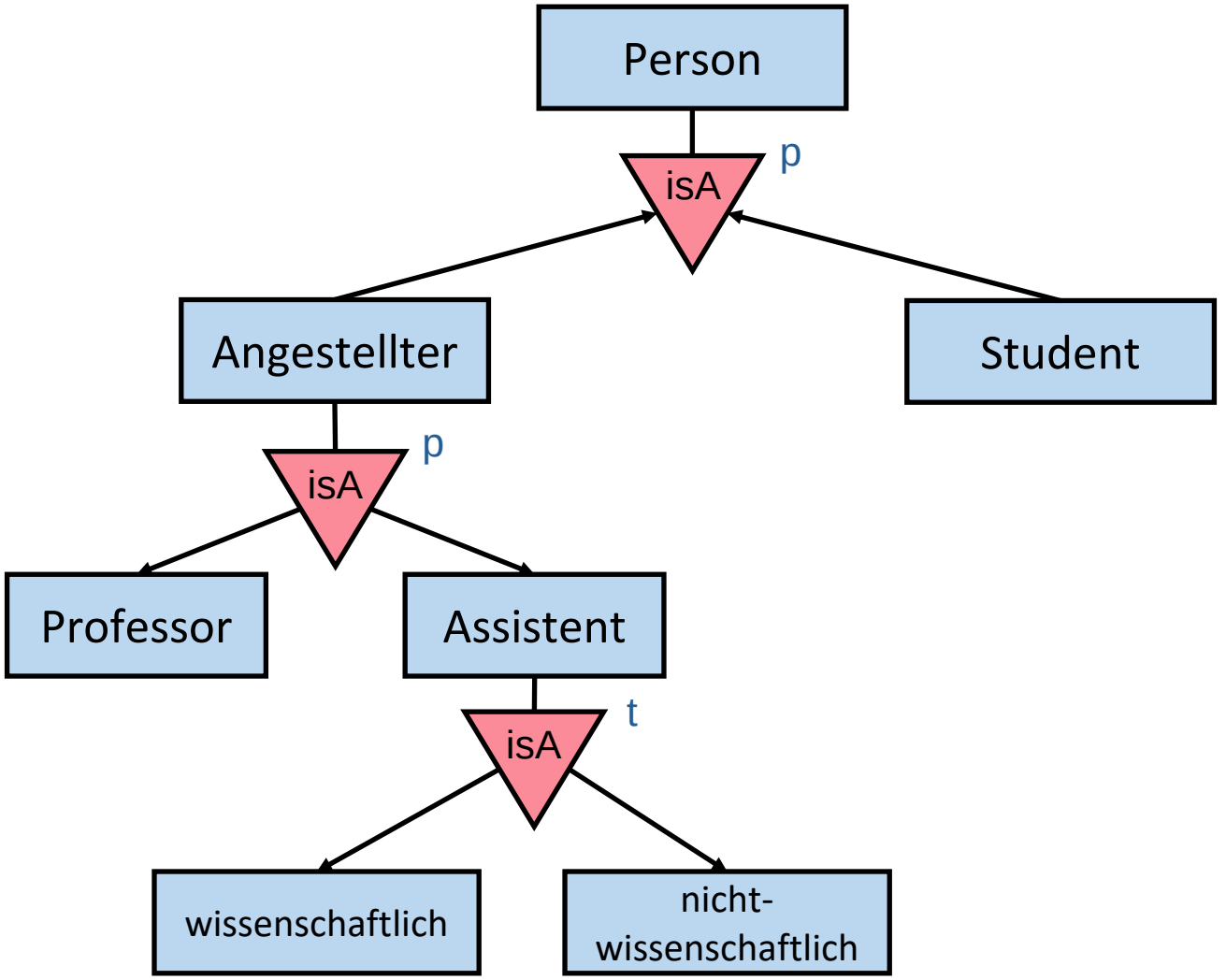
Elemente des ER-Modells (Generalisierung/Spezialisierung)

- Merkmale (Generalisierung/Spezialisierung)
  - **Disjunkt**
  - Spezialisierungen sind disjunkt (Ein Angestellter kann nicht Assistent und Professor sein)
  - *Pfeile zeigen auf die Spezialisierung*
  - **Nicht disjunkt**
  - Spezialisierung sind nicht disjunkt (Eine Person kann Angestellter und Student sein)
  - Pfeile Zeigen auf in Richtung der Generalisierung

  - **Total (t)**
  - Die Dekomposition der Generalisierung ist vollständig (Es gibt entweder wissenschaftliche oder nicht wissenschaftliche Mitarbeiter)
  - Wird durch "*t*" neben der isA Beziehung dargestellt
  - **Partiell (p)**
  - Die Vereinigung der Spezialisierung ist eine echte Untermenge der Generalisierung
  - Wird durch "*p*" neben der isA Beziehung dargestellt

Elemente des ER-Modells (Generalisierung/Spezialisierung)

Beispiel:



Elemente des ER-Modells (Generalisierung/Spezialisierung)

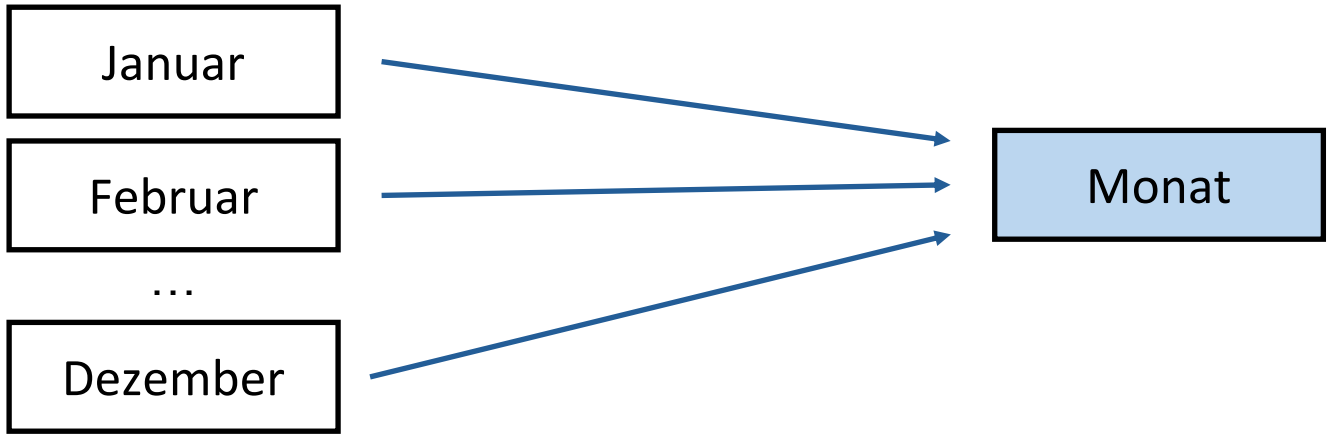
Beispiele	Disjunkt	Nicht disjunkt
Total	• Raum unterteilt in: Hörsaal, Büro, Seminarraum,… • Studenten dieser Vorlesung: Klausur teilgenommen vs. Nicht teilgenommen	• Student mit Sub-Entity-Typ für **jeden** Studiengang • Student unterteilt in Master, Bachelor, Promotion (man kann in mehreren Studiengängen eingeschrieben sein)
Partiell	• Studenten dieser Vorlesung: Klausur bestanden vs. Nicht bestanden (es gibt auch welche, die nicht teilgenommen haben)	• Student mit Sub-Entity-Typ **nur** für Studiengänge der Fakultät 1

2. Das Entity-Relationship-Modell

1. Der Datenbankentwurf
2. Entity-Relationship-Modell
3. **Konzeptueller Entwurf**

Konzeptueller Entwurf (Abstraktionskonzepte)

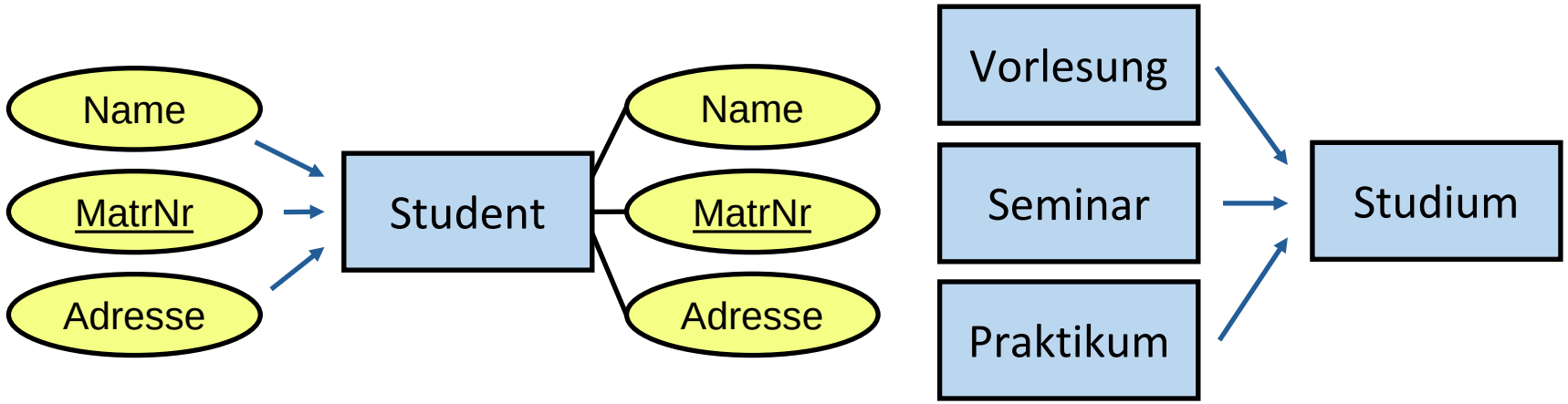
- **Klassifikation**
  - *Abstraktion von einer Menge von Objekten mit ähnlichen Eigenschaften auf eine Klasse*
  - *Beispiel:* Klassifikation von Monatsobjekten (*Januar*, …, *Dezember*) zum Entity-Typ *Monat*



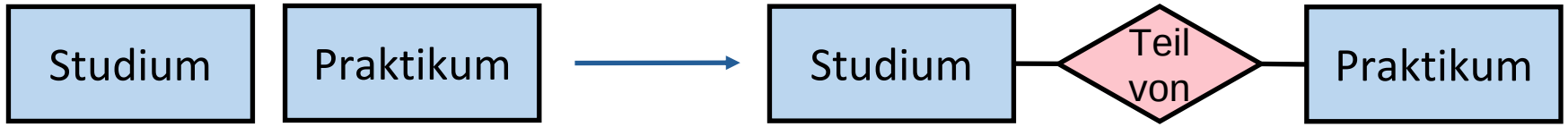
- **Identifikation**
  - Hinzufügen von Identifikators (Beispiel: Schlüsselattribute)

Konzeptueller Entwurf (Abstraktionskonzepte)

- **Aggregation**
  - Abstraktion von einer Menge von Komponenten oder Teilen auf das Ganze
    - Verschiedene Attribute werden zu einem Entity-Typ aggregiert
    - Verschiedene Entity-Typen werden zu einem neuen Entity-Typ aggregiert



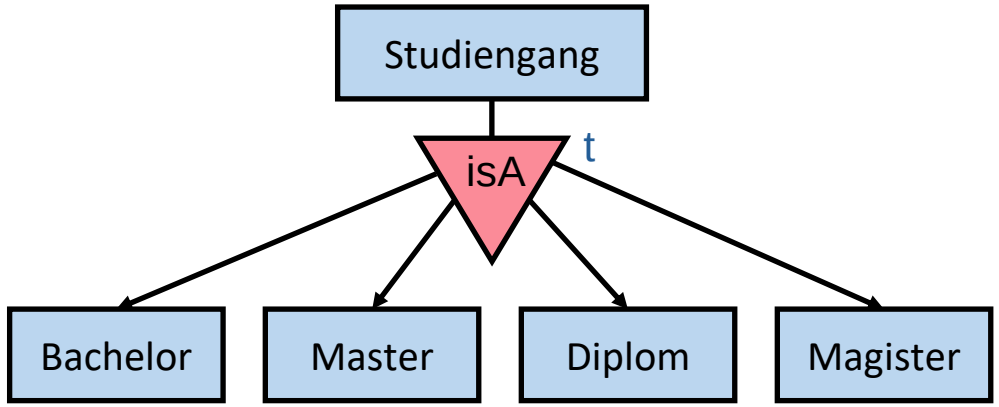
- **Assoziation**
  - Unabhängige Klassen werden in Beziehung gestellt



Konzeptueller Entwurf (Abstraktionskonzepte)

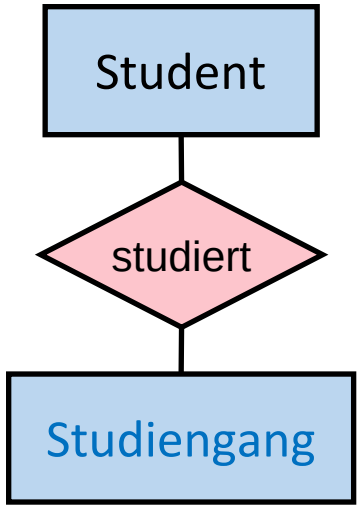
- **Generalisierung/Spezialisierung**
- Zwei Möglichkeiten der Modellierung

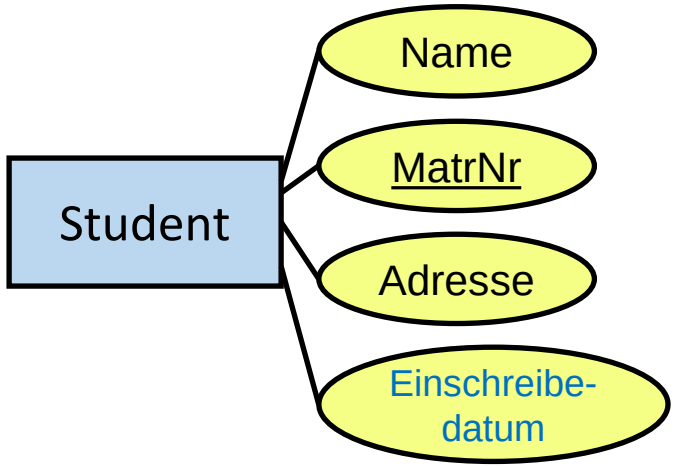
- Top Down (Spezialisierung)
  - Es werden zuerst sehr große Informationsblöcke modelliert
  - Anschließend werden die Blöcke schrittweise spezialisiert
- Bottom Up (Generalisierung)
  - *Es werden zuerst sehr detaillierte Informationen modelliert*
  - *Schrittweise werden diese Informationen verallgemeinert*



Konzeptueller Entwurf (Richtlinien)

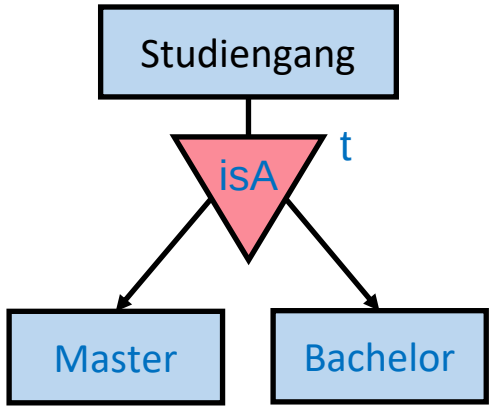
- **Entity-Typ oder Attribut**
  - Entity-Typ, falls das Konzept selber Eigenschaften hat oder mehrfach im Modell auftaucht (z.B.: in Beziehung zu anderen Entities)
  - Andernfalls wird das Konzept als Attribut modelliert
  - *Beispiele:*
    - Studiengang eines Studenten Entity-Typ
    - Einschreibedatum eines Studenten Attribut

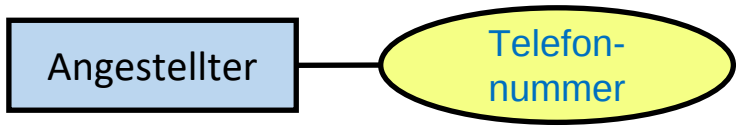




Konzeptueller Entwurf (Richtlinien)

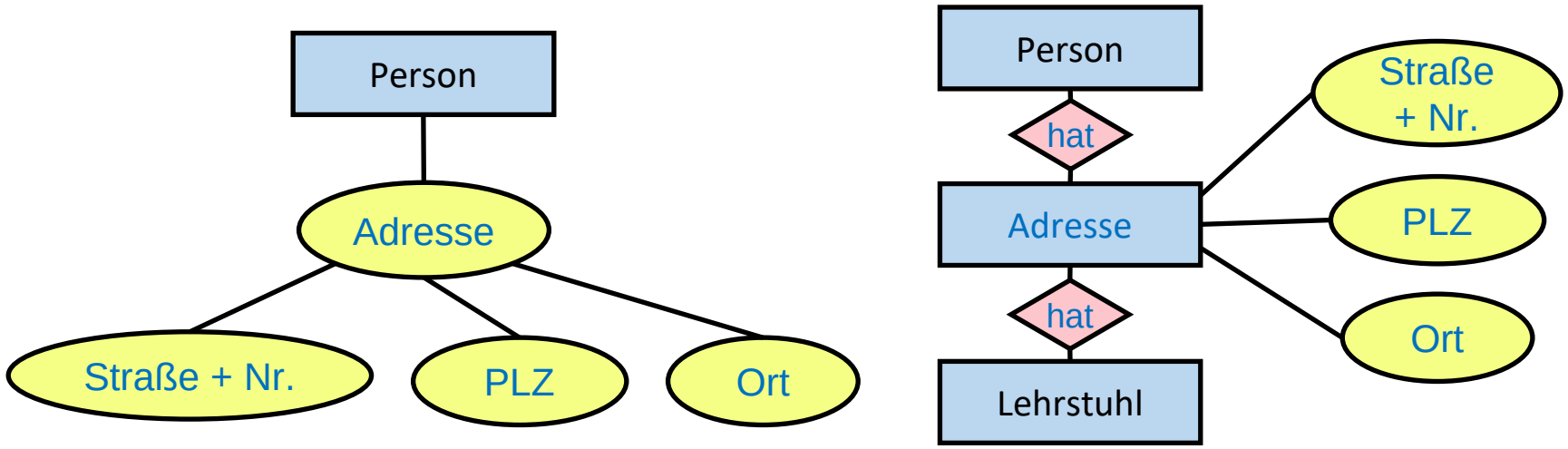
- **Generalisierung/Spezialisierung oder Attribut**
  - Generalisierung/Spezialisierung, falls das Unterkonzept zusätzliche Attribute oder Beziehungen hat, oder nun spezielle semantische Constraints ausdrückt
  - Andernfalls wird das Konzept als Attribut modelliert
  - *Beispiele:*
    - Bachelor und Master Studiengänge Spezialisierung
    - Telefonnummer eines Angestellten Attribut



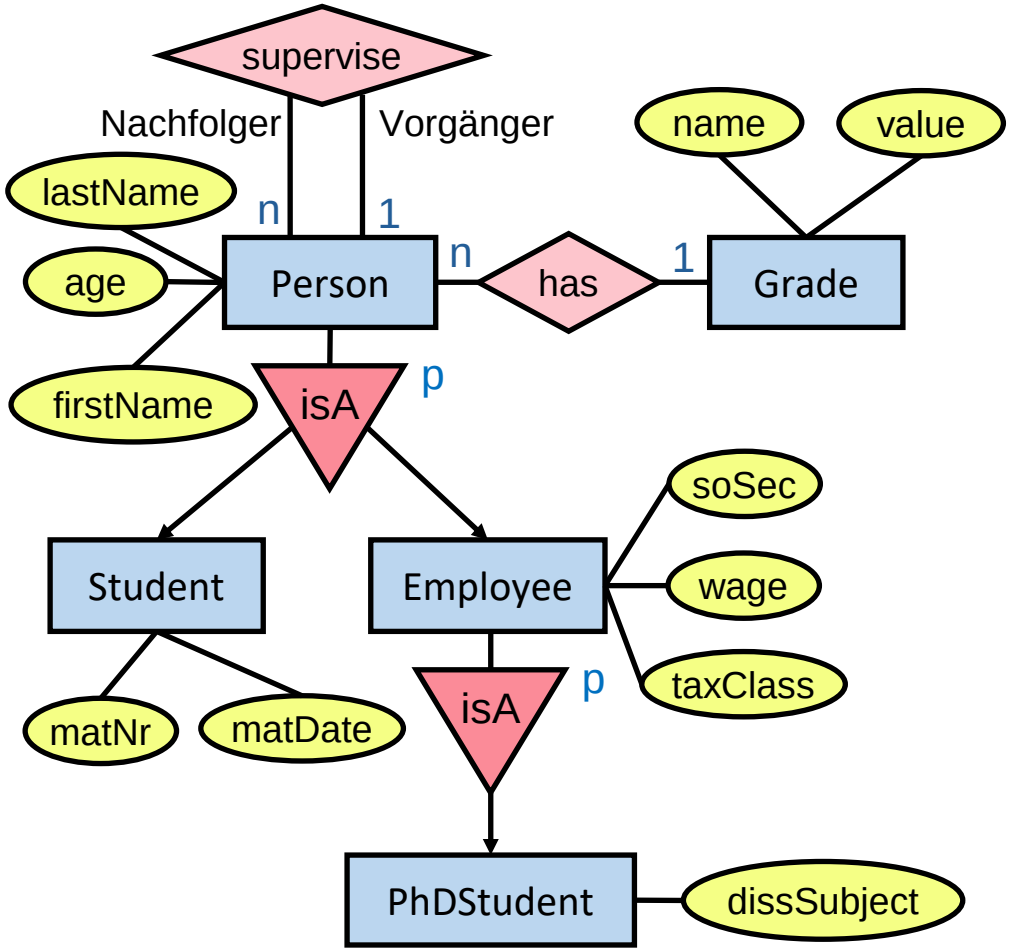


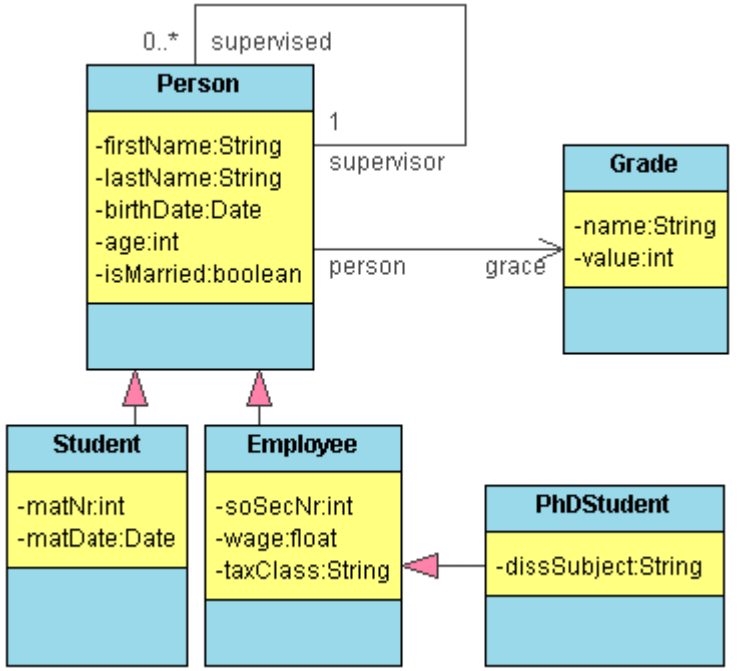
Konzeptueller Entwurf (Richtlinien)

- **Zusammengesetztes Attribut, einfaches Attribut oder Entity-Typ**
  - Zusammengesetztes Attribut, falls der Name des Attributs eine spezielle Bedeutung hat
  - Entity-Typ, falls es mehrfach verwendet wird und die Information gemeinsam genutzt werden soll
  - Andernfalls wird das Konzept als Attribut modelliert
  - *Beispiel:*
    - Adresse von Personen Zusammengesetztes Attribut
    - Adresse von Personen und Lehrstühlen Entity-Typ



ER und UML


supervise
Nachfolger
Vorgänger
n
1
lastName
age
firstName
Person
n
has
1
name
value
Grade
isA
p
Student
Employee
soSec
wage
taxClass
matNr
matDate
isA
p
PhDStudent
dissSubject

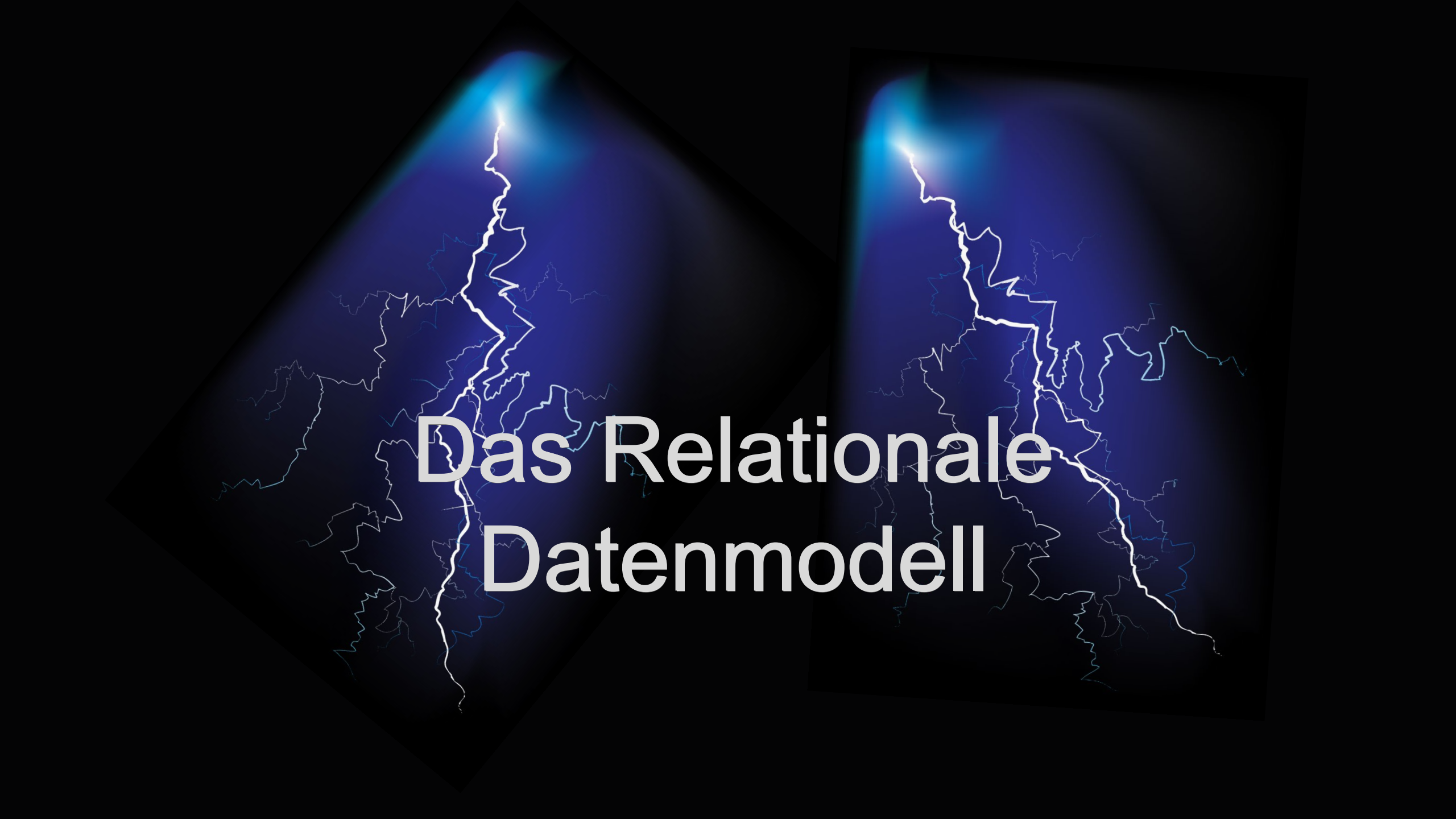

0..*
supervised
Person
-firstName:String
-lastName:String
-birthDate:Date
-age:int
-isMarried:boolean
1
supervisor
person
grade
Grade
-name:String
-value:int
Student
-matNr:int
-matDate:Date
Employee
-soSecNr:int
-wage:float
-taxClass:String
PhDStudent
-dissSubject:String

Zusammenfassung

- **Der Datenbankentwurf**
  - Datenbank-Lebenszyklus
  - Qualitätskriterien für den Datenbankentwurf

- **Entity-Relationship-Modell**
  - Elemente des ER-Modell
    - Entities, Entity-Typen
    - Attribute
    - Relationships (Beziehungen)
    - Generalisierung/Spezialisierung
  - Nicht standardisiert unterschiedliche Notationen (1:n, (min,max))

- **Konzeptueller Entwurf**
  - Abstraktionskonzepte
  - Richtlinien



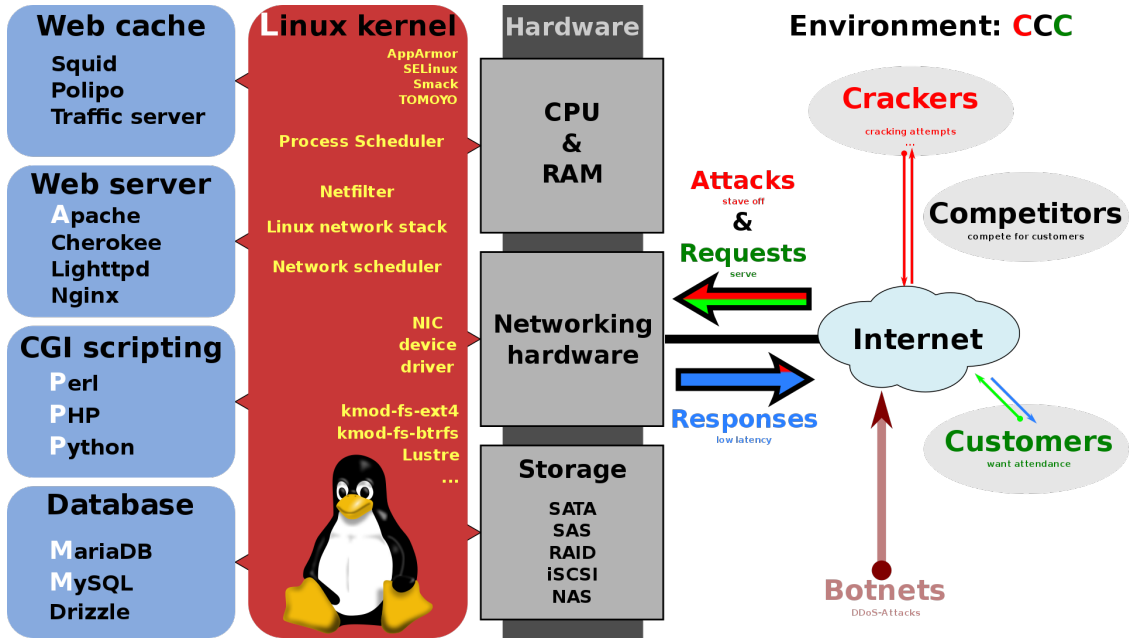
Das Relationale Datenmodell

3. Das Relationale Datenmodell

1. **Das Datenmodell**
2. Transformation von ER-Diagrammen in das Relationale Modell
3. Relationale Algebra
4. Relationaler Kalkül

Beispiele für Relationale Datenbanken-Anwendungen: Teil des LAMP Stacks

- LAMP (Linux, Apache, MySQL, PHP/Perl/Python)
  - Verbreiteter Software-Stack für Webanwendungen
- MySQL: relationales Datenbankmanagementsystem (RDBMS) im LAMP-Stack
  - Andere RDBMS-Optionen: PostgreSQL, MariaDB
- Anwendungsbeispiele:
  - Content-Management-Systeme (z.B. WordPress, Joomla)
  - E-Commerce-Plattformen (z.B. Magento, PrestaShop)
  - Webanwendungen mit Datenbankzugriff und -verwaltung (z.B. Kundenportale, CRM-Systeme)



Datenbanken – Wie machen es die “Großen”?
Beispiel: Google Spanner und Facebook TAO

Google Spanner [1][2]:

- Verteilte “NewSQL” Datenbank mit “multi-version concurrency control”.
  - “Semi-relationales” Datenmodel.
  - Unterstützt SQL Anfragen und Transaktionen.

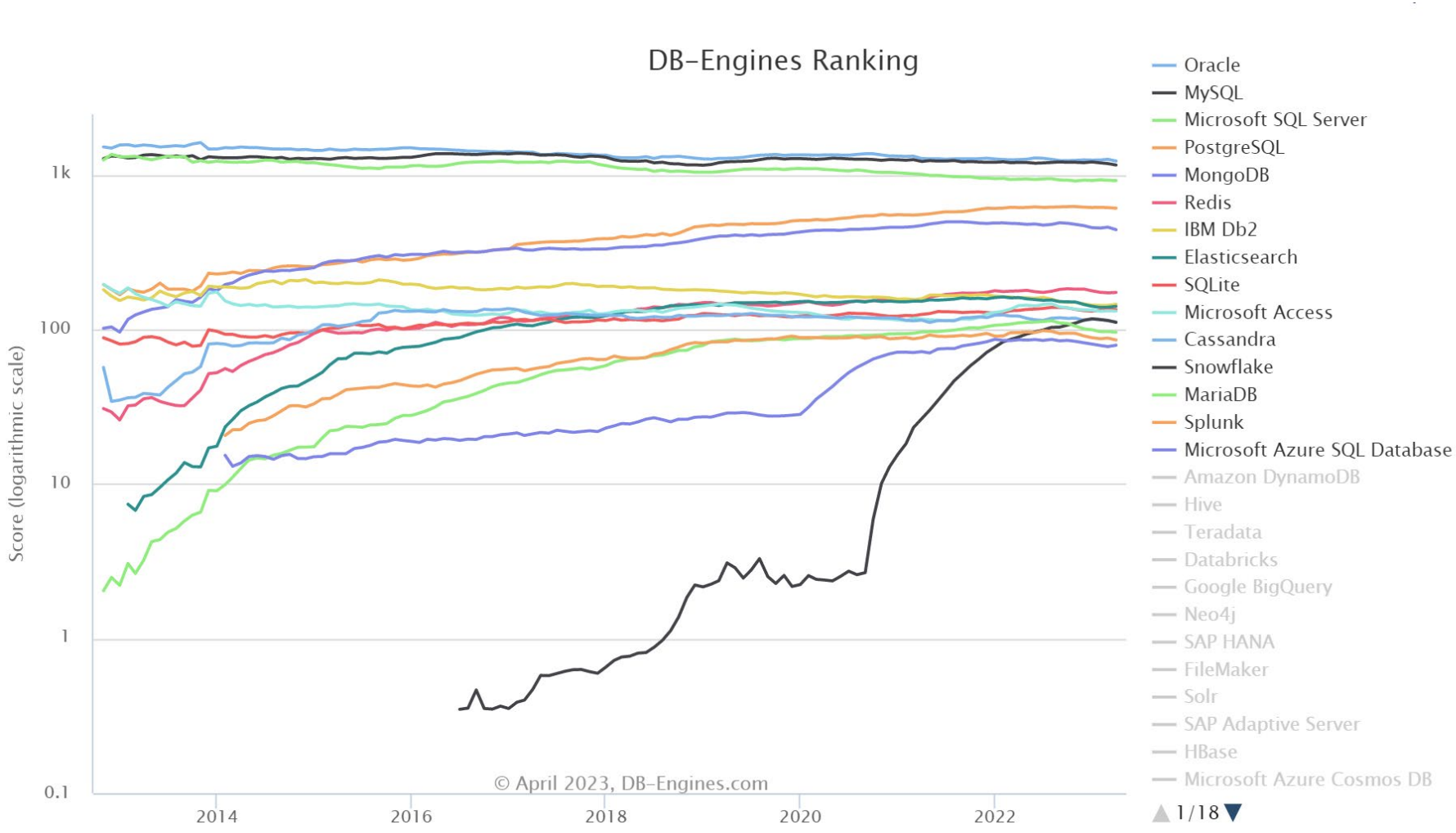
Facebook TAO [3]:

- Verteilte Graphdatenbank – verwendet MySQL zur Persistenz.
  - Optimiert für Lesen.
  - Wertet Effizienz und Verfügbarkeit höher als Konsistenz.

TAO

[1] J.C. Corbett et al.: Spanner: Google's Globally-Distributed Database. OSDI 2012: 251-264
[2] D. F. Bacon et al: Spanner: Becoming a SQL System. SIGMOD Conference 2017: 331-343
[3] N. Bronson, et al.: TAO: Facebook's Distributed Data Store for the Social Graph. USENIX Annual Technical Conference 2013: 49-60

Datenbankranking



https://db-engines.com/de/ranking_trend

Einführung

- Das relationale Datenmodell nutzt das einfache Strukturierungsprinzip “Tabelle” zur Modellierung sowohl von Objekten als auch von Beziehungen.

- Das Modell geht auf *Codd* (1970) zurück und hat sich seither auf breiter Basis durchgesetzt.
  - Edgar F. Codd: *A Relational Model of Data for Large Shared Data Banks*. Commun. ACM 13(6): 377-387 (1970), see http://dblp.dagstuhl.de/pers/hd/c/Codd:E=_F=

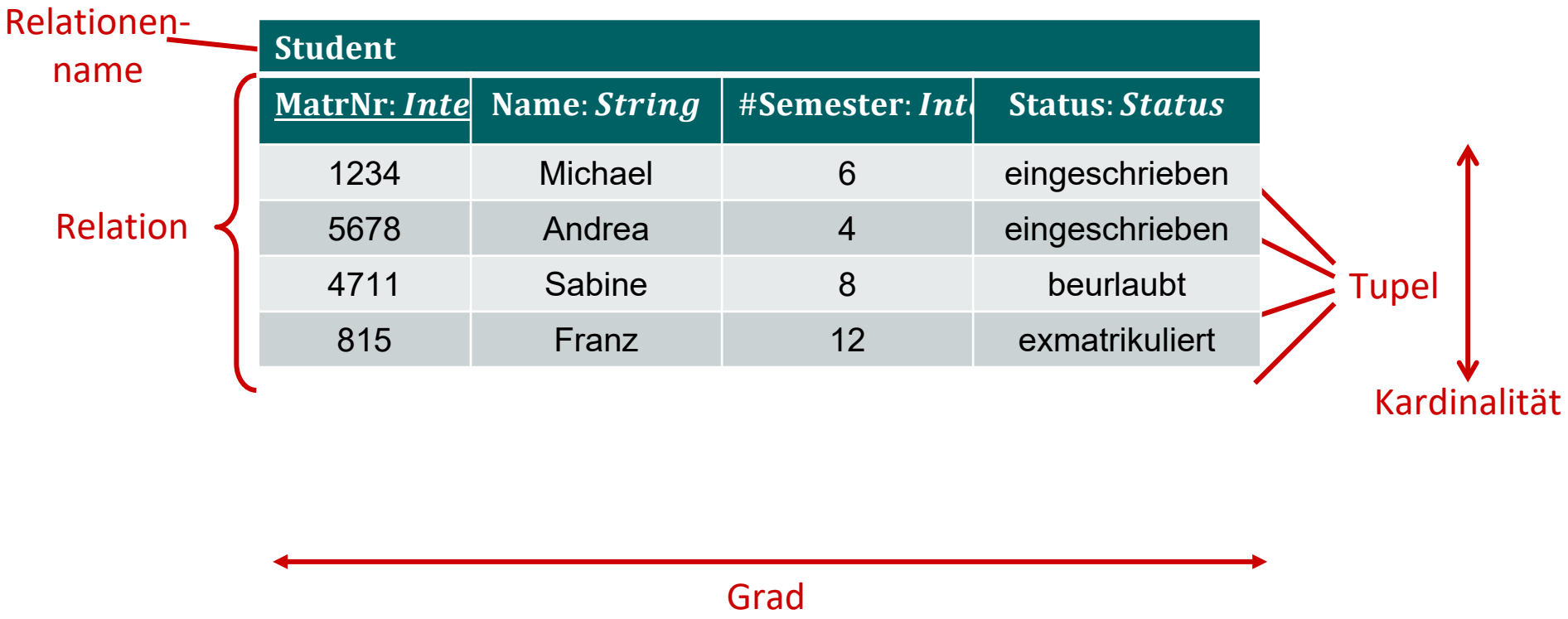
- Heute ist es Grundlage vieler kommerzieller Datenbanksysteme (Oracle, IBM DB/2, Informix, Ingres, Sybase, MS SQL-Server, MS Access, usw.) und damit wichtiger Bestandteil vieler großer Anwendungssysteme.

- Im naturwissenschaftlich-technischen Bereich dient es vielfach als Grundlage für komplexere Datenmodelle, insbesondere für sogenannte “Nichtstandard-Anwendungen”.

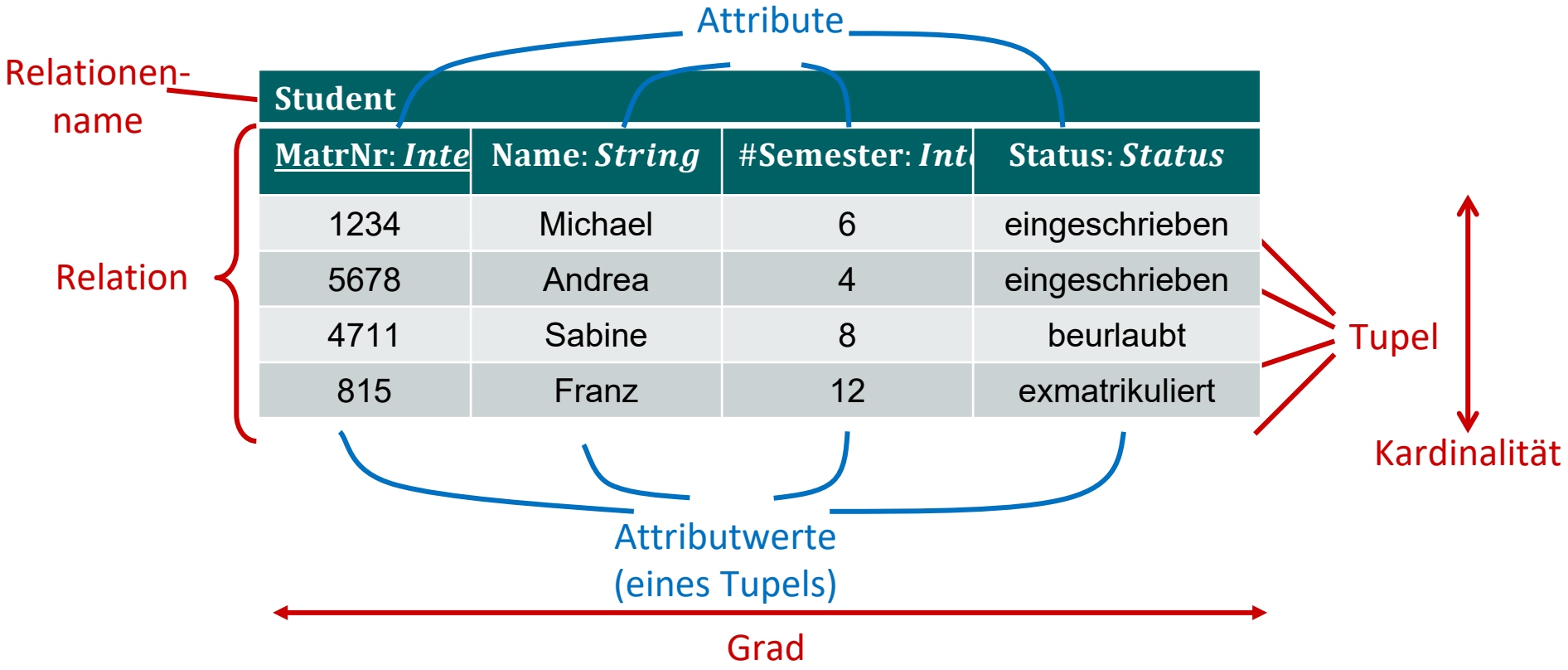
Relationenschemata Terminologie

Student			
MatrNr: *Inte*	Name: *String*	#Semester: *Int*	Status: *Status*
1234	Michael	6	eingeschrieben
5678	Andrea	4	eingeschrieben
4711	Sabine	8	beurlaubt
815	Franz	12	exmatrikuliert

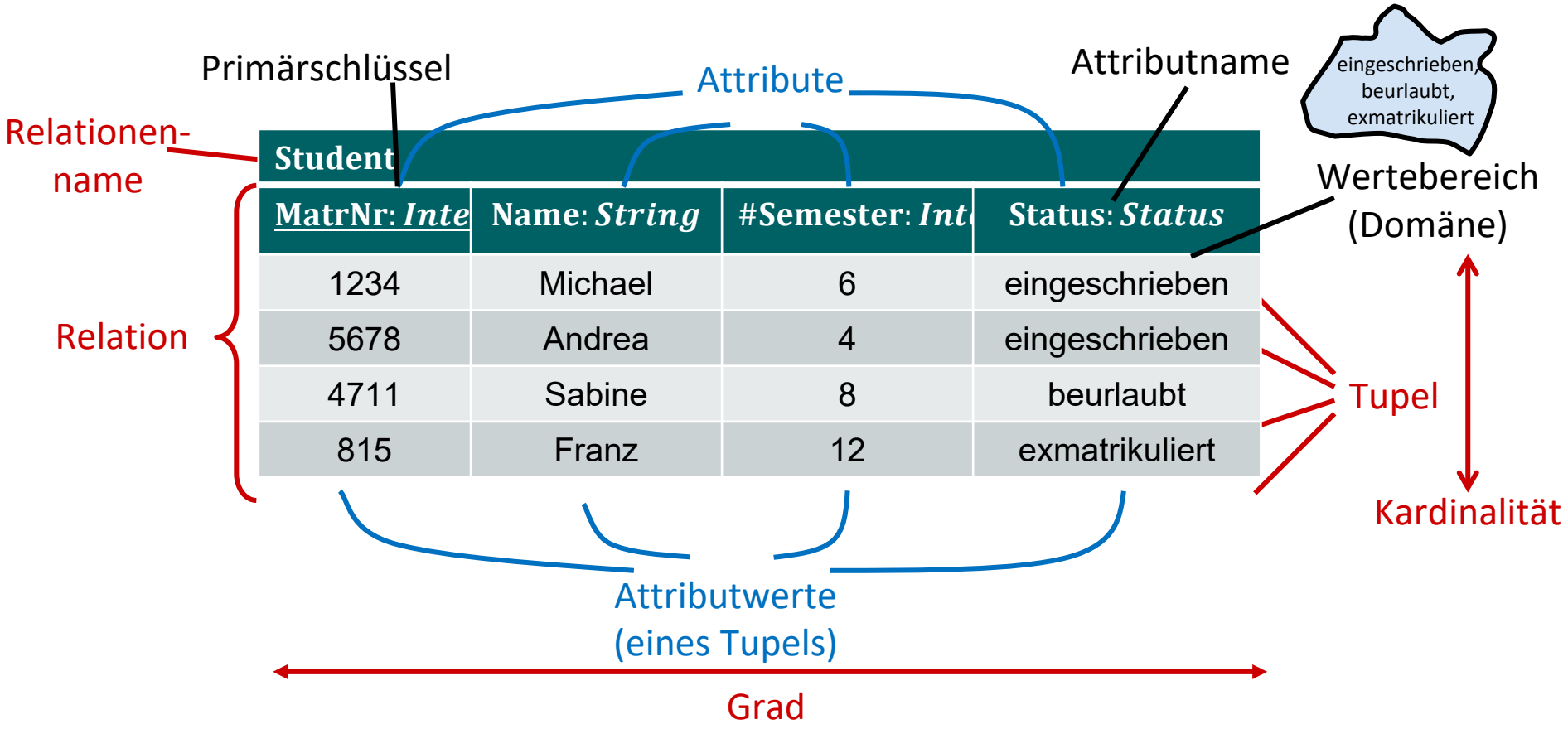
Relationenschemata Terminologie



Relationenschemata Terminologie



Relationenschemata Terminologie



Relationen, Domänen

- Ein *Wertebereich (Domäne, Domain)* ist eine (logisch zusammengehörige) Menge von Werten, z.B. INTEGER, STRING, DATUM, {1, … , 10}, etc. Eine Domäne kann *endliche* oder *unendliche* Kardinalität haben. Die Funktion dom bildet Attribute auf ihren Wertebereich ab.

- Ist $k \geq 1$ und $\{A_1, \ldots, A_k\}$ eine Folge von Attributnamen mit den Wertebereichen $D_i = dom(A_i), 1 \leq i \leq k$, dann ist eine *Relation r* Teilmenge des kartesischen Produktes der Wertebereiche $D_1, \ldots, D_k$:

$$r \subseteq D_1 \times \cdots \times D_k$$

- *Beispiel*
  - Gegeben seien die Wertebereiche $D_1 = \{a, b, c\}$ und $D_2 = \{0,1\}$. Dann gibt es unter anderem die Relationen:
    - $r_1 = D_1 \times D_2 = \{(a,0), (a,1), (b,0), (b,1), (c,0), (c,1)\}$
    - $r_2 = \{(a,0), (b,0), (c,0), (c,1)\}$
    - $r_3 = \emptyset$

Tupel, Tabellen

- Einzelne Elemente einer Relation heißen *Tupel*.
- Für $r \subseteq D_1 \times D_2 \times \cdots \times D_k$ heißt k der *Grad* oder die *Stelligkeit* der Relation; alle Tupel in r haben k Komponenten.

- Relationen kann man als *Tabellen* verstehen und darstellen. Die Zeilen einer Tabelle entsprechen den Tupeln. Die Spalten heißen Attribute; sie können Namen tragen, im Beispiel *MatNr* und *Name*.

- Bemerkung: Im Datenbankbereich betrachtet man gewöhnlich nur endliche Relationen. Unendliche Relationen können nicht materialisiert werden, sie treten z.B. im Bereich der deduktiven Datenbanksysteme auf.

$$R_2 = \{(30487, Bob), (30548, Ede), (30375, Anna), (30455, Pia)\}$$

R_2	
$MatNr: D_1$	$Name: D_2$
30487	Bob
30548	Ede
30375	Anna
30455	Pia

$D_1 = \text{Integer} = \{1,2, \dots\}$,
$D_2 = \text{String}$

Relationenschemata und Relation

- **Geordnetes Relationenschema**
  - *Geordnetes einfaches Relationenschema* $R' = R(A_1, \ldots, A_k)$
    - Relationsname R
    - Liste von (voneinander verschiedenen) Attributen $A_1, \ldots, A_k$. Jedes Attribut A_i, $1 \leq i \leq k$ ist der Name einer Rolle, die von einem Wertebereich (Domain) D im *Relationenschema* R' gespielt wird..D_i wird die Domain von A_i genannt und wird mit $dom(A_i)$ bezeichnet (also $\mathrm{dom}(A_i) = D_i$).
    - Der Grad einer Relation ist die Anzahl der Attribute k ihres Relationenschemas.
- **Relation**:
  - Eine Relation r des Relationenschemas $R(A_1, \ldots, A_k)$ (auch $r(\mathrm{R}')$), ist eine Menge von k-Tupeln $\mathrm{r} = \{t_1, \ldots, t_n\}$.
  - Jedes k -Tupel t ist eine geordnete Liste von k Werten $t = (v_1, \ldots, v_k)$ wobei jeder Wert $v_i, 1 \leq i \leq k$, ein Element von $dom(A_i)$ ist oder ist.
  - Der i-te Wert im Tupel t, der dem Wert des Attribut A_i entspricht, wird als $\mathrm{t}(\mathrm{A_i})$ oder $\mathrm{t}.A_i$ bezeichnet (oder $\mathrm{t}[i]$ bei Verwendung der Positionsnotation).

- Hinweise:
  - Auf die Attributwerte eines Tupels t kann man also über die Namen zugreifen: $t(A)$ bzw. $t.A$ oder $t(B)$ bzw. $t.B$ – d.h., man kann die Attribute als Abbildungen betrachten. Im folgenden verwenden wir jeweils die einfacher anzuwendende Alternative.
  - Wenn es aus dem Kontext klar wird, verwenden wir den Buchstaben R sowohl für das Relationenschema als auch für den Namen der Relation
  - Das Relationenschema einer Relation r bezeichnen wir auch manchmal mit $\mathrm{Sch}(r)$. Wir verwenden (später) auch Audrücke der Form $(A_1 : D_1, \ldots, A_k : D_k)$ als Notation für das Schema einer Relation. Gemeint ist $\mathrm{dom}(A_i) = D_i$.
  - Wir erlauben uns einige Freiheiten bei der Verwendung von Groß- und Kleinbuchstaben. Was gemeint ist sollte eindeutig aus dem Kontext hervorgehen.

Relationen, Reihenfolgen, Datenbanken

- Reihenfolgeabhängigkeiten
  - Die Reihenfolge der Zeilen (= Tupel) spielt keine Rolle (Relation ist Menge).
  - Im geordneten Relationenschema ist die Reihenfolge der „Spalten“ wichtig:

$$\{(a, 0), (b, 1)\} \neq \{(0, a), (1, b)\}$$

- **Relation, Datenbank**
  - Eine *Relation* ist eine Ausprägung eines Relationenschemas.
  - Ein *Datenbankschema* ist eine Menge von Relationenschemata.
  - Eine *Datenbank* ist eine Menge der aktuellen Relationen (Ausprägungen)

Beispiel: Relation Städte

Städte		
Name	*Einwohner*	*Land*
München	1.211.617	Bayern
Bremen	535.058	Bremen

- Als *"geordnetes Relationenschema"*:
  - Schema: Städte(Name, Einwohner, Land)
  - dom(Name) = STRING, dom(Einwohner) = INTEGER, dom(Land) = STRING
  - Ausprägung: {(München, 1.211.617, Bayern), (Bremen, 535.058, Bremen)}

- *Tupel als Abbildung*:
  - Ausprägung: $\{t_1 , t_2\}$ mit:
    - t_1(Name) = München, t_1 (Einwohner) = 1.211.617, t_1 (Land) = Bayern
    - t_2 (Name) = Bremen, t_2 (Einwohner) = 535.058, t_2 (Land) = Bremen

Schlüssel

- Eine minimale Teilmenge der Attribute eines Relationenschemas, anhand der alle Tupel einer (möglichen) Relation unterscheidbar sind, heißt Schlüssel.

- Definition *(Schlüssel):*
  - Eine Teilmenge S der Attribute eines Relationenschemas R ist ein *Schlüssel* (genauer *Schlüsselkandidat*), wenn gilt:
    - (i) **Eindeutigkeit:** Keine Ausprägung von R kann zwei verschiedene Tupel enthalten, die sich in allen Attributen von S gleichen.
    - (ii) **Minimalität:** Keine echte Teilmenge von S erfüllt die Bedingung (i).

- Formal:
  - Sei r eine Relation über dem Schema R und $S \subseteq R$ eine Teilmenge der Attribute von R; $t[S]$ bezeichne die Einschränkung (Projektion) des Tupels t auf die Attribute in S.
  - S ist ein Schlüssel der Relation r, wenn gilt:
    - (i) Für alle möglichen Ausprägungen r und Tupel $t_1, t_2 \in r$ gilt: $t_1 \neq t_2 \Rightarrow t_1[S] \neq t_2[S]$
    - (ii) Für alle Attributmengen T, die (i) erfüllen, gilt: $T \subseteq S \Rightarrow T = S$

Beispiel für Schlüssel

- *Wichtig:*
  - Die Schlüsseleigenschaft ist abhängig von der Semantik des Schemas, nicht von der aktuellen Ausprägung einer Relation!

- *Beispiel*: Relation mit Personalnummer und Gehalt

Personal	
PNr	*Gehalt*
1	1.700 €
2	2.172 €
3	3.189 €
4	2.167 €

  - In der Relation *Personal* erfüllen sowohl die Attributmengen $\{PNr\}$ als auch $\{Gehalt\}$ die Bedingung (*i*). Schlüssel ist jedoch nur $\{PNr\}$, da Personalnummern logisch stets eindeutig sind, Gehälter jedoch nicht.
  - Auch $\{PNr, Gehalt\}$ erfüllt Bedingung (*i*), kann jedoch auf $\{PNr\}$ reduziert werden und ist deshalb kein Schlüssel.

- Gelegentlich gibt es mehrere Schlüsselkandidaten für eine Relation, unter denen dann einer als *Primärschlüssel* ausgewählt wird (üblicherweise der mit den kürzesten Attributswerten).
- Wie bereits im ER-Diagramm werden die zum Schlüssel gehörigen Attribute im Schema unterstrichen: Mitarbeiter (<u>PNr</u>, Gehalt)

Das Relationale Datenmodell

1. Das Datenmodell
2. **Transformation von ER-Diagrammen in das Relationale Modell**
3. Relationale Algebra
4. Relationaler Kalkül

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Datenabhängigkeiten

- Im relationalen Modell:
  - Objekte (Entities) und alle Arten von Beziehungen (Relationships) werden durch Relationen dargestellt
- keine Einschränkungen: an den Relationen können beliebige Mengen von Attributen beliebiger Objekte beteiligt sein
  - auch „ungültige“ Kombinationen können auftreten
- Integritätsbedingungen um Konsistenz der Datenbank zu gewährleisten:
  - intra- und interrelationale Abhängigkeiten
  - wichtig zur Modellierung von 1:n-, Is-A-Beziehungen, uvm.

Relationenschema

- Intrarelationale Abhängigkeiten
  - Beispiel: Schlüssel einer Relation
  - Erlaubt Aussagen über die Konsistenz/Gültigkeit einer Relation
    - Ein Schlüssel einer Relation darf in der Relation nicht doppelt vorkommen

- Formal: Erweiterung der Relationenschemata
  - Es bezeichne X die Menge der Attribute eines geordneten Relationschemas R'
  - $\mathrm{Tup}(X)$ bezeichnet die Menge aller Tupel über X
  - $\mathrm{Rel}(X) = \{ r \mid r \subseteq \mathrm{Tup}(X) \}$ ist die Menge aller Relationen über X
  - eine intrarelationale Abhängigkeit über X ist eine Abbildung

    $$\sigma : \mathrm{Rel}(X) \to \{\mathbf{true}, \mathbf{false}\},$$

    formal spezifiziert sie, welche der möglichen Relationen eine gegebene Bedingung erfüllen

Relationenschema

- Intrarelationale Abhängigkeiten
  - Beispiel: Schlüssel einer Relation
  - Sei $K \subseteq X$. Eine Schlüsselabhängigkeit σ_K bezeichnet folgende intrarelationale Abhängigkeit

$$\sigma_K : \mathrm{Rel}(X) \to \{true, false\}, \qquad r \mapsto \begin{cases} true, & \text{falls } K \text{ Schlüssel von } r \text{ ist} \\ false, & \text{sonst} \end{cases}$$

- $R'' = (R', \Sigma_X)$ ist ein erweitertes *Relationenschema*, dabei sind
  - R' ein geordnetes Relationenschema
  - Σ_X eine Menge intrarelationaler Abhängigkeiten (Schlüsselabhängigkeiten) über X, dabei ist X die Menge der Attribute in R'
  - eine Relation r mit erweitertem Relationenschema $R'' = (R', \Sigma_X)$ heißt *gültig* oder *konsistent*, falls sie alle intrarelationalen Abhängigkeiten Σ_X erfüllt

Datenbankschema

- Interrelationale Abhängigkeiten
  - Beispiel 2 Tabellen:
    - Student(MatrNr, Name, Semester)
    - Hört(MatrNr, VorlNr)
    - $\text{Hört}[\text{MatrNr}] \subseteq \text{Student}[\text{MatrNr}]$
  - Erlaubt Aussagen über die Konsistenz/Gültigkeit einer Datenbank
    - jede Matrikelnummer die in der Relation Hört enthalten ist, muss auch in einem Eintrag in Student existieren

- Formal: Erweiterung der Datenbankschemata
  - Es bezeichne $\mathcal{R} = \{R_1, \dots, R_k\}$ eine endliche Menge von erweiterten Relationenschemata
  - Eine Datenbank $D = \{r_1, \dots, r_k\}$ ist eine Menge von Relationen (Ausprägungen)
  - $\text{Dat}(\mathcal{R})$ bezeichnet die Menge aller Datenbanken über $\mathcal{R}$
  - eine interrelationale Abhängigkeit über $\mathcal{R}$ ist eine Abbildung

$$\sigma : \text{Dat}(\mathcal{R}) \to \{\mathbf{true}, \mathbf{false}\},$$

formal spezifiziert sie, welche der möglichen Datenbanken die gegebene Bedingung erfüllen

Datenbankschema

- Interrelationale Abhängigkeiten
  - Beispiel: $\text{Hört}[\text{MatrNr}] \subseteq \text{Student}[\text{MatrNr}]$
  - Inklusionsabhängigkeit: Sei $\mathcal{R} = \{R_1, \dots, R_k\}$ eine endliche Menge von Relationenschemata

    und $V \subseteq X_i, W \subseteq X_j$,$|V| = |W|, u: V \mapsto W$ eine bijektive Abbildung sowie $r_i \in Rel(X_i), r_j \in Rel(X_j)$.

Eine Inklusionsabhängigkeit:

$$R_i[V] \subseteq R_j[W]$$

bezeichnet folgende interrelationale Abhängigkeit

$$\sigma_{R_i[V] \subseteq R_j[W]} : \text{Dat}(\mathcal{R}) \to \{true, false\}, \qquad D \longmapsto \begin{cases} true, & \text{falls } \{ t[V] \mid t \in r_i \} \subseteq \{ s[W] \circ u \mid s \in r_j \} \\ false, & \text{sonst} \end{cases}$$

Anschaulich: jede Matrikelnummer die in der Relation Hört enthalten ist, muss auch in einem Eintrag in Student existieren. u ist bei gleichen Attributnamen die Identitätsfunktion, ansonsten wird die Definition von u aus dem Kontext klar, da u die Attributnamen aufeinander abbildet.

Beachte: Attribute werden hier als Abbildungen behandelt – daher die Verkettung mit u!

Datenbankschema

- Interrelationale Abhängigkeiten
  - Beispiel: $\text{Assistent}[\text{PersNr}] \cap \text{Professor}[\text{PersNr}] = \emptyset$
  - Zweiter Typ, Exklusionsabhängigkeit: Sei wieder $\mathcal{R} = \{R_1, \ldots, R_k\}$ eine endliche Menge von Relationenschemata, $V \subseteq X_i, W \subseteq X_j$,$|V| = |W|, u: V \mapsto$ W eine bijektive Abbildung sowie $r_i \in Rel(X_i), r_j \in Rel(X_j)$ .Eine Exklusionsabhängigkeit

$$R_i[V] \cap R_j[W] = \emptyset$$

bezeichnet die Abhängigkeit

$$\sigma_{R_i[V] \cap R_j[W] = \emptyset} : \text{Dat}(\mathcal{R}) \to \{true, false\}, D \mapsto \begin{cases} true, \text{falls } \{ t[V] \mid t \in r_i \} \cap \{ s[W] \circ u \mid s \in r_j \} = \emptyset \\ false, \text{sonst} \end{cases}$$

Datenbankschema

- $D = (\mathcal{R}, \Sigma_{\mathcal{R}})$ ist ein *Datenbankschema*, dabei sind
  - $\mathcal{R}$ eine Menge von Relationenschemata
  - $\Sigma_{\mathcal{R}}$ eine Menge interrelationaler Abhängigkeiten über $\mathcal{R}$
- Eine Datenbank $\mathrm{d} \in Dat(\mathcal{R})$ mit Datenbankschema $D = (\mathcal{R}, \Sigma_{\mathcal{R}})$ heißt *konsistent* oder *gültig*, falls alle intra- und interrelationalen Abhängigkeiten erfüllt sind

Fremdschlüssel

- Inklusionsabhängigkeiten: Modellierung von $1{:}n$ – und Is-A – Beziehungen

- Sei $D = (\mathcal{R}, \Sigma_{\mathcal{R}})$ ein Datenbankschema. Eine Attributmenge $F \subseteq X$ der Attributmenge eines Relationenschemas $R_1 \in \mathcal{R}$ heißt Fremdschlüssel, bezogen auf eine Relation r_2, falls
  - F den selben Wertebereich besitzt wie der Schlüssel von r_2
  - die interrelationale Abhängigkeit $R_1[F] \subseteq R_2[K]$, wobei K der Schlüssel von r_2 ist, gilt in D, d.h.

    $\sigma_{R_1[F] \subseteq R_2[K]} \in \Sigma_{\mathcal{R}}$

- Die zweite Bedingung nennt man auch *referentielle Integrität* des Fremdschlüssels
- Anschaulich: jedes Tupel in den Fremdschlüsselattributen verweist auf ein existierendes Tupel der referenzierten Relation oder die Fremdschlüsselattribute sind auf Null gesetzt

ER Modell und Relationales Modell

- Ausgangslage nach konzeptueller Entwurf: ER-Modell
  - Entity-Typen
  - Beziehungs-Typen

- Wo wollen wir hin: Relationales Modell
  - nur ein Strukturierungskonzept: Relationen

Unsere Aufgaben:

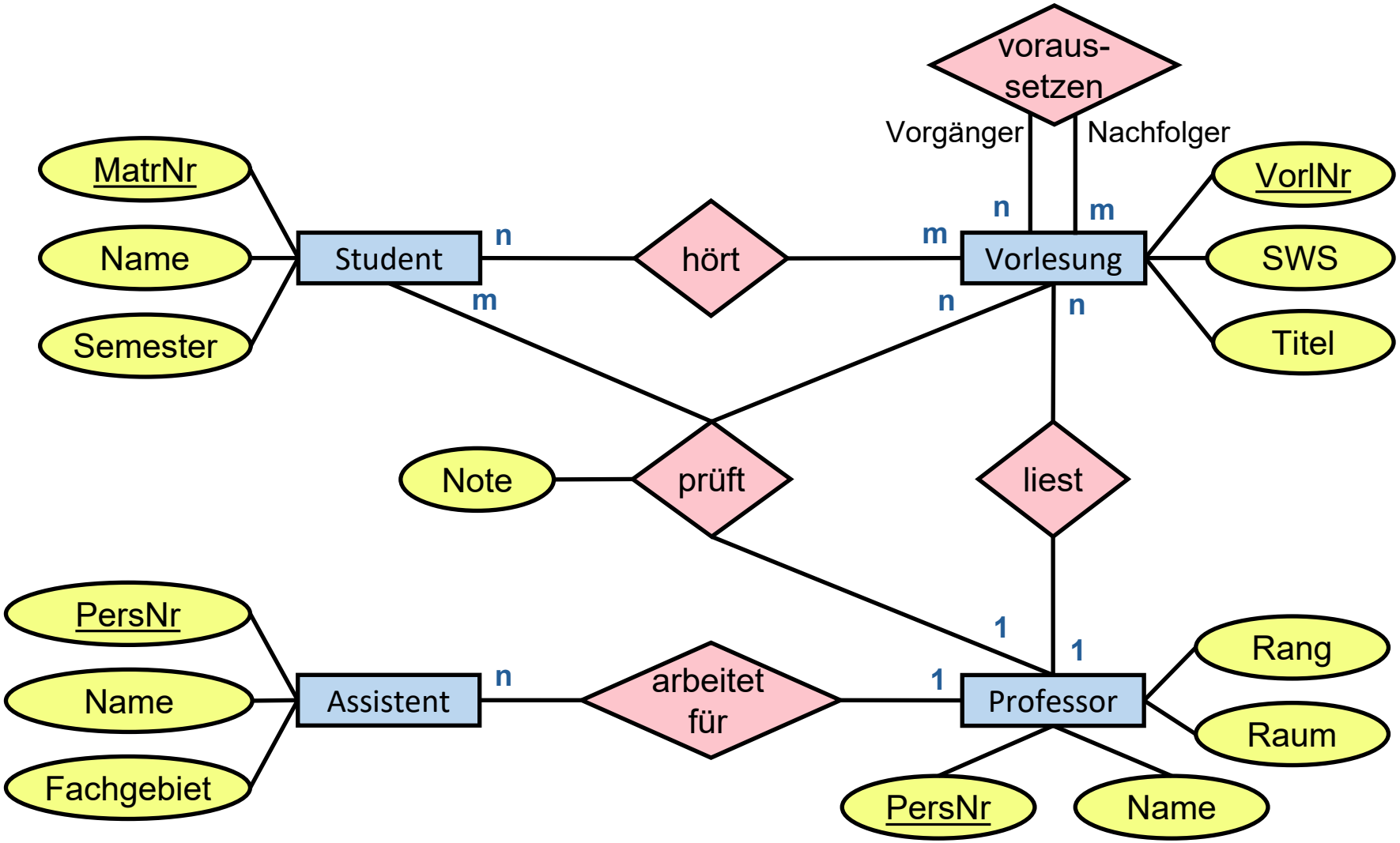
- Entities und Beziehungen müssen auf Relationen abgebildet werden
- Intrarelationale und interrelationale Abhängigkeiten ableiten

Unsere Todo-Liste

Wir müssen abbilden:

- Entitytypen mit Attributen
- Beziehungen
  - n:m, 1:n, 1:1
  - Rekursive Beziehungen
- Generalisierung / Spezialisierung
  - Partiell, total
- Spezialfälle
  - Schwache Entity-Typen
  - n-stellige Beziehungen

Beispiel : ER-Diagramm (Universität)



Unsere Todo-Liste

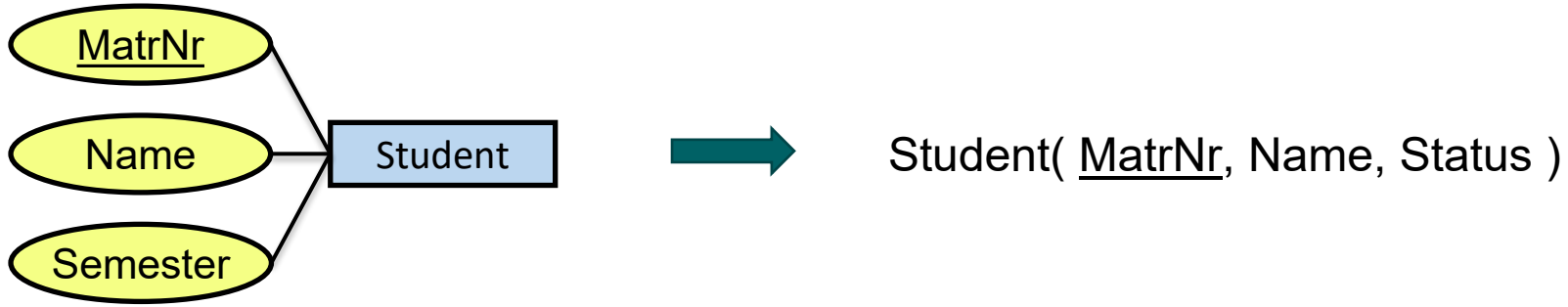
Wir müssen abbilden:

- **Entitytypen mit Attributen**
- Beziehungen
  - n:m, 1:n, 1:1
  - Rekursive Beziehungen
- Generalisierung / Spezialisierung
  - Partiell, total
- Spezialfälle
  - Schwache Entity-Typen
  - n-stellige Beziehungen

Entitytyp mit Attributen

- Jeder Entity-Typ wird zu einer Tabelle/Relation mit entsprechenden Attributen

- Spezialfälle
  - Zusammengesetzte Attribute
    - Jedes (Teil-) Attribut als einzelnes Attribut
    - Oder das zusammengesetzte Attribute als Ganzes
  - Mehrwertige Attribute
    - Neue Relation mit zusammengesetztem Schlüssel (bestehend aus eigenem Schlüssel und Fremdschlüssel auf den Entity-Typ)
    - Ähnlich zu einer 1:n – Beziehung



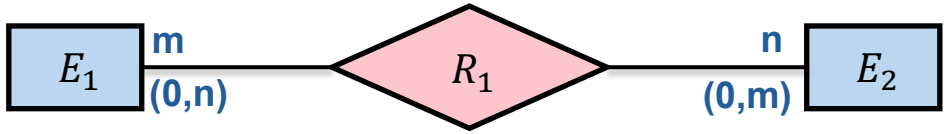
Unsere Todo-Liste

Wir müssen abbilden:

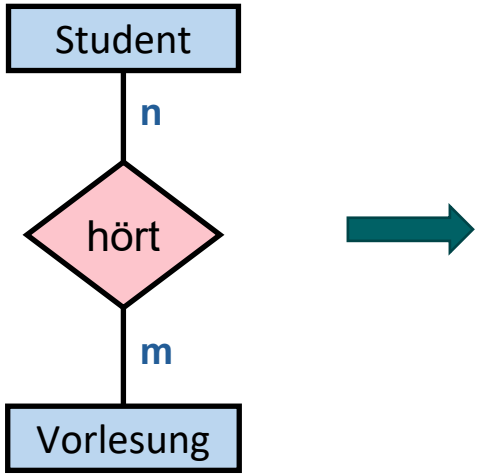
- Entitytypen mit Attributen
- **Beziehungen**
  - n:m, 1:n, 1:1
  - Rekursive Beziehungen
- Generalisierung / Spezialisierung
  - Partiell, total
- Spezialfälle
  - Schwache Entity-Typen
  - n-stellige Beziehungen

n:m – Beziehungen

- werden durch eigene Relationen dargestellt



  - Schlüssel: besteht aus zwei Fremdschlüsseln auf die Relationen E_1 und E_2

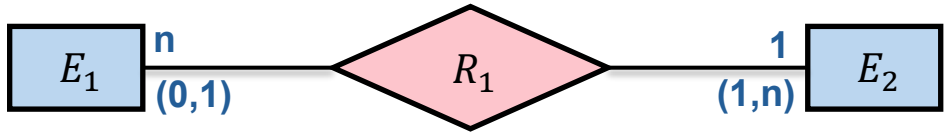


Entitäten:
Student(<u>MatrNr</u>)
Vorlesung(<u>VorlNr</u>)
Hört(<u>MatrNr</u>, <u>VorlNr</u>)

Interrelationale Abhängigkeiten:
Hört[MatrNr] ⊆ Student[MatrNr]
Hört[VorlNr] ⊆ Vorlesung[VorlNr]

1:n – Beziehungen

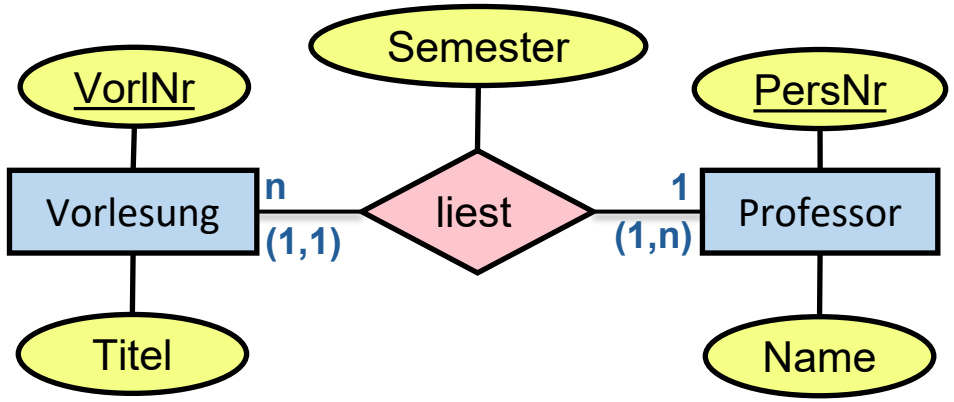
- 1:n – Spezialfall von n:m
  - Schlüssel der Relation ist ein Fremdschlüssel auf E_1 (E_2 ist dann durch E_1 bestimmt)



- Praktikabel falls nur wenige Objekte aus E_1 an R_1 teilnehmen, da die hinzugefügte Relation R_1 dann klein ist.

1:n – Beziehungen

- Effizienter: die Beziehung wird in die Relation des n-Entity-Typs integriert
  - E_1 bekommt Schlüssel von E_2 als Fremdschlüssel



Professor(PersNr, Name)

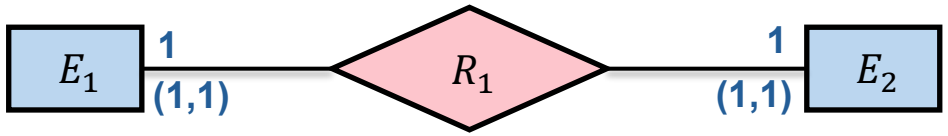
Vorlesung(VorlNr, Titel, Semester,
ProfessorPersNrLiest)

Vorlesung[ProfessorPersNrLiest] ⊆
Professor[PersNr]

  - Vorteil: Eine zusätzliche Relation ist nicht notwendig
  - Nachteil: falls nur wenige Objekte aus E_1 an R_1 teilnehmen enthält E_2 viele null Werte
    - im Beispiel: falls Vorlesungen meistens nicht von einem Professor gehalten werden (andere (min,max)-Notation nötig)

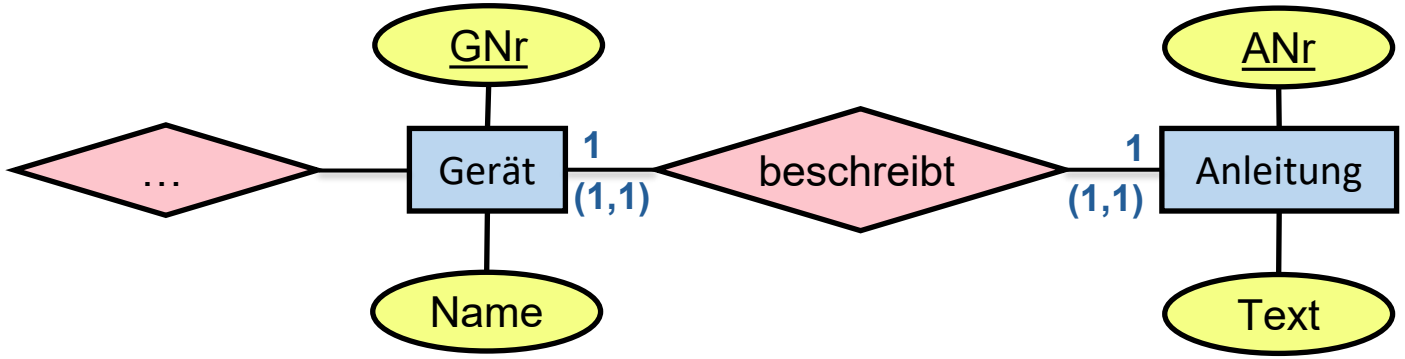
1:1 – Beziehungen

- Mögliche Lösungen:
  - wie 1:n – Beziehung, Fremdschlüssel in E_1 verweist auf E_2 oder eigene Relation



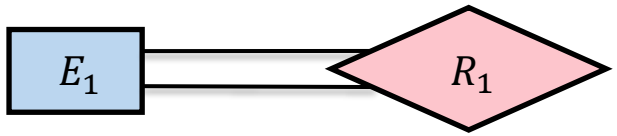
  - Effizienter: Verschmelzen der beteiligten Relationen zu einer einzigen
    - nur falls Teilnahme an R_1 total ist und E_1 und/oder E_2 nicht an anderen Beziehungen teilnehmen

- Beispiel:



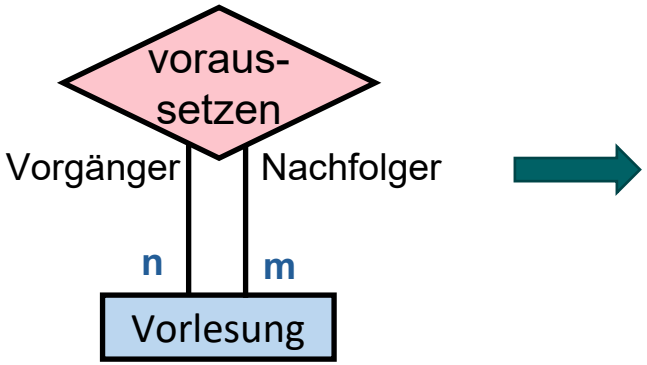
➡ Gerät(GNr, Name, ANr, Text)

Rekursive Beziehungen



- Möglichkeit für 1:1 und 1:n – Beziehung:
  - E_1 hat Fremdschlüssel auf sich selbst
  - Attribute von R_1 werden zu E_1 hinzugefügt

- m:n – Beziehung:
  - eigene Relation für R_1



Voraussetzen(<u>Vorgänger</u>, <u>Nachfolger</u>)

Voraussetzen[Vorgänger] ⊆ Vorlesung[VorlNr]

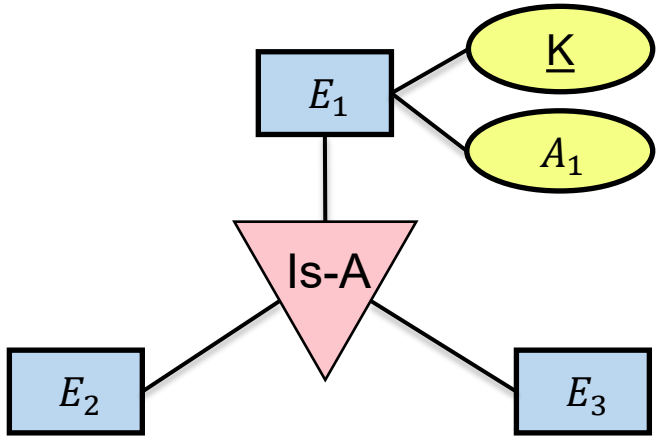
Voraussetzen[Nachfolger] ⊆ Vorlesung[VorlNr]

Unsere Todo-Liste

Wir müssen abbilden:

- Entitytypen mit Attributen
- Beziehungen
  - n:m, 1:n, 1:1
  - Rekursive Beziehungen
- **Generalisierung / Spezialisierung**
  - Partiell, total
- Spezialfälle
  - Schwache Entity-Typen
  - n-stellige Beziehungen

Generalisierung / Spezialisierung



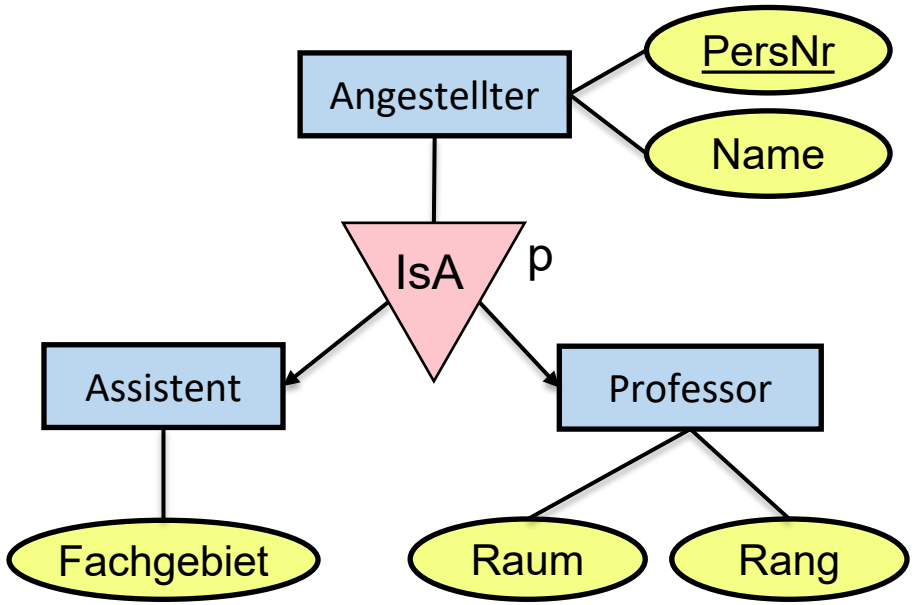
- jede Spezialisierung besitzt einen Fremdschlüssel auf die Generalisierung
  - ist die Beziehung disjunkt: Exklusionsbeziehungen
  - ist die Beziehung zusätzlich total: keine eigene Relation für E_1

- Nachteil (außer bei totalen, disjunkten Beziehungen):
  - Volle Informationen zu E_2 und E_3 können nur durch join mit E_1 abgerufen werden
  - Views/Sichten relationaler DBMS verschaffen Abhilfe

Möglichkeiten zur Übersetzung von isA-Beziehungen

- Erzeuge Tabellen für **jeden** Entity-Typ
  - Für partielle isA-Beziehungen
- Erzeuge **nur** Tabellen **für Subtypen**
  - Bei totalen isA-Beziehungen
- Erzeuge **eine einzige Tabelle**, die alle Attribute aller Entity-Typen enthält, sowie ein **Typ-Attribut**
  - Für disjunkte isA-Beziehungen
- Erzeuge **eine einzige Tabelle**, die alle Attribute aller Entity-Typen enthält, sowie ein **boole'sches Typ-Attribut** für jeden der Entity-Typen
  - falls die Subtypen überlappen (nicht disjunkt)

Generalisierung / Spezialisierung (Partielle Beziehung)

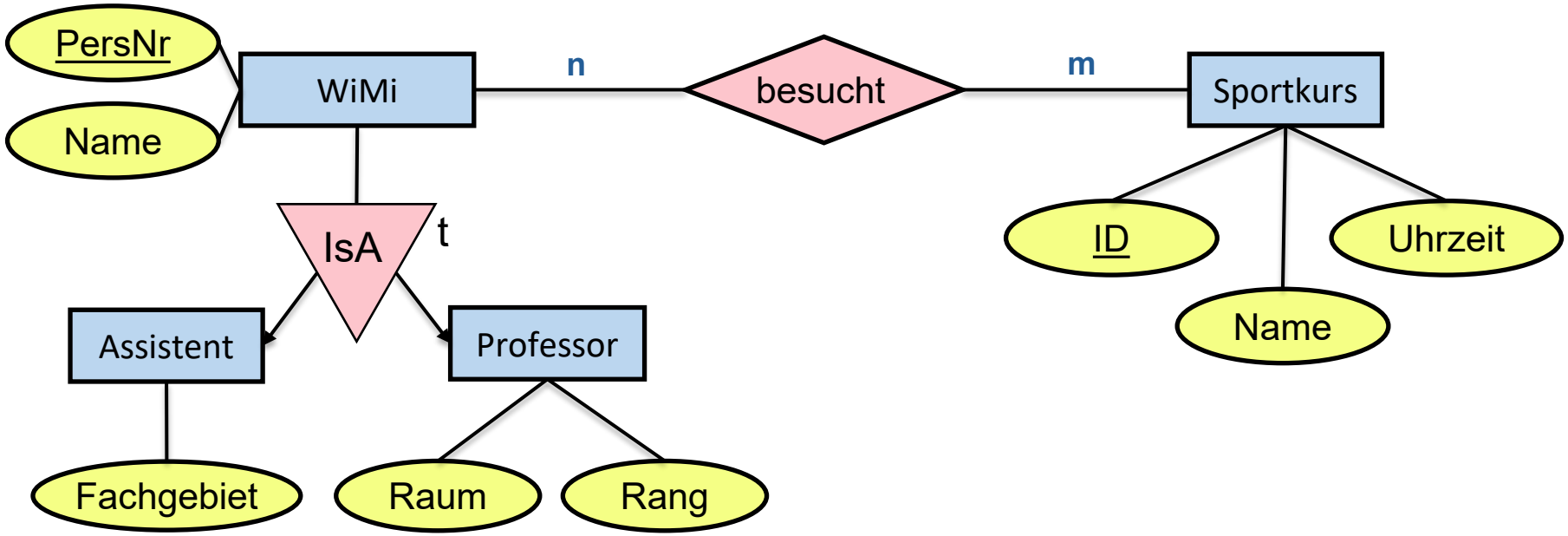


Angestellter(PersNr, Name)
Professor(PersNr, Rang, Raum)
Assistent(PersNr, Fachgebiet)

$Professor[PersNr] \subseteq Angestellter[PersNr]$
$Assistent[PersNr] \subseteq Angestellter[PersNr]$

$Professor[PersNr] \cap Assistent[PersNr] = \emptyset$

Generalisierung / Spezialisierung (totale Beziehung)



Professor(PersNr, Name, Raum, Rang)
Assistent(PersNr, Name, Fachgebiet)
Sportkurs(ID, Name, Uhrzeit)
Besucht(PersNr, SportkursID)

$Professor[PersNr] \cap Assistent[PersNr] = \emptyset$

$Besucht[SportkursID] \subseteq Sportkurs[ID]$
$Besucht[PersNr] \subseteq Assistent[PersNr] \cup Professor[PersNr]$

Unsere Todo-Liste

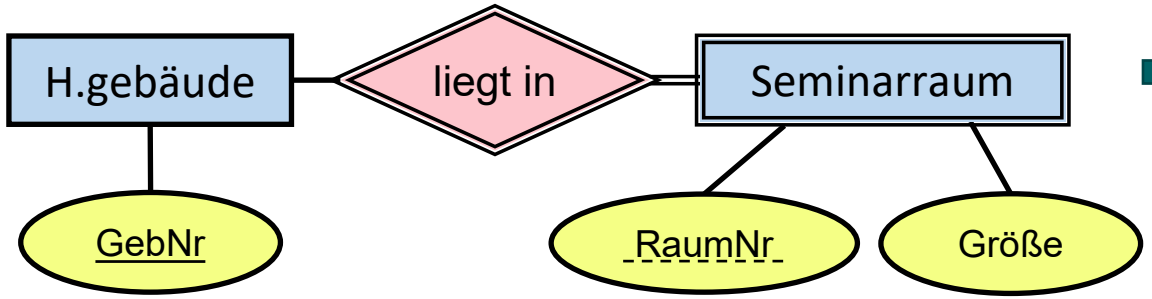
Wir müssen abbilden:

- Entitytypen mit Attributen
- Beziehungen
  - n:m, 1:n, 1:1
  - Rekursive Beziehungen
- Generalisierung / Spezialisierung
  - Partiell, total
- **Spezialfälle**
  - Schwache Entity-Typen
  - n-stellige Beziehungen

Schwache Entity-Typen

- Schwacher Entity-Typ E_1 mit
  - Teilschlüssel K_1
  - Übergeordneter Entity-Typ E_2 mit Schlüssel K_2

- Transformation
  - Relationenschema für E_1: E_1($\underline{K_1}$, $\underline{K_2}$, ...)
  - Schlüssel wird gebildet aus K_1 und K_2
  - K_2 ist ein Fremdschlüssel auf E_1



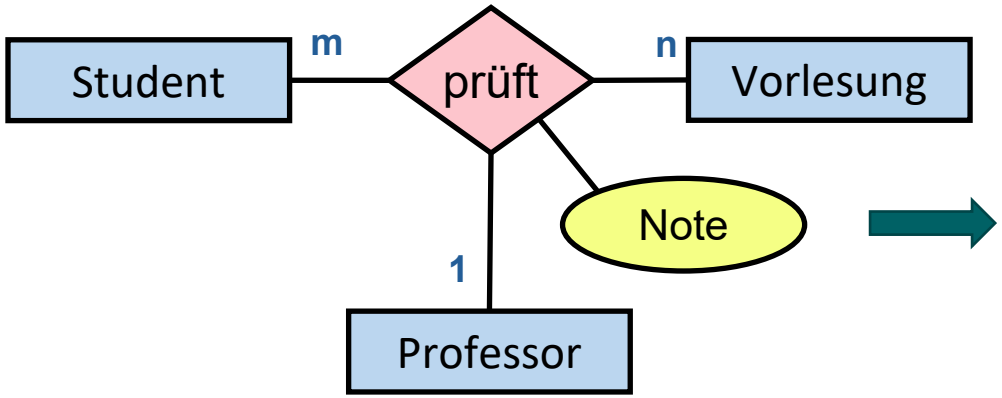
Seminarraum($\underline{\text{RaumNr}}$, $\underline{\text{GebNr}}$, Größe)

Seminarraum[GebNr] ⊆ H.gebäude[GebNr]

n-stellige Beziehungen

- können wie üblich durch eine eigene Relation dargestellt werden
  - Kardinalitäten geben an, wie sich der Schlüssel zusammensetzt

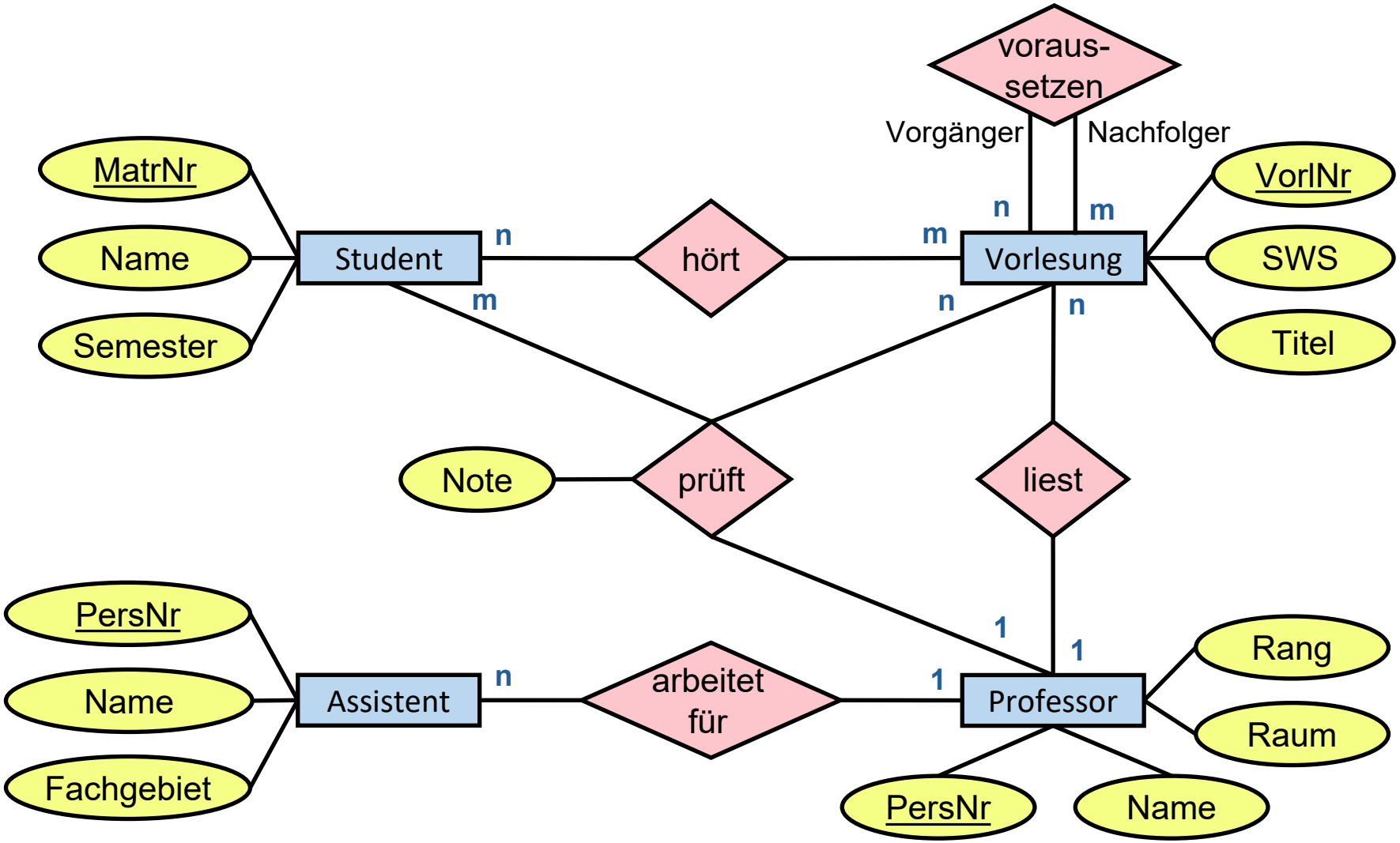
- Beispiel:



Prüft(<u>MatrNr</u>, <u>VorlNr</u>,
PersNr, Note)

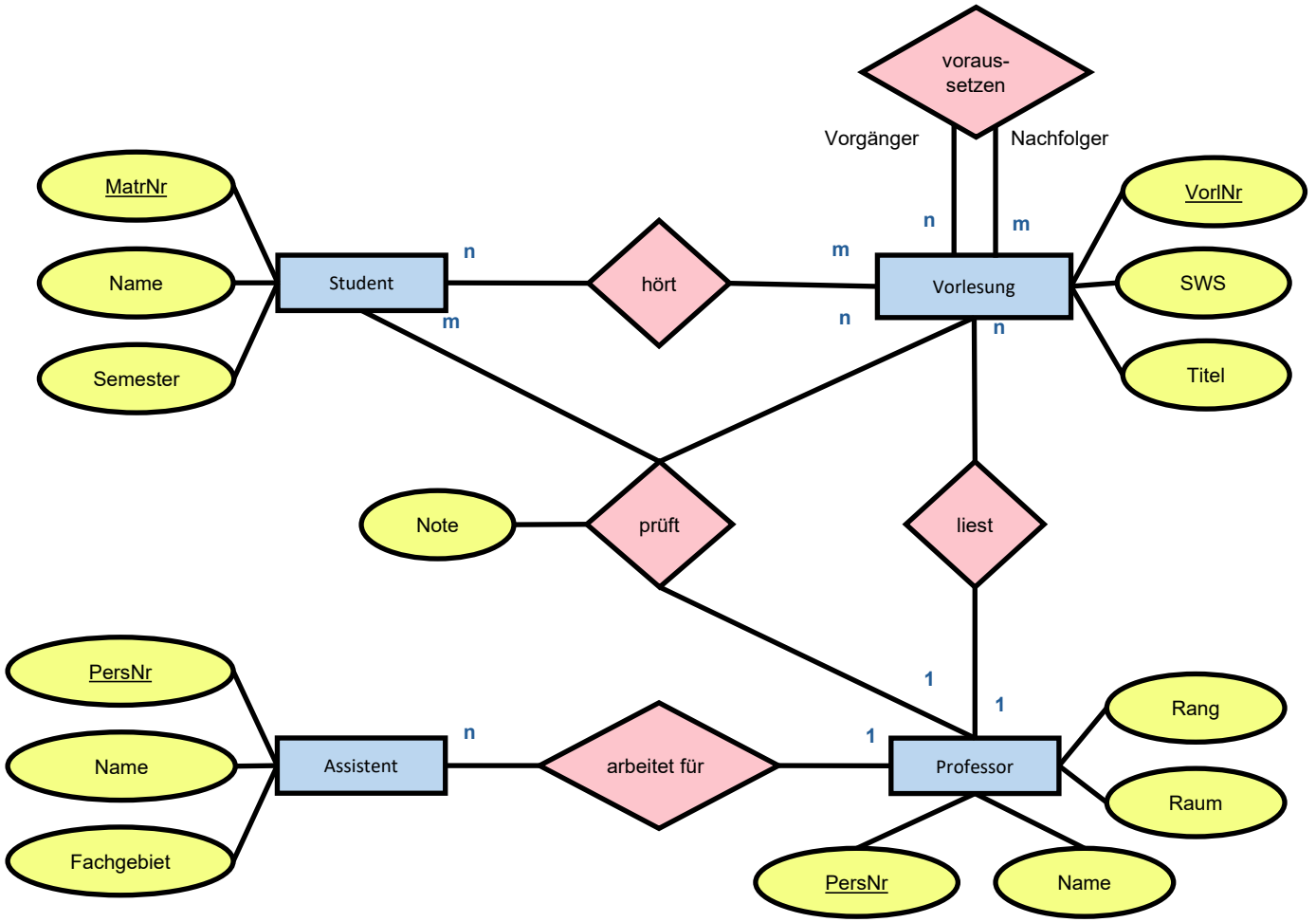
Prüft[MatrNr] ⊆ Student[MatrNr]
Prüft[VorlNr] ⊆ Vorlesung[VorlNr]
Prüft[PersNr] ⊆ Professor[PersNr]

Beispiel : ER-Diagramm (Universität)



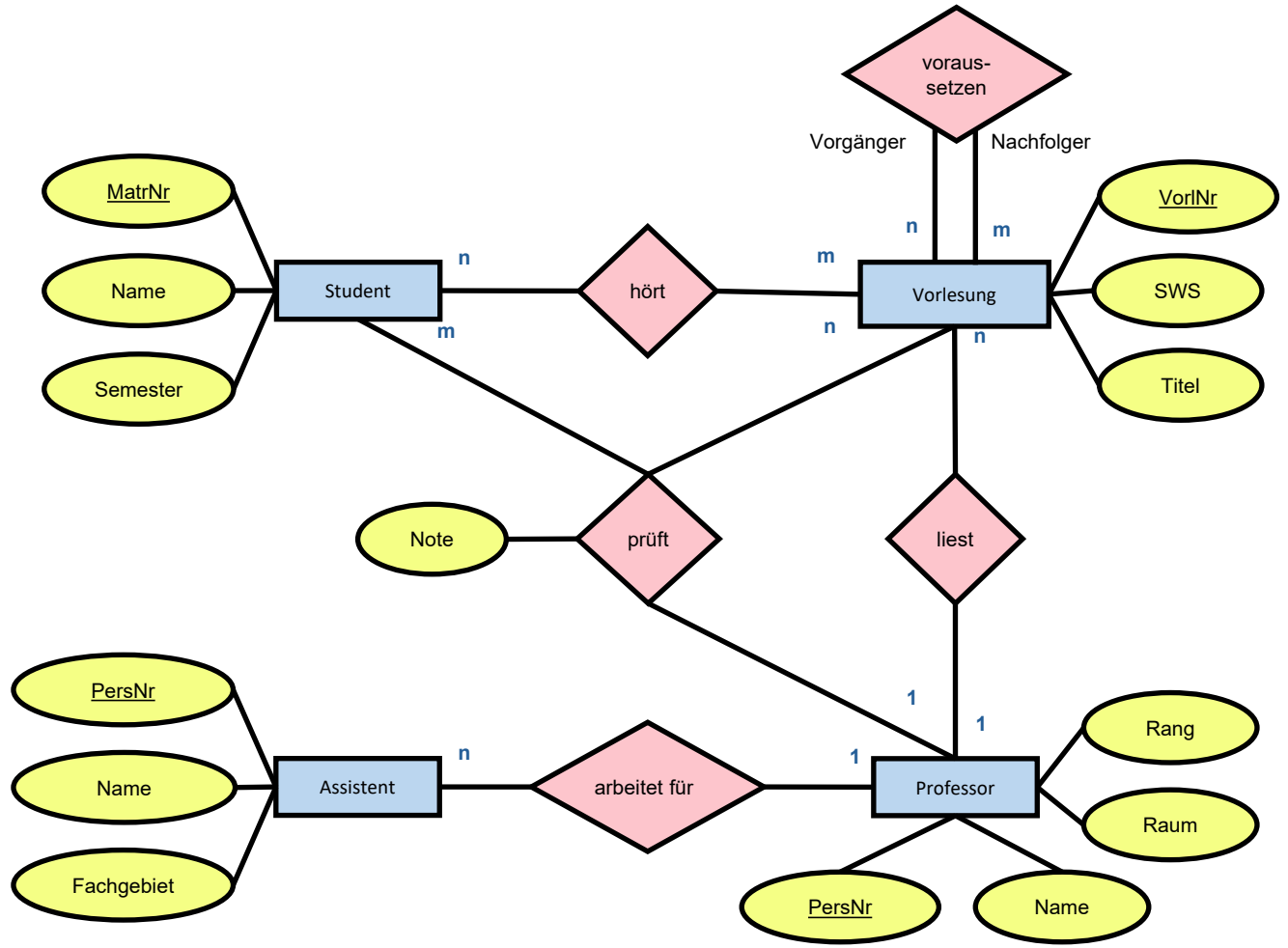
Beispiel für ein Datenbankschema

- **Relationsschema R:**
  - Student(MatrNr, Name, Semester)
  - Professor(PersNr, Name, Rang, Raum)
  - Vorlesung(VorlNr, Titel, SWS, gelesenVon)
  - Assistent(PersNr, Name, Fachgebiet, Boss)
  - Voraussetzen(Vorgänger, Nachfolger)
  - Hört(MatrNr, VorlNr)
  - Prüft(MatrNr, VorlNr, PersNr, Note)



Beispiel für ein Datenbankschema

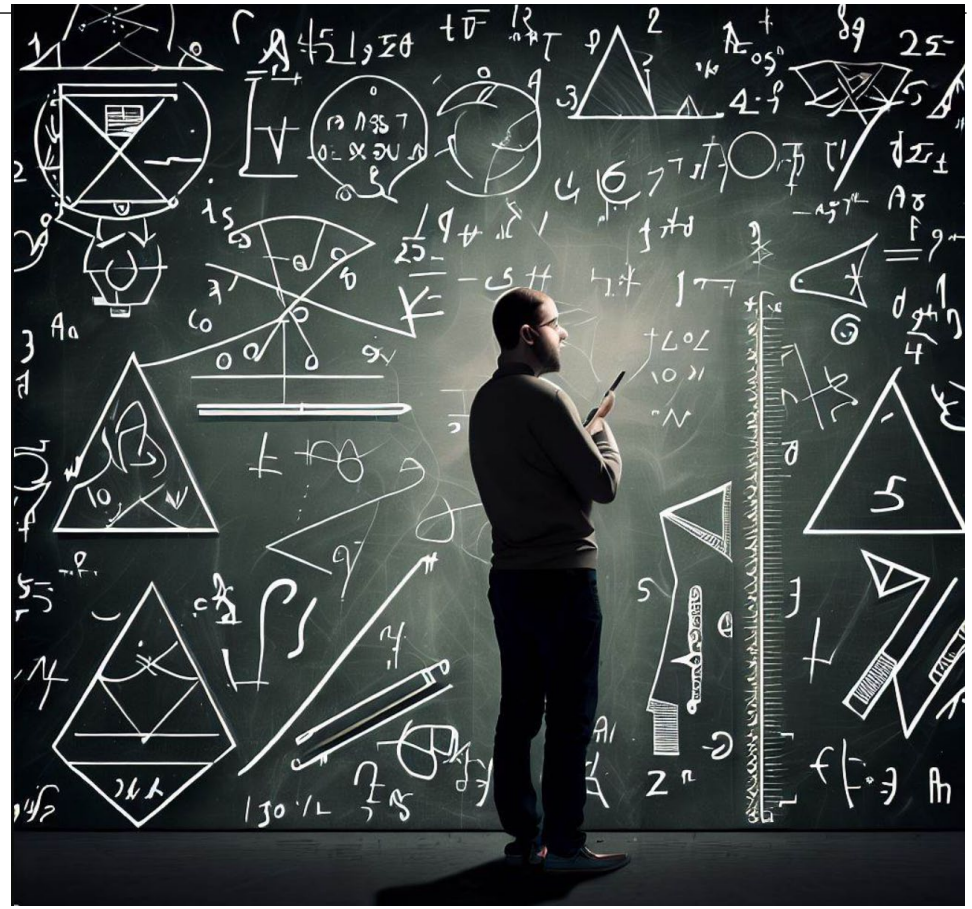
- **Interrelationale Abhängigkeiten:**
  - Vorlesung[gelesenVon] ⊆ Professor[PersNr]
  - Assistent[Boss] ⊆ Professor[PersNr]
  - Voraussetzen[Vorgänger] ⊆ Vorlesung[VorlNr]
  - Voraussetzen[Nachfolger] ⊆ Vorlesung[VorlNr]
  - Hört[MatrNr] ⊆ Student[MatrNr]
  - Hört[VorlNr] ⊆ Vorlesung[VorlNr]
  - Prüft[MatrNr] ⊆ Student[MatrNr]
  - Prüft[VorlNr] ⊆ Vorlesung[VorlNr]
  - Prüft[PersNr] ⊆ Professor[PersNr]
  - Assistent[PersNr] ∩ Professor[PersNr] = ∅



3. Das Relationale Datenmodell

1. Das Datenmodell
2. Transformation von ER-Diagrammen in das Relationale Modell
3. **Relationale Algebra**
4. Relationaler Kalkül

Was ist eine Algebra?



Definition:
- Mathematisches System
- Kombination von Symbolen und Regeln
- Darstellung und Manipulation abstrakter Objekte

Komponenten einer Algebra:
- Menge von Elementen (z.B., Zahlen, Vektoren, Matrizen)
- Operationen (z.B., Addition, Multiplikation, Subtraktion)
- Axiome (z.B., Assoziativität, Kommutativität, Distributivität)

Anwendungen:
- Mathematik (z.B., Lineare Algebra, Boolesche Algebra)
- Informatik (z.B., Relationale Algebra in Datenbanken)
- Physik und Ingenieurwissenschaften (z.B., Tensoralgebra, Lie-Algebra)

Warum Relationale Algebra?

- Teilung der Aufgaben (erste Vorlesung: Schichtenbildung!):
  - Verwendung der Algebra: z.B., Entwurf von Abfragesprachen mit relationalen Algebraoperatoren
  - Implementierung der Algebra
    - Effiziente Algorithmen und Datenstrukturen für Algebraoperationen
      - Zugriffsmethoden (z.B., B-Bäume, Hash-Indizes, Bitmap-Indizes)
      - Abfrageoptimierung (Kostenabschätzung, Auswahl der besten Pläne)
      - Abfrageausführung (z.B., geschachtelte Schleifenverknüpfungen, Hash-Verknüpfungen, Sortierungs-Zusammenführungsverknüpfungen)
      - Nebenläufigkeitskontrolle (Datenkonsistenz und Integrität)
      - Wiederherstellung (z.B., Protokollierung und Kontrollpunkte)

Relationale Algebra

- Eine Algebra ist allgemein gegeben durch
  - Menge von Objekten (Wertebereich)
  - Operationen zur Verknüpfung von Objekten

- Relationale Algebra
  - *Relationen* als Wertebereich
  - *relationale Operationen*, die Relationen als Argumente haben und wiederum Relationen als Resultate liefern

- Anfragen
  - Formulierung von Anfragen als Ausdrücke der relationalen Algebra, d.h. durch (rekursive) Anwendung von Operationen auf Relationen. Die Bearbeitung der Anfragen ist durch tatsächliche Auswertung der Operationen auf den Relationen einer Datenbank möglich.
  - Maß für die Ausdruckskraft einer Anfragesprache:
    - Sprache L heißt relational vollständig: jeder Ausdruck der Relationenalgebra kann in L simuliert werden kann (es gibt Ausdrücke in L, mit dem selben Ergebnis)
    - Eine Sprache L heißt streng relational vollständig: jeder Ausdruck der Relationenalgebra kann durch einen einzigen Ausdruck von L simuliert

Sechs Grundoperationen der Relationalen Algebra

Für die relationale Algebra gibt es sechs Grundoperationen, aus denen sich alle anderen Operationen nachbilden lassen:

- Vereinigung
- Differenz
- Kartesisches Produkt
- Selektion
- Projektion
- Umbenennung

Unsere Aufgabe im folgenden:

- Identifikation der Voraussetzungen für die Anwendung der Operatoren hat
- Identifizieren was die Operatoren mit den Tupeln **und** dem Schema der Relationen machen

Sechs Grundoperationen - Vereinigung

- *Vereinigung* $R \cup S$
  - Die beiden Relationenschema $R(A_1, \ldots, A_k)$ und $S(A_1, \ldots, A_k)$ müssen die gleichen Attribute besitzen. Dann ist $R \cup S$ definiert als die mengentheoretische Vereinigung:

$$R \cup S = \{t \mid t \in R \vee t \in S\}$$

  - Das neue Relationenschema ist $Q(A_1, \ldots, A_k)$. Dabei ist Q ein neuer Relationenname.
  - Anschaulich: Packe alle Tupel zusammen in eine Relation (lösche dabei Duplikate)
  - Beispiel: $\{(a, 0), (a, 1), (b, 1)\} \cup \{(a, 2), (b, 1)\} = \{(a, 0), (a, 1), (a, 2), (b, 1)\}$

Sechs Grundoperationen - Differenz

- *Differenz* $R - S$
  - Die beiden Relationenschema $R(A_1, \ldots, A_k)$ und $S(A_1, \ldots, A_k)$ müssen die gleichen Attribute besitzen. Dann ist $R - S$ definiert als die mengentheoretische Differenz:

$$R - S = \{t \mid t \in R \wedge t \notin S\}$$

  - Das neue Relationenschema ist $Q(A_1, \ldots, A_k)$. Dabei ist Q ein neuer Relationenname.
  - Anschaulich: Nimm alle Tupel aus R heraus, die es auch in S gibt.
  - Beispiel: $\{(a, 0), (a, 1), (b, 1)\} - \{(a, 1), (b, 2)\} = \{(a, 0), (b, 1)\}$

Sechs Grundoperationen – Kartesisches Produkt

- *Kartesisches Produkt* $R \times S$ (Kreuzprodukt)
  - Seien r und s der Grad von R bzw. S. Die Attributmenge der Relationenschema $R(A_1, \ldots, A_r)$ und $S(B_1, \ldots, B_s)$ seien disjunkt.
    Dann ist $R \times S$ die Menge aller $(r+s)$-Tupel, deren erste r Komponenten ein Tupel in R und deren letzte s Komponenten ein Tupel in S bilden, d.h.

$$R \times S = \{(a_1, \ldots, a_r, b_1, \ldots, b_s) \mid (a_1, \ldots, a_r) \in R \wedge (b_1, \ldots, b_s) \in S\}$$

  - Das neue Relationenschema ist $Q(A_1, \ldots, A_r, B_1, \ldots, B_s)$. Dabei ist Q ein neuer Relationenname.
  - Für die Anzahl der Tupel gilt: $|R \times S| = |R| \cdot |S|$
  - Anschaulich: verknüpfe jedes Tupel aus R mit jedem Tupel aus S
  - Beispiel: $\{(a,0),(b,1)\} \times \{(c,h),(d,v)\} = \{(a,0,c,h),(a,0,d,v),(b,1,c,h),(b,1,d,v)\}$

Sechs Grundoperationen – Selektion

- *Selektion* $\sigma_{\boldsymbol{F}}(R)$
  - Mit der Selektion $\sigma_{\boldsymbol{F}}(R)$ werden diejenigen Tupel aus der Relation R mit *Relationenschema* $R(A_1, \ldots, A_k)$ ausgewählt, die eine durch die logische Formel $\boldsymbol{F}$ ausgedrückte Eigenschaft erfüllen.
  - Das Relationschema der neuen Relation ist $Q(A_1, \ldots, A_k)$. Dabei ist Q ein neuer Relationenname.
  - Die Formel $\boldsymbol{F}$ besteht aus
    - **Operanden:** Attributsnamen $A_1, \ldots, A_k$ oder Konstanten
    - **Vergleichsoperatoren:** $=, \neq, <, \leq, >, \geq$
    - **Boolesche Operatoren:** $\wedge, \vee, \neg$ (and, or, not)

- Beispiel (Universität):
  - Zur Bestimmung von $\sigma_{\boldsymbol{F}}(R)$ wird die Formel $\boldsymbol{F}$ für jedes Tupel t betrachtet. Jedes Attribut A in $\boldsymbol{F}$ wird durch den Wert $t.A$ ersetzt.
    - eine naive Auswertungsstrategie überprüft jedes Tupel; die Verwendung von Indexstrukturen ermöglicht hier in manchen Fällen Einsparungen.

$\sigma_{\text{Semester}>11}(\text{Student})$

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12

Sechs Grundoperationen – Projektion

- *Projektion* $\pi_{A_{i_1},\ldots,A_{i_m}}(R)$
  - Die Projektion erlaubt es Spalten (Attribute) aus einer Relation mit Relationenschema $R(A_1, \ldots, A_k)$ auszuwählen.
  - Sei k der Grad von R und $A_{i_1}, \ldots, A_{i_m}$ eine Auswahl von m paarweise verschiedenen Attributen (oder entsprechenden Indizes) aus $A_1, \ldots, A_k$ .Dann gilt:

$$\pi_{A_{i_1},\ldots,A_{i_m}}(R) = \{(a_{i_1}, \ldots, a_{i_m}) \mid (a_1, \ldots, a_k) \in R\}$$

  - Das Relationschema der neuen Relation ist $Q(A_{i_1}, \ldots, A_{i_m})$. Dabei ist Q ein neuer Relationenname.
  - Beispiel:

π_{Rang} (Professor)
Rang
W3
W2

  - Die Anzahl der Tupel kann sich durch implizite Elimination von Duplikaten verringern, solange in der Projektion keine (vollständigen) Schlüssel enthalten sind.

Sechs Grundoperationen – Umbenennung

- *Umbenennung $\rho_S(R)$ oder $\rho_{A' \leftarrow A}(R)$*
  - Die Umbenennung kann sowohl Relationen (R nach S) als auch Attribute einer Relation (A nach A' in der Relation R) umbenennen
  - Sei $R(A_1, \ldots, A_k)$ ein Relationschema. Nach Anwendung von $\rho_S(R)$ ist das neue Relationenschema $S(A_1, \ldots, A_k)$. Dabei ist S ein neuer Relationenname.
  - Sei $R(A_1, \ldots, A, \ldots A_k)$ ein Relationschema. Nach Anwendung von $\rho_{A' \leftarrow A}(R)$ ist das neue Relationenschema $Q(A_1, \ldots, A', \ldots A_k)$ für einen neuen Relationennamen Q.

Beispiel Umbenennung

– Beispiel: Wir wollen die Voraussetzungen 2. Stufe (also die Voraussetzungen der Voraussetzungen) der Vorlesung 5216 bestimmen

$$\pi_V(\sigma_{N=\text{Vorgänger} \wedge \text{Nachfolger}=5216}(\rho_{V \leftarrow Vorgänger}(\rho_{N \leftarrow Nachfolger}(voraussetzen)) \times voraussetzen))$$

- Zwischenergebnis:

Q		voraussetzen	
V	N	Vorgänger	Nachfolger
5001	5041	5001	5041
…	…	…	…
5001	**5041**	**5041**	**5216**
…	…	…	…
5052	5259	5052	5259

- Ergebnis: 5001 ist Vorgänger von 5041, diese ist wiederum Vorgänger von 5216

Zur Übung - wir wiederholen:

- Vereinigung
- Differenz
- Kartesisches Produkt
- Selektion
- Projektion
- Umbenennung

Beispiele - Grundoperationen

- Zwei Relationen (R, S) und ihre Verknüpfung mit relationalen Operationen

R	A	B	C
	a	b	c
	d	a	f
	c	b	d

S	A	B	C
	b	g	a
	d	a	f

$R \cup S$	A	B	C
	a	b	c
	d	a	f
	c	b	d
	b	g	a

$R - S$	A	B	C
	a	b	c
	c	b	d

$R \times S$	R.A	R.B	R.C	S.A	S.B	S.C
	a	b	c	b	g	a
	a	b	c	d	a	f
	d	a	f	b	g	a
	d	a	f	d	a	f
	c	b	d	b	g	a
	c	b	d	d	a	f

$\pi_{A, C}(R)$	A	C
	a	c
	d	f
	c	d

$\sigma_{B = b}(R)$	A	B	C
	a	b	c
	c	b	d

- Mit den vorgestellten Grundoperationen lassen sich nun alle Operationen der relationalen Algebra (induktiv) definieren.

Die Ausdrücke der relationalen Algebra

Induktive Definition der Audrücke der relationalen Algebra:

- ein Basisausdruck ist:
  - eine konstante Relation c (für ein beliebiges Attribute A und einen Wert c $\in dom(A)$ ist die Relation $\{(c)\}$ mit dem Relationenschema $Q(A)$ mit einem neuen Relationennamen Q ein Ausdruck).
  - eine Relation R der Datenbank

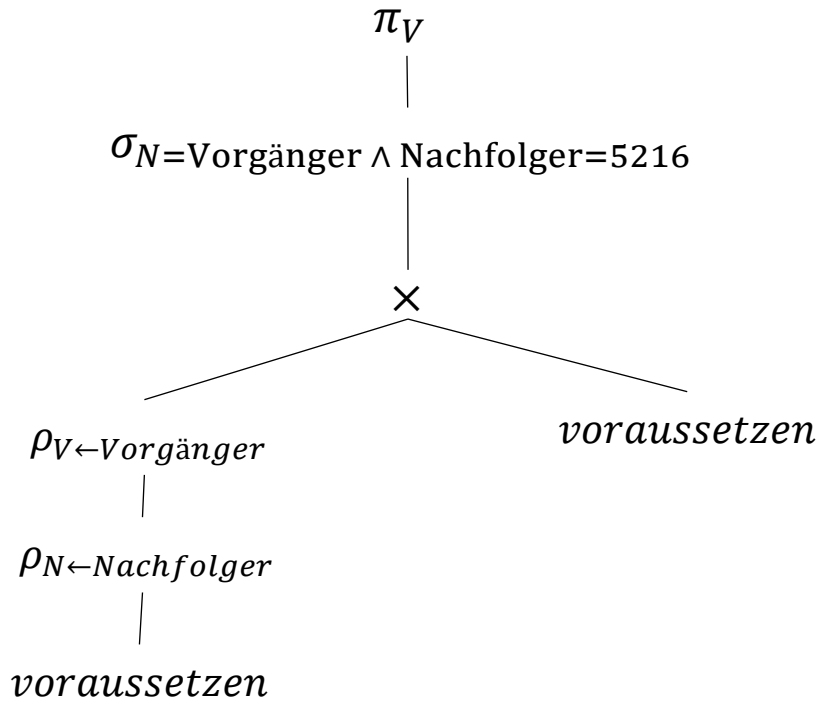
- seien E und F Relationenalgebra-Ausdrücke, A' und A Attribute, T, S und V wie bei den Algebra-Operatoren definiert. Dann sind folgendes auch gültige Ausdrücke, wobei die Anforderungen der Algebraoperatoren an die Relationenschemata gelten.
  - $E \cup F,\ E - F, E \times F, \sigma_T(E), \pi_S(E),\ \rho_V(E)$ und $\rho_{A' \leftarrow A}(E)$
  - Das sind alle (Minimalität)
- Mit der vorgestellten Algebra lassen sich beliebig komplexe Anfragen (Queries) formulieren, zum Beispiel die im Weiteren vorgestellten zusätzlichen Operationen

Operatorbäume

- Jeder Ausdruck einer Algebra lässt sich auch als *Operatorbaum* darstellen.
- Beispiel:

$$\pi_V(\ \sigma_{N=\text{Vorgänger} \wedge \text{Nachfolger}=5216}(\ \rho_{V \leftarrow Vorgänger}(\ \rho_{N \leftarrow Nachfolger}(voraussetzen)) \times voraussetzen))$$

  - Die Blätter enthalten *Relationen*.
  - Die inneren Knoten repräsentieren die *Operationen*.



Weitere Operationen – Durchschnitt

- Neben den sechs Grundoperationen existiert eine Reihe anderer nützlicher relationaler Operationen.
- *Durchschnitt* $R \cap S$
  - Die beiden Relationen R und S müssen das gleiche Schema besitzen. Dann ist $R \cap S$ definiert als die Schnittmenge der beiden Relationen, d.h.

$$R \cap S = \{t \mid t \in R \wedge t \in S\} \qquad = R - (R - S)$$

  - Beispiel: Finde die PersNr der W3-Professoren die mindestens eine Vorlesung halten.

$$\pi_{\text{PersNr}}\left(\rho_{\text{PersNr} \leftarrow \text{gelesenVon}}(\text{Vorlesungen})\right) \cap \pi_{\text{PersNr}}(\sigma_{\text{Rang=W3}}(\text{Professoren}))$$

    - Ergebnis

PersNr

Vorlesung

VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

Professor

PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
2126	Russel	W3	232
2127	Kopernikus	W2	310
2133	Popper	W2	52
2134	Augustinus	W2	309
2136	Curie	W3	36
2137	Kant	W3	7

Weitere Operationen – Durchschnitt

- Neben den sechs Grundoperationen existiert eine Reihe anderer nützlicher relationaler Operationen.
- *Durchschnitt* $R \cap S$
  - Die beiden Relationen R und S müssen das gleiche Schema besitzen. Dann ist $R \cap S$ definiert als die Schnittmenge der beiden Relationen, d.h.

$$R \cap S = \{t \mid t \in R \wedge t \in S\} \qquad = R - (R - S)$$

  - Beispiel: Finde die PersNr der W3-Professoren die mindestens eine Vorlesung halten.

$$\pi_{\text{PersNr}}\left(\rho_{\text{PersNr} \leftarrow \text{gelesenVon}}(\text{Vorlesungen})\right) \cap \pi_{\text{PersNr}}(\sigma_{\text{Rang=W3}}(\text{Professoren}))$$

    - Ergebnis

PersNr
2125
2126
2137

Vorlesung

VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

Professor

PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
2126	Russel	W3	232
2127	Kopernikus	W2	310
2133	Popper	W2	52
2134	Augustinus	W2	309
2136	Curie	W3	36
2137	Kant	W3	7

Weitere Operationen – Natürlicher Verbund (natural join)

- Natürlicher Verbund $R \bowtie S$
  - Idee: Selektiere aus dem Kreuzprodukt $R \times S$ nur zueinander „passende“ Tupel. Es sind nicht immer alle Tupel eines Kreuzprodukts relevant.

  - Sei $|\mathrm{Sch}(R) \cap \mathrm{Sch}(S)| = k$,d.h. R und S haben k gemeinsame Attribute. Wir schreiben die $m + k$ Attribute von R als $A_1, \ldots, A_m, B_1, \ldots, B_k$ und die $n + k$ Attribute von S als $B_1, \ldots, B_k, C_1, \ldots, C_n$. Weiter sind $D_1, \ldots, D_k$ neue Attributenamen. Dann ist

$$R \bowtie S = \pi_{A_1,\ldots,A_m,B_1,\ldots,B_k,C_1,\ldots,C_n}\left(\sigma_{D_1=B_1 \wedge \ldots \wedge D_k=B_k}\left(\rho_{B_1 \leftarrow D_1}(\ldots(\rho_{B_k \leftarrow D_k}(R))) \times S\right)\right)$$

  - Beispiel: Zugriff auf die Informationen einer m:n-Beziehung, wie zum Beispiel

$$\mathrm{Student} \bowtie \mathrm{hört} \bowtie \mathrm{Vorlesung}$$

Weitere Operationen – Natürlicher Verbund (natural join) (2)

– Beispiel: $\text{Student} \bowtie \text{hört} \bowtie \text{Vorlesung}$

Student		
MatrNr	Name	Sem.
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
27550	Carnap	8
…	…	…

hört	
MatrNr	VorlNr
25403	5022
26120	5001
27550	5001
27550	4052
…	…

Vorlesung			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5049	Mäeutik	2	2125
4052	Logik	4	2125
…	…	…	

$(\text{Student} \bowtie \text{hört}) \bowtie \text{Vorlesung}$						
MatrNr	Name	Semester	VorlNr	Titel	SWS	gelesenVon
25403	Jonas	12	5022	Glaube und Wissen	2	2134
26120	Fichte	10	5001	Grundzüge	4	2137
27550	Carnap	8	5001	Grundzüge	4	2137
…	…	…	…	…	…	…

Weitere Operationen – Natürlicher Verbund (natural join) (3)

- Der natürliche Join ist sowohl **assoziativ**, d.h.

$$(\text{Student} \bowtie \text{hört}) \bowtie \text{Vorlesung} = \text{Student} \bowtie (\text{hört} \bowtie \text{Vorlesung}),$$

  als auch **kommutativ** bis auf Vertauschung der Attributreihenfolge, zum Beispiel ist

$$\text{Student} \bowtie \text{hören} = \text{hören} \bowtie \text{Student}.$$

- Da der natürliche Join nur auf Attributen mit gleichen Attributnamen arbeitet, kann es notwendig sein Attribute umzubenennen.
  - Wollen wir die Relationen *Vorlesung* und *Professor* miteinander verbinden, müssen wir eines der Attribute *Vorlesung.gelesenVon* oder *Professor.PersNr* umbenennen.

$$\text{Vorlesung} \bowtie \rho_{\text{gelesenVon} \leftarrow \text{PersNr}}(\text{Professor})$$

- Wird der natürliche Join auf zwei Relationen ohne gleiche Attributnamen angewandt, dann entspricht des Ergebnis dem Ergebnis des kartesischen Produktes der beiden Relationen.

Weitere Operationen – Theta-Join

- Der Theta-Join $R \bowtie_\theta S$:
  - Idee: Selektiere aus dem Kreuzprodukt $R \times S$ nur zueinander „passende“ Tupel. „Passend“ wird hierbei durch die Formel θ festgelegt.
  - Seien $\mathrm{Sch}(R) = A_1, \ldots, A_n$ und $\mathrm{Sch}(S) = B_1, \ldots, B_m$ und θ eine Formel über $A_1, \ldots, A_n, B_1, \ldots, B_m$ .Dann ist

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

  - Doppelte Attribute werden nicht aussortiert – d.h., müssen vorher umbenannt werden! Wir können dies implizit verlangen indem wir den Relationennamen mit angegeben.
  - Beispiel:

$$\text{Student} \bowtie_{\text{S.MatrNr=h.MatrNr} \wedge \text{ S.Semester>9}} \text{hört}$$

steht für

$$\text{Student} \bowtie_{\text{MatrNr=}ma \wedge \text{ Semester>9}} \rho_{\text{ma} \leftarrow MatrNr}(hört)$$

Student. MatrNr	Name	Semester	hört. MatrNr	VorlNr
25403	Jonas	12	25403	5022
26120	Fichte	10	26120	5001
…	…	…		…

Weitere Operationen – Äußere Join-Operatoren

- Im Gegensatz zum natürlichen Join bleiben bei äußeren Join-Operatoren auch Tupel erhalten, die keinen "Joinpartner" gefunden haben.
  - left outer join $⟕$: Die Tupel der linken Relation bleiben erhalten.
  - right outer join $⟖$: Die Tupel der rechten Relation bleiben erhalten.
  - (full) outer join $⟗$: Die Tupel beider Relationen bleiben erhalten.

- Darüber hinaus gibt es die Semi-Join-Operatoren $\ltimes$ und $\rtimes$. Der linke Semi-Join von R mit L – in Zeichen $L \ltimes R$ – ist definiert als

$$L \ltimes R = \pi_{\mathcal{L}}(L \bowtie R),$$

  wobei $\mathcal{L}$ die Menge der Attribute von L ist.

Der rechte Semi-Join $(L \rtimes R)$ analog dazu (mit $\mathcal{L}$ dann Menge der Attribute von R)

Join-Operatoren – Beispiele

natural join

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

left outer join

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-

full outer join

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-
-	-	c_3	d_2	e_2

left semi-join L with R

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\ltimes$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat		
A	B	C
a_1	b_1	c_1

3. Das Relationale Datenmodell

1. Das Datenmodell
2. Transformation von ER-Diagrammen in das Relationale Modell
3. Relationale Algebra
4. **Relationaler Kalkül**

Relationaler Kalkül

- **Relationale Algebra**
  - Algebra: Menge mit Operationen (hier: Relationen und relationale Operatoren).
  - Anfragen werden als Operationen auf den Relationen einer Datenbank ausgedrückt.
  - Die Antwortmenge wird durch die tatsächliche Anwendung der Operationen berechnet.
  - *konstruktives* (**prozedurales**) Vorgehen: Es wird angegeben, **WIE** eine Anfrage zu bearbeiten ist

- **Relationenkalkül(e)**
  - Kalkül: Logischer Formalismus zum Ableiten von Ergebnissen.
  - Anfragen werden durch Formeln der Prädikatenlogik erster Stufe ausgedrückt.
  - Die Antwortmenge enthält genau diejenigen Tupel, welche die Formel erfüllen.
  - *deskriptives* (**deklaratives**) Vorgehen: Vorgabe des **WAS**, aber nicht des WIE einer Anfrage
  - Für den Relationenkalkül unterscheiden wir zwei Arten: **Tupel- und Bereichskalkül**

- **Äquivalenz**: Ausdrücke der Relationenalgebra, sichere Ausdrücke des Tupelkalküls und sichere Ausdrücke des Bereichskalküls äquivalent in ihrer Mächtigkeit.
  - Ein Ausdruck des Tupelkalküls bzw. des Bereichskalküls heißt sicher, wenn sich aus der syntaktischen Struktur erkennen lässt, dass bei der Auswertung einer Anfrage nur endlich viele Tupelkandidaten überprüft werden müssen.
  - Das bedeutet, zu jedem Ausdruck der in der Relationenalgebra formuliert ist, gibt es einen Ausdruck im Relationenkalkül der die gleiche Rückgabe liefert, und andersherum.

Relationaler Kalkül

- Relationenkalkül: Prädikatenlogik erster Stufe.
  - *Syntax*: wie können gültige Ausdrücke des Kalküls gebildet werden. Dabei treten Konstanten, Variablen, Operatoren und Quantoren auf (symbolische Ebene).
  - *Semantik*: beschreibt die Bedeutung von syntaktisch korrekt gebildeten Ausdrücken. Dabei geht es um Relationen und die Erfüllung von Bedingungen (Bedeutungsebene).

- Zwei ähnliche Arten des Relationenkalküls:
  - *Tupelkalkül*: Variablen repräsentieren die Tupel einer Relation
  - *Domänenkalkül*: Variablen repräsentieren mögliche Werte eines Attributs

- Zunächst: Beispiele zu Relationenkalkül-Anfragen und Quantoren.
- Später: Formale Definition von Syntax und Semantik

Der Tupelkalkül – Beispiele

- Form der Ausdrücke im Tupelkalkül:

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

  Hier ist t eine *Tupelvariable* und F eine *Formel*.

  – die neue Relation besteht aus den Tupeln die die Formel F erfüllen

  – die Variable t muss in F eine *freie Variable* sein (= nicht quantifiziert, dazu gleich mehr)

  – Beispiel: Finde alle Professoren mit Rang ‘W3’.

$$\{ p \mid p \in \text{Professor} \wedge p.\text{Rang} = 'W3' \}$$

    - Auswertung:
      - p wird an jedes Tupel der Relation Professor gebunden
      - dann wird für jedes Tupel die $\text{Rang} = 'W3'$ Bedingung überprüft

Der Tupelkalkül – Beispiele

- Form einer Aunfrage im Tupelkalkül:

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

  Hier ist t eine *Tupelvariable* und F eine *Formel*.

  – die neue Relation besteht aus den Tupeln die die Formel F erfüllen

  – die Variable t muss in F eine *freie Variable* sein (= nicht quantifiziert, dazu gleich mehr)

  – Beispiel: Finde alle Professoren mit Rang ‘W3’.

$$\{ p \mid p \in \text{Professor} \wedge p.\text{Rang} = 'W3' \}$$

    - Auswertung:
      - p wird an jedes Tupel der Relation Professor gebunden
      - dann wird für jedes Tupel die $\text{Rang} = 'W3'$ Bedingung überprüft

Der Tupelkalkül – Quantifizierung

- Existenz- und Allquantoren $\exists, \forall$
  - $\exists t \in R\ \big(P(t)\big)$: es existiert ein t in R, sodass $P(t)$ gilt.
  - $\forall t \in R\ \big(P(t)\big)$: für alle t aus R gilt $P(t)$.
  - hierbei muss t in P eine freie Variable sein

  - Beispiel: Finde die Studenten, die mindestens eine Vorlesung bei der Professorin Curie hören.

    s ist in dem folgenden Prädikat die einzige freie Tupelvariable!

$$\{ s \mid s \in \text{Student} \ \wedge\ \exists h \in \text{hört}$$
$$(s.\text{MatrNr} = h.\text{MatrNr} \ \wedge\ \exists v \in \text{Vorlesung}$$
$$(h.\text{VorlNr} = v.\text{VorlNr} \ \wedge\ \exists p \in \text{Professor}$$
$$(p.\text{PersNr} = v.\text{gelesenVon} \ \wedge p.\text{Name} = \text{'Curie'}\ ))) \}$$

Der Tupelkalkül – Quantifizierung

- Existenz- und Allquantoren $\exists, \forall, \nexists$

  – Beispiel: Finde die Studenten, die alle von Professor Sokrates (PersNr 2125) angebotenen Vorlesungen besuchen.

$$\{ s \mid s \in \text{Student} \wedge \forall v \in \text{Vorlesung}$$
$$(v.\text{gelesenVon} = 2125 \rightarrow \exists h \in \text{hören}$$
$$(h.\text{VorlNr} = v.\text{VorlNr} \wedge h.\text{MatrNr} = s.\text{MatrNr}))\}$$

„Finde die Studierenden, für die alle Vorlesungen v, die von 2125 gelesen werden, impliziert, dass eine Beziehung „hören“ existiert, für die die Vorlesungsnummer der Vorlesung v und die Matrikelnummer der Studierenden übereinstimmt. „

Auch hier ist s die einzige freie Variable des Prädikats.

$$F \rightarrow G := \neg F \vee G$$

Der Tupelkalkül – Konstruktion neuer Relationen

- Schema einer Tupelvariable:
  - Eine Tupelvariable t besitzt ein Schema $(A_1: D_1, \ldots, A_k: D_k)$

$$\{\, t \in (A_1: D_1, \ldots, A_k: D_k) \mid F(t) \,\}$$

  - ist das Schema $\mathrm{Sch}(t)$ aus dem Zusammenhang klar, so wird auf die Angabe verzichtet

$$\{\, p \mid p \in \mathrm{Professor} \wedge p.\mathrm{Rang} = \text{'}W3\text{'} \,\}$$

  - über das Schema können neue Tupel konstruiert werden (vgl. Projektion und Join):
    - PersNr aller Professoren mit $\mathrm{Rang} =$ 'W3'

$$\{\, t \in (\mathrm{PersNr}: \mathrm{String}) \mid \exists p \in \mathrm{Professor}\ (t.\mathrm{PersNr} = \mathrm{p}.\mathrm{PersNr} \wedge p.\mathrm{Rang} = \text{'}W3\text{'}) \,\}$$

    - Kurzschreibweise (Tupelkonstruktor, nächste Seite):

$$\{\, [p.\mathrm{PersNr}] \mid p \in \mathrm{Professor} \wedge p.\mathrm{Rang} = \text{'}W3\text{'} \,\}$$

Der Tupelkalkül – Konstruktion neuer Relationen

- Konstruktion neuer Tupel:
  - Tupelkonstruktor [...]: $\{ [t_1.A_1, \ldots, t_n.A_n] \mid P(t_1, \ldots, t_n) \}$
    - die Attribute $A_1, \ldots, A_n$ müssen in den Schemata der Relationen enthalten sein, an die die $t_1, \ldots, t_n$ gebunden werden
  - Beispiel (Projektion und Join):
    - Ordne jedem Professor (Name) den ihm zugeordneten Assistenten (PersNr) zu

$$\{ [p.\text{Name}, a.\text{PersNr}] \mid p \in \text{Professor} \wedge a \in \text{Assistent} \wedge p.\text{PersNr} = a.\text{Boss} \}$$

    - Kurzschreibweise für:

$$\{ t \in (\text{Name: String}, \text{PersNr: Int}) \mid \exists p \in \text{Professor}\ \exists a \in \text{Assistent}$$
$$(t.\text{Name} = p.\text{Name} \wedge t.\text{PersNr} = a.\text{PersNr} \wedge p.\text{PersNr} = a.\text{Boss} \}$$

Der Tupelkalkül – Formale Definition (Syntax)

- Syntax: **Tupelvariable** t ⇒ Formel $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Ein Ausdruck liefert als Ergebnis die Relation aller Tupel, die die Formel F erfüllen.
  - Konstruktion von gültigen Tupelkalkül-Ausdrücken
  - <u>Tupelvariablen:</u>
    - Eine Tupelvariable t besitzt ein Schema $\mathrm{Sch}(t) = (A_1 : D_1, \ldots, A_k : D_k)$

      $$\{ t \in (A_1 : D_1, \ldots, A_k : D_k) \mid F(t) \}$$

    - Das Schema $\mathrm{Sch}(t)$ ist oft implizit festgelegt durch
      - eine Formel der Form $t \in R$, oder
      - die Struktur des Tupelkonstruktors $[t_1.A_1, \ldots, t_n.A_n]$
    - Eine Tupelvariable kann *frei* oder *gebunden* auftreten (siehe Syntax: Formel)

Der Tupelkalkül – Formale Definition (Syntax)

- Syntax: Tupelvariable t ⇒ **Formel** $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Ein Ausdruck liefert als Ergebnis die Relation aller Tupel, die die Formel F erfüllen.
  - Die Grundbausteine der Formeln sind die Atome
  - Tupelvariablen kommen in Atomen grundsätzlich nur *frei* vor
  - Es gibt drei Arten von Atomen:
    - $t \in R$ ist ein Atom, wobei t eine Tupelvariable und R eine Relation ist
    - $t.A\ \theta\ s.B$ ist ein Atom, wobei t, s Tupelvariablen sind, A und B Attributnamen von t bzw. s und $\theta \in \{=, \neq, <, \leq, >, \geq\}$ ein Vergleichsoperator. $t.A$ und $t.B$ müssen bzg. θ vergleichbar sein
    - $t.A\ \theta\ c$, $c\ \theta\ t.A$ wobei im Unterschied zu oben c eine Konstante ist

Der Tupelkalkül – Formale Definition (Syntax)

- Syntax: Tupelvariable t ⇒ **Formel** $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Ein Ausdruck liefert als Ergebnis die Relation aller Tupel, die die Formel F erfüllen.
  - Der Aufbau von Formeln ist rekursiv definiert:
    - Atome: Jedes Atom ist eine Formel
    - Verknüpfungen: Seien F, G Formeln. Dann sind auch $\neg F$, $F \wedge G$ und $F \vee G$ Formeln
    - Quantoren: Sei F eine Formel in der t als *freie* Variable auftritt. Dann sind auch $\exists t \in R\ (F)$ und $\forall t \in R\ (F)$ Formeln. t ist in dieser Formel dann eine *gebundene* Variable
  - Beispiel: In der folgenden Formel ist s eine freie Variable, h eine gebundene.

$$s \in \text{Student} \wedge \exists h \in \text{hören}\ (h.\text{MatrNr} = s.\text{MatrNr})$$

Der Tupelkalkül – Formale Definition (Syntax)

- Syntax: Tupelvariable t ⇒ Formel $F(t)$ ⇒ **Ausdruck** $\{ t \mid F(t) \}$
  - Idee: Ein Ausdruck liefert als Ergebnis die Relation aller Tupel, die die Formel F erfüllen.
  - Ausdruck: Ein Ausdruck des Tupelkalküls hat die Form
    - $\{ t \mid F(t) \}$, t ist die einzige freie Tupelvariable in der Formel F

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: Tupelvariable t ⇒ Formel $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Analoge drei Schritte:
    - Tupelvariablen ⇒ konkrete Tupel
    - Formeln (Atome, Operatoren & Quantoren) ⇒ true, false
    - Ausdrücke ⇒ Relationen

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: **Tupelvariable** t ⇒ Formel $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Belegung von Tupelvariablen: Seien
    - t eine Tupelvariable mit Schema $\mathrm{Sch}(t) = (A_1 : D_1, \ldots, A_k : D_k)$,
    - $F(t)$ eine Formel, die (i.A. nicht nur) t als freie Tupelvariable enthält
    - $r \in D_1 \times \cdots \times D_k$ ein beliebiges Tupel (muss i.A. nicht zu geg. Relation gehören)

    Bei der Belegung von t mit r wird jedes freie Vorkommen von t in $F(t)$ durch r ersetzt:

    $$F(r|t)$$

  - Beispiel: Sei $F(t) = t \in \mathrm{Professor} \wedge t.\mathrm{Rang} = 'W3'$ mit $\mathrm{Sch}(t) = \mathrm{Sch}(\mathrm{Professor})$.

    Für $r_1 = (2125, 'Sokrates', 'W3', 226)$ ist

    $$F(r_1|t) = r_1 \in \mathrm{Professor} \wedge 'W3' = 'W3'$$

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: Tupelvariable t ⇒ **Formel** $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Interpretation $I(F)$ einer Formel F (analog zu syntaktischem Aufbau):
    - Die Formel F darf keine freien Variablen mehr enthalten.
    - Atome haben dann nur noch zwei Formen:
      - $I(r \in R) = \mathbf{true}$,falls r in R enthalten ist; sonst $\mathbf{false}$
      - $I(c_1 \theta c_2) = \mathbf{true}$,falls c_1 in Beziehung θ zu c_2 steht (Bsp.: $3 < 7$)
  - Beispiel:

$$I\big((2125, \text{'Sokrates'}, \text{'}W3\text{'}, 226) \in \text{Professor}\big) = \mathbf{true}$$

$$I(\text{'}W3\text{'} = \text{'}W3\text{'}) = \mathbf{true}$$

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: Tupelvariable t ⇒ **Formel** $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Interpretation $I(F)$ einer Formel F (analog zu syntaktischem Aufbau):
    - Logische Operatoren:
      - $I(\neg F) = \mathbf{true}$, falls $I(F) = \mathbf{false}$ ist und umgekehrt
      - $I(F_1 \wedge F_2) = \mathbf{true}$ genau dann, wenn $I(F_1) = I(F_2) = \mathbf{true}$
      - $I(F_1 \vee F_2) = \mathbf{true}$, falls mindestens eines von $I(F_1), I(F_2) = \mathbf{true}$ ist
  - Beispiel:

$$I\big((2125, \text{'Sokrates'}, \text{'W3'}, 226) \in \text{Professor} \wedge \text{'W3'} = \text{'W3'}\big) = \mathbf{true}$$

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: Tupelvariable t ⇒ **Formel** $F(t)$ ⇒ Ausdruck $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Interpretation $I(F)$ einer Formel F (analog zu syntaktischem Aufbau):
    - Quantoren:
      - Zur Interpretation von $\exists t \in R\ (F(t))$ und $\forall t \in R\ (F(t))$ darf nur t in F frei sein
      - $I\left(\exists t \in R\ (F(t))\right) = \mathbf{true}$ gdw ein $r \in R$ existiert, sodass $I(F(r|t)) = \mathbf{true}$ ist
      - $I\left(\forall t \in R\ (F(t))\right) = \mathbf{true}$ gdw für alle $r \in R$ gilt: $I(F(r|t)) = \mathbf{true}$
  - Beispiel:

$$I(\forall t \in \text{Vorlesung}\ (t.\text{gelesenVon} = '2125')) = \mathbf{false}$$

Der Tupelkalkül – Formale Definition (Semantik)

- Semantik: Tupelvariable t ⇒ Formel $F(t)$ ⇒ **Ausdruck** $\{ t \mid F(t) \}$
  - Idee: Interpretation der Tupelkalkül-Ausdrücke
  - Interpretation eines Ausdrucks:

    Sei $E = \{t \mid F(t)\}$ oder $E = \{[t_1.A_1, \ldots, t_n.A_n] \mid F(t_1, \ldots, t_n)\}$ ein Ausdruck und $D_1 \times \cdots \times D_k$ das Schema von t bzw. $[t_1.A_1, \ldots, t_n.A_n]$.
    - t bzw. $t_1, \ldots, t_n$ dürfen die einzigen freien Variabeln in F sein.
    - Der Wert von E ist die Menge aller Tupel $r \in D_1 \times \cdots \times D_k$ für die gilt

$$I\big(F(r|t)\big) = \mathbf{true}$$

Der Domänenkalkül

- Im Unterschied zum Tupelkalkül: *Bereichsvariablen*
  - Ein Ausdruck mit k Bereichsvariablen $x_1 : D_1, \ldots, x_k : D_k$ hat die Form

$$\{x_1, \ldots, x_k \mid F(x_1, \ldots, x_k)\}$$

- Definitionen:
  - Atome:
    - $R(x_1, \ldots, x_k)$ Bedeutung: ist wahr, falls nach Belegung der x_i mit $u_i \in D_i$ das Tupel $(u_1, \ldots, u_k)$ in R enthalten ist
    - $x\ \theta\ y$ Bedeutung: ist wahr, falls nach Belegung x in der Beziehung θ zu y steht
  - Formeln: Analog zum Tupelkalkül, Quantoren $\exists, \forall$ beziehen sich auf Domänen
  - Ausdruck: siehe oben

Der Domänenkalkül

- Beispiele:
  - Finde MatNr und Name der Studenten, die mindestens eine Prüfung bei Professor Curie abgelegt haben.

$$\{ m, n \mid \exists s \, (Student(m, n, s) \; \wedge \exists v, p, g \, (prüfen(m, v, p, g)$$
$$\wedge \exists a, r, b \, (Professor(p, a, r, b) \wedge a = \text{'Curie'}))) \; \}$$

  - Operationen der Relationalen Algebra:

$$R \cup S = \{x_1, \ldots, x_n \mid R(x_1, \ldots, x_n) \vee S(x_1, \ldots, x_n)\}$$
$$R - S = \{x_1, \ldots, x_n \mid R(x_1, \ldots, x_n) \wedge \neg S(x_1, \ldots, x_n)\}$$
$$P \bowtie Q = \{x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+k}, \ldots, x_{n+k+m} \mid$$
$$P(x_1, \ldots, x_{n+k}) \wedge Q(x_{n+1}, \ldots, x_{n+k+m}) \}$$

Beispiele Tupelkalkül vs. Domänenkalkül

- MatNr und Name der Studenten, die mindestens eine Prüfung bei Professor Curie abgelegt haben.

- Tupelkalkül

$$\{ [s.\text{MatrNr}, s.\text{Name}] \mid s \in \text{Student} \\ \wedge \exists h \in \text{hört}\ (s.\text{MatrNr} = h.\text{MatrNr} \\ \wedge \exists v \in \text{Vorlesung}\ (h.\text{VorlNr} = v.\text{VorlNr} \\ \wedge \exists p \in \text{Professor}\ \ (p.\text{PersNr} = v.\text{gelesenVon} \\ \wedge p.\text{Name} = \text{'Curie'}\ ))) \}$$

- Domänenkalkül

$$\{ m, n \mid \exists s\ (Student(m, n, s)\ \wedge \exists v, p, g\ (prüfen(m, v, p, g) \\ \wedge \exists r, b\ (Professor(p, \text{'Curie'}, r, b))\ )\ ) \}$$

Kurzschreibweise

Beispiele Tupelkalkül vs. Domänenkalkül

- PersNr aller Professoren mit $\text{Rang} = \text{'W3'}$

- Tupelkalkül

$$\{ [p.\text{PersNr}] \mid p \in \text{Professor} \wedge p.\text{Rang} = 'W3' \}$$

- Domänenkalkül

$$\{p | \exists n, b\ (Professor(p, n, 'W3', b))\}$$

Beispiele Tupelkalkül vs. Domänenkalkül

- Ordne jedem Professor (Name) den ihm zugeordneten Assistenten (PersNr) zu

- Tupelkalkül

$$\{ [p.\text{Name}, a.\text{PersNr}] \mid p \in \text{Professor} \wedge a \in \text{Assistent} \wedge p.\text{PersNr} = a.\text{Boss} \}$$

- Domänenkalkül

$$\{ \text{np}, \text{pa} \mid \exists \text{p}, \text{r}, \text{b}, \text{na}, \text{f}(\text{Professor}(\text{p}, \text{np}, \text{r}, \text{b}) \wedge \text{Assistent}(\text{pa}, \text{na}, \text{f}, p)) \}$$

Folie 1

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - **Einführung**
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Auschnitt aus unserer Datenbank

Student		
MatrNr	**Name**	**Semester**
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

8 Tupel

hört	
MatrNr	**VorlNr**
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

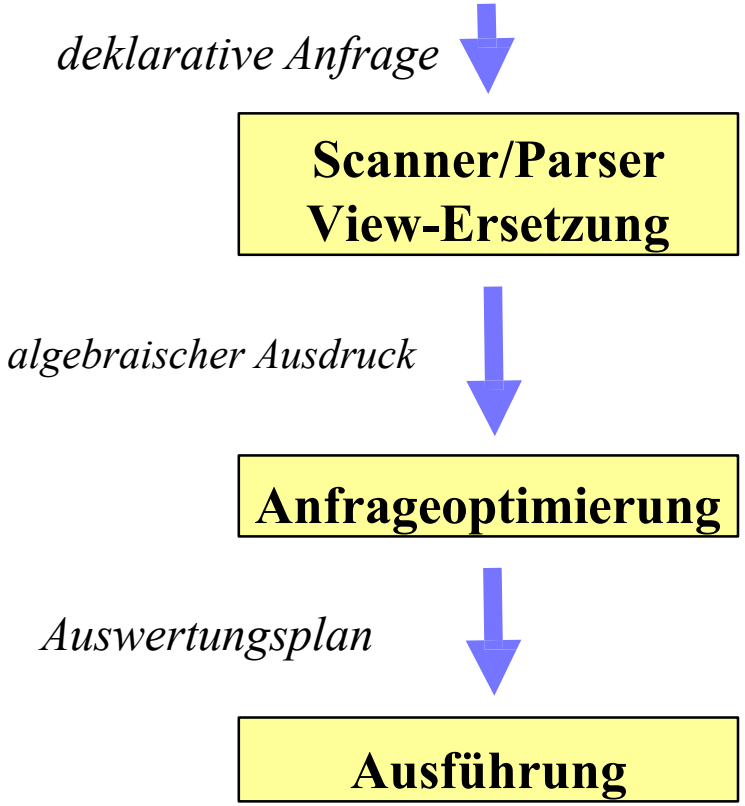
12 Tupel

Vorlesung			
VorlNr	**Titel**	**SWS**	**gelesenVon**
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

10 Tupel

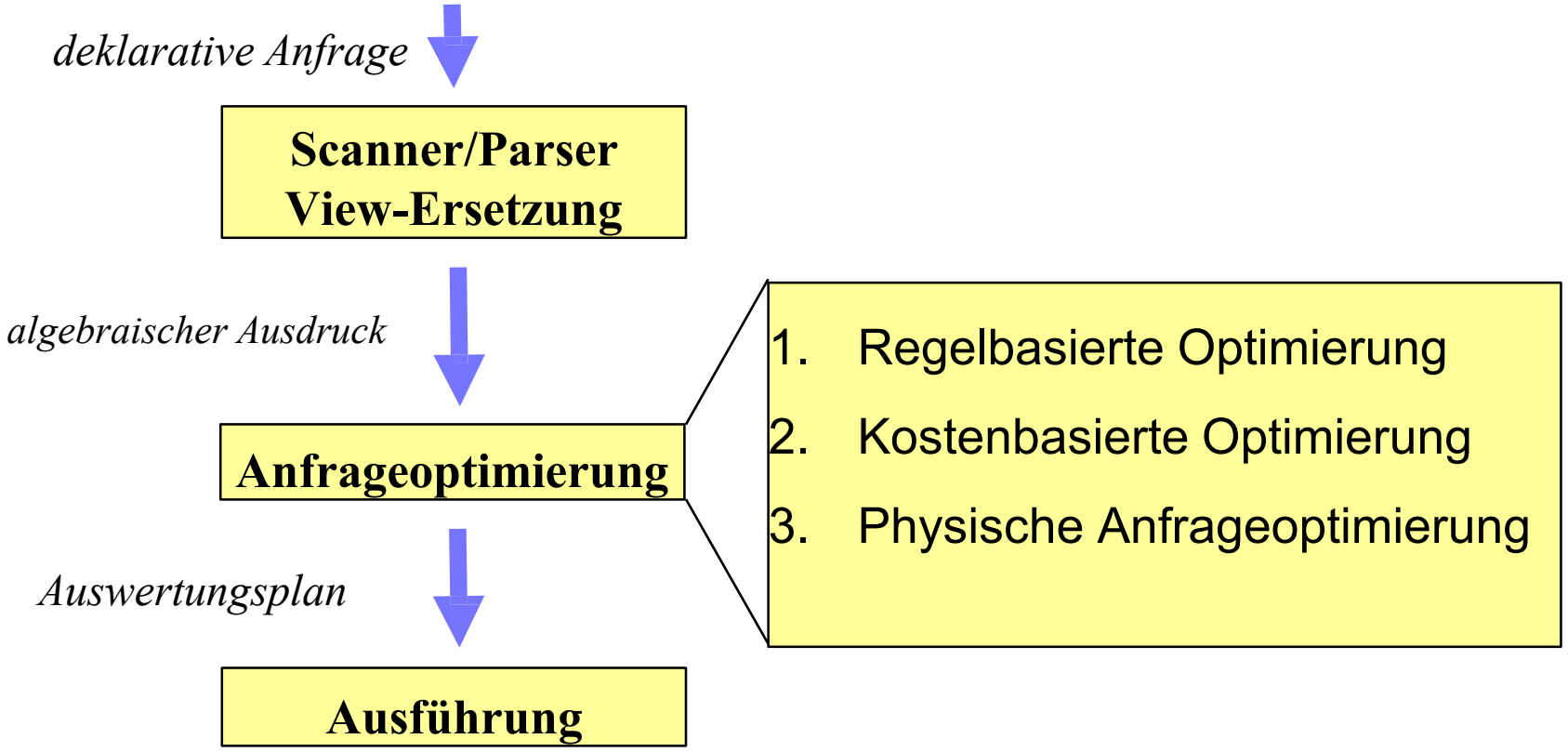
Aufgabe der Anfragebearbeitung

– Übersetzung der *deklarativen* Anfrage in einen *effizienten, prozeduralen* Auswertungsplan



Aufgabe der Anfragebearbeitung

– Übersetzung der *deklarativen* Anfrage in einen *effizienten, prozeduralen* Auswertungsplan



Drei Arten von Optimierungen

- Regelbasierte Optimierung
  - Ausnutzung von Gleichungen der relationalen Algebra
  - Heuristiken auf Basis der Schemainformation
  - Keine Betrachtung der betroffenen Daten

- Kostenbasierte Optimierung
  - Berücksichtigung der Daten
  - Verwendung von Statistiken (z.B. Histogramme)
  - Schätzung der Kosten von Auswertungsplänen
  - Zum Beispiel für die Join-Reihenfolge

- Physische Anfrageoptimierung
  - Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation
  - Verwendung Index bei Selektionsoperation (und welche Art)
  - Pipelined vs. Materialisierung

Beobachtungen

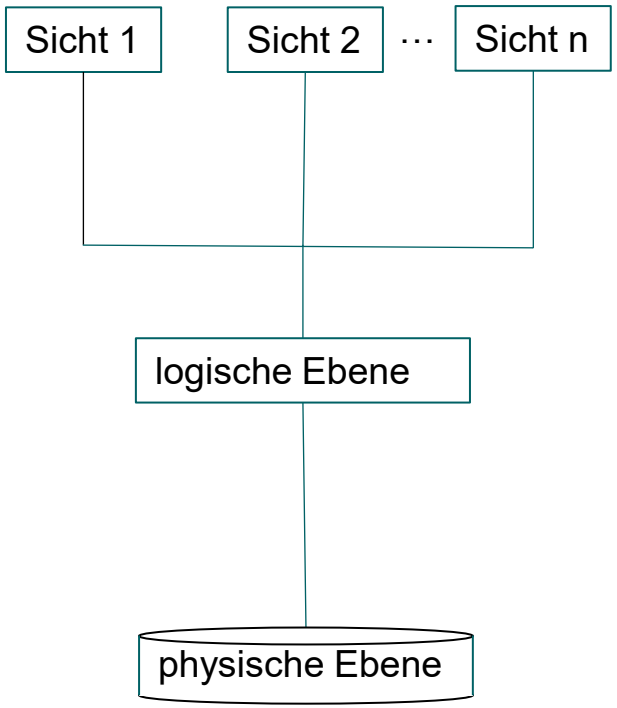
- Es kann viele verschiedene, *gleichwertige* Auswertungspläne für dieselbe Anfrage geben.
- Die Performanz gleichwertiger Auswertungspläne variiert häufig zwischen wenigen Sekunden (schnellster Plan) und vielen Stunden (Standardplan).
- Die Aufgabe der Anfrageoptimierung:

  ⇒ Den günstigsten Auswertungsplan zu ermitteln

  (bzw. zumindest einen sehr günstigen Plan zu ermitteln).
- Wegen des großen Unterschiedes zwischen günstigstem und ungünstigstem Plan ist die Optimierung bei der relationalen Anfragebearbeitung wesentlich wichtiger als z.B. bei der Übersetzung von (imperativen) Programmiersprachen.

Weitere Gründe für die Optimierung

- „physischen Datenunabhängigkeit“: Benutzer soll von der physischen Organisatisation der Daten abgeschirmt sein
- Der Benutzer braucht eine Anfrage nicht selbst zu optimieren.
- Indexe können zur Leistungssteigerung vom DB-Administrator angelegt werden.
- Die Änderungen sind für Anwendungsprogramme und adhoc-Anfragen transparent.



3-Ebenen Modell nach ANSI/SPARC (1975)

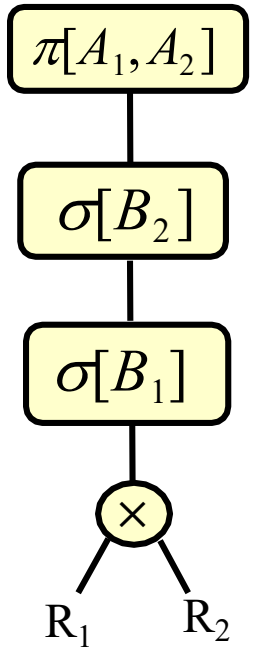
Kanonischer Auswertungsplan einer SQL Anfrage

SELECT A_1, A_2

FROM R_1, R_2

WHERE B_1 *AND* B_2

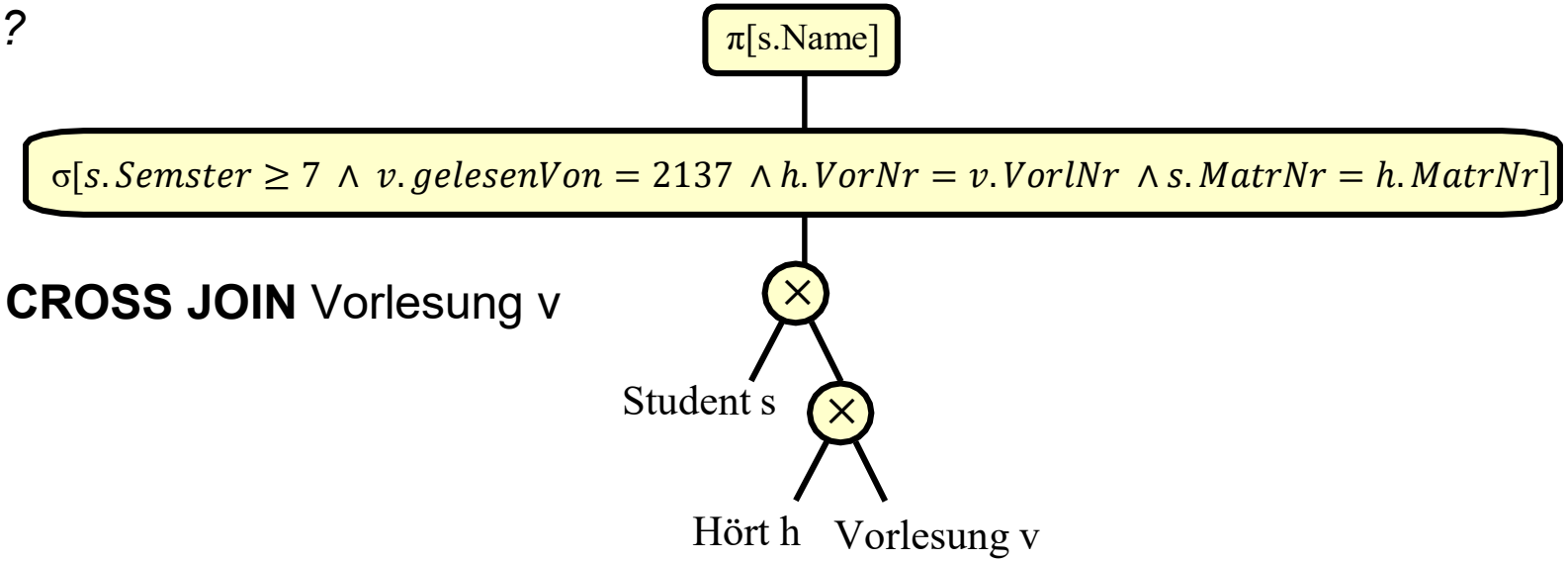
$$\pi_{A_1,A_2}(\sigma_{B_2}(\sigma_{B_1}(R_1 \times R_2)))$$



1. Bilde das kartesische Produkt der Relationen R_1, R_2
2. Führe Selektionen mit den Bedingungen B_1, B_2 durch.
3. Projiziere die Ergebnis-Tupel auf die erforderlichen Attribute A_1, A_2

Beispiel: SQL-Anfrage

- Anfrage: *Welche Studierenden (Name) sind mindestens im siebten Semester und haben eine Vorlesung besucht, die von Professor 2137 gelesen wurde?*



- SQL:

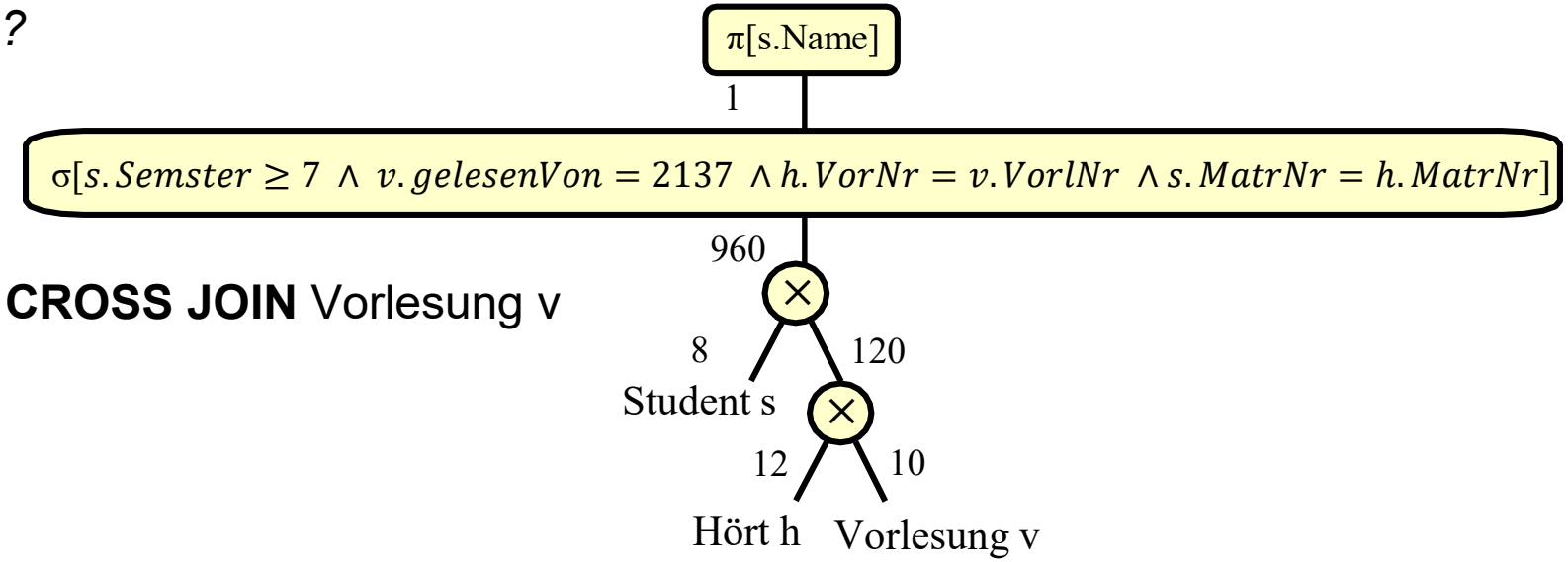
```
SELECT DISTINCT s.Name
FROM Student s CROSS JOIN Hört h CROSS JOIN Vorlesung v
WHERE s.MatrNr = h.MatrNr
   AND h.VorlNr = v.VorlNr
   AND v.gelesenVon = 2137
   AND s.Semester >= 7
```

- Übersetzung in die rel. Algebra (kanonisch):

$$\pi_{s.Name}(\sigma_{s.Semster \geq 7 \wedge v.gelesenVon = 2137 \wedge h.VorNr = v.VorlNr \wedge s.MatrNr = h.MatrNr}(Student \times (Hört \times Vorlesung)))$$

Beispiel: SQL-Anfrage

- Anfrage: *Welche Studierenden (Name) sind mindestens im siebten Semester und haben eine Vorlesung besucht, die von Professor 2137 gelesen wurde?*



- SQL:

```
SELECT DISTINCT s.Name
FROM Student s CROSS JOIN Hört h CROSS JOIN Vorlesung v
WHERE s.MatrNr = h.MatrNr
   AND h.VorlNr = v.VorlNr
   AND v.gelesenVon = 2137
   AND s.Semester >= 7
```

- Übersetzung in die rel. Algebra (kanonisch):

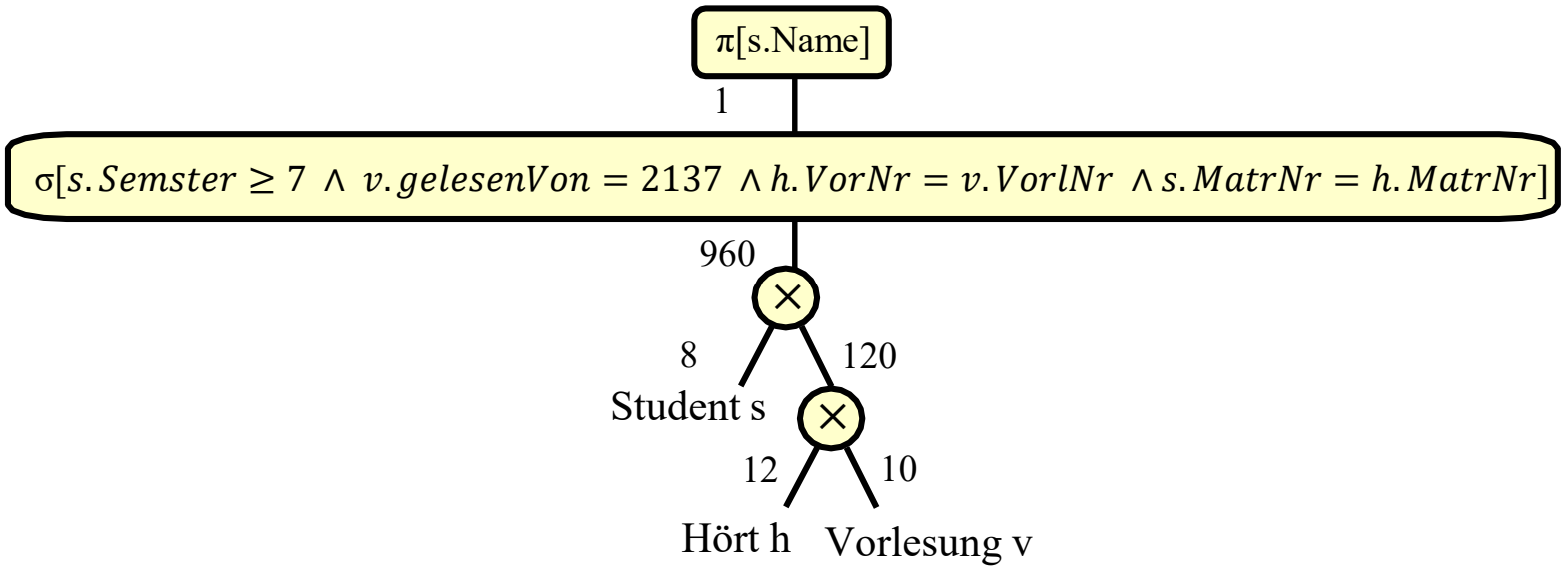
$$\pi_{s.Name}(\sigma_{s.Semster \geq 7 \wedge v.gelesenVon=2137 \wedge h.VorNr=v.VorlNr \wedge s.MatrNr=h.MatrNr}(Student \times (Hört \times Vorlesung)))$$

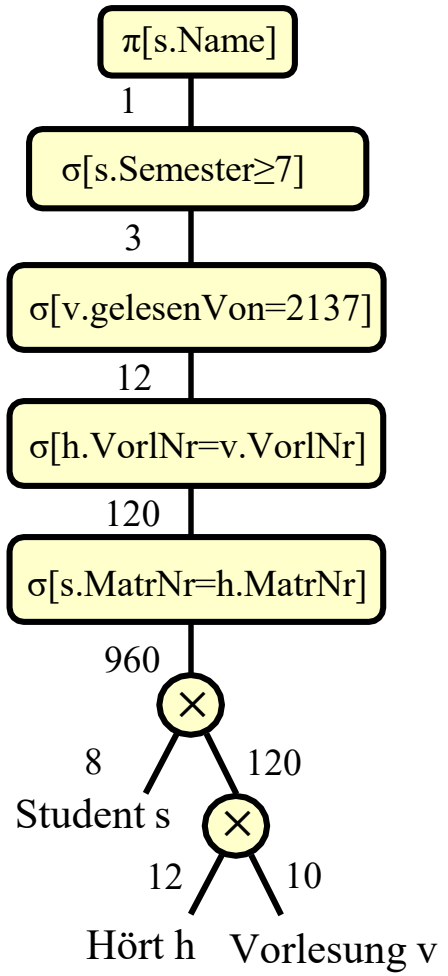
Prinzipien:

So früh wie möglich während der Abfragebearbeitung:

- Reduzierung der Anzahl der Tupel
- Reduzierung der Anzahl der Spalten

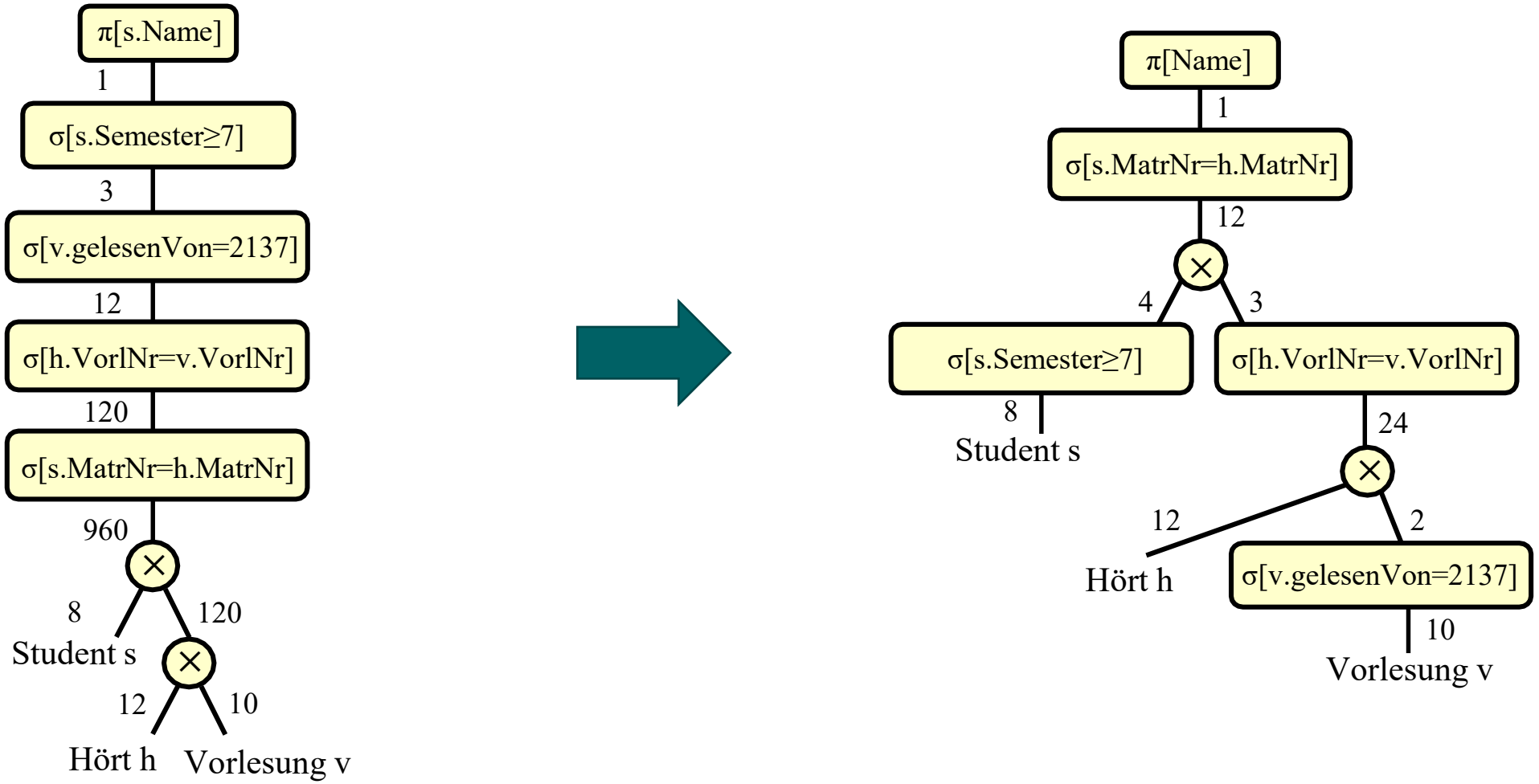
Anfrageoptimierung: Aufbrechen der Selektionen



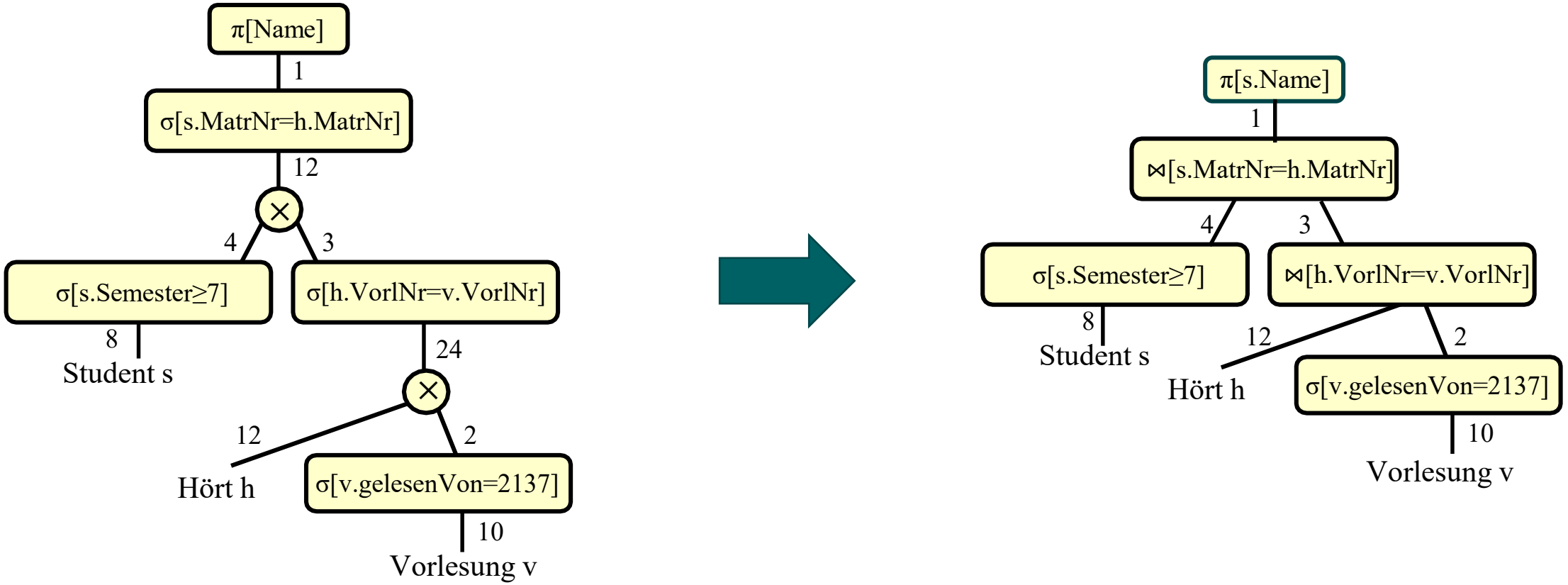


$$\pi_{s.Name}(\sigma_{s.Semster \geq 7}(\sigma_{v.gelesenVon=2137}(\sigma_{h.VorNr=v.VorlNr}(\sigma_{s.MatrNr=h.MatrNr}(Student \times (Hört \times Vorlesung))))))$$

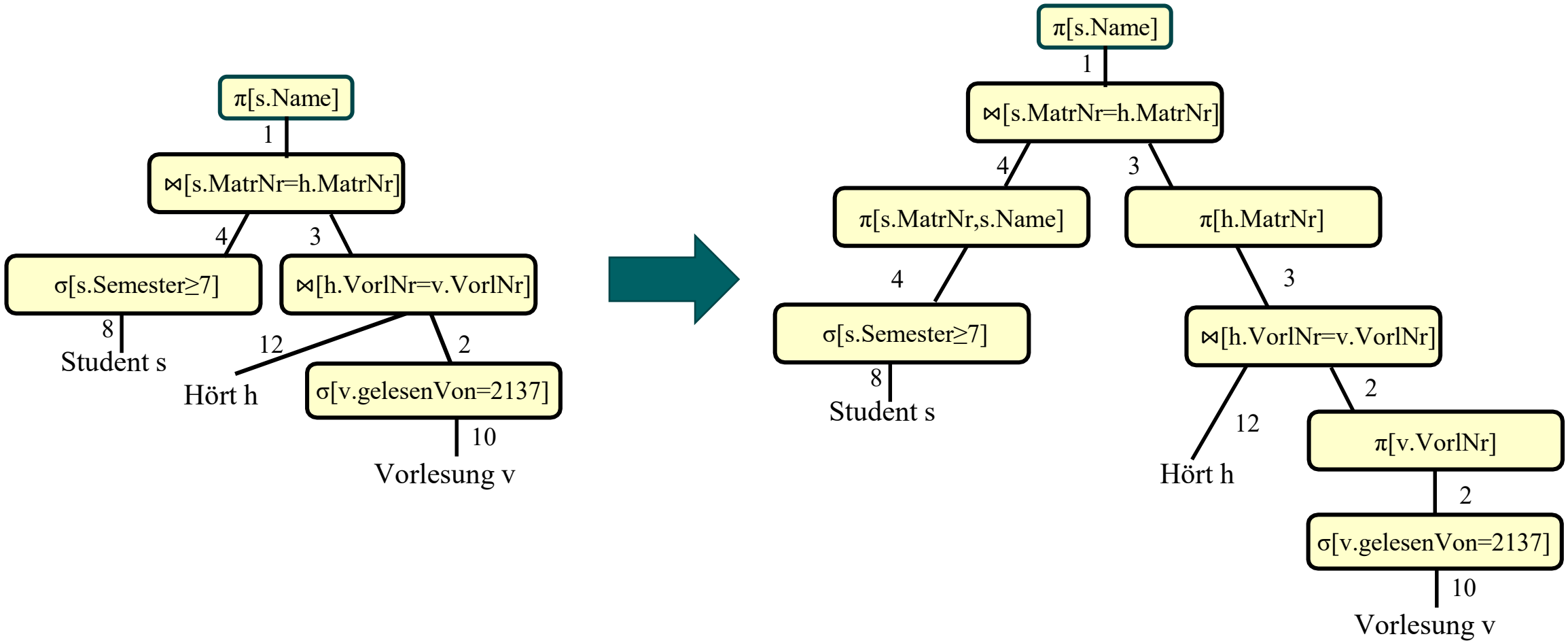
Anfrageoptimierung: Verschieben der Selektionen



Anfrageoptimierung: Zusammenfassen zu Joins

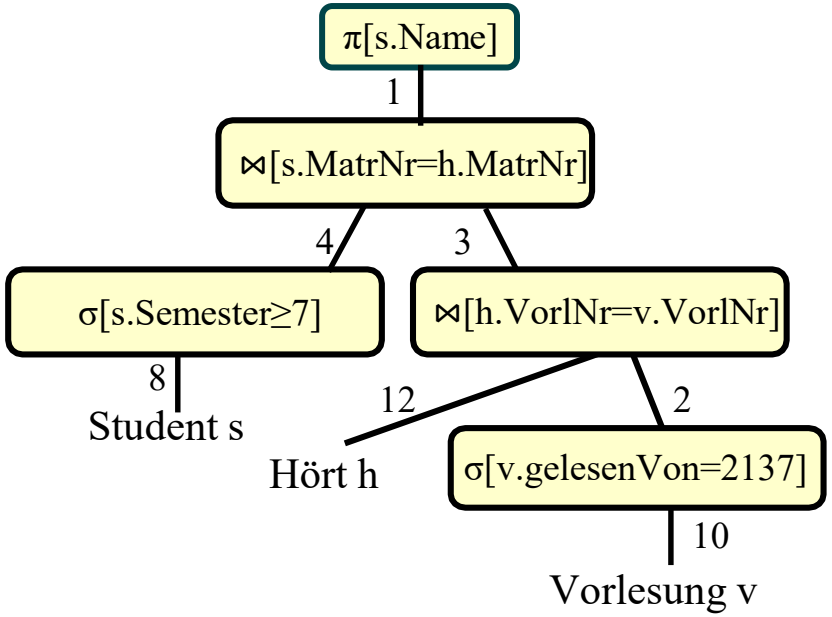


Logische Anfrageoptimierung: Einfügen Zusätzlicher Projektionen

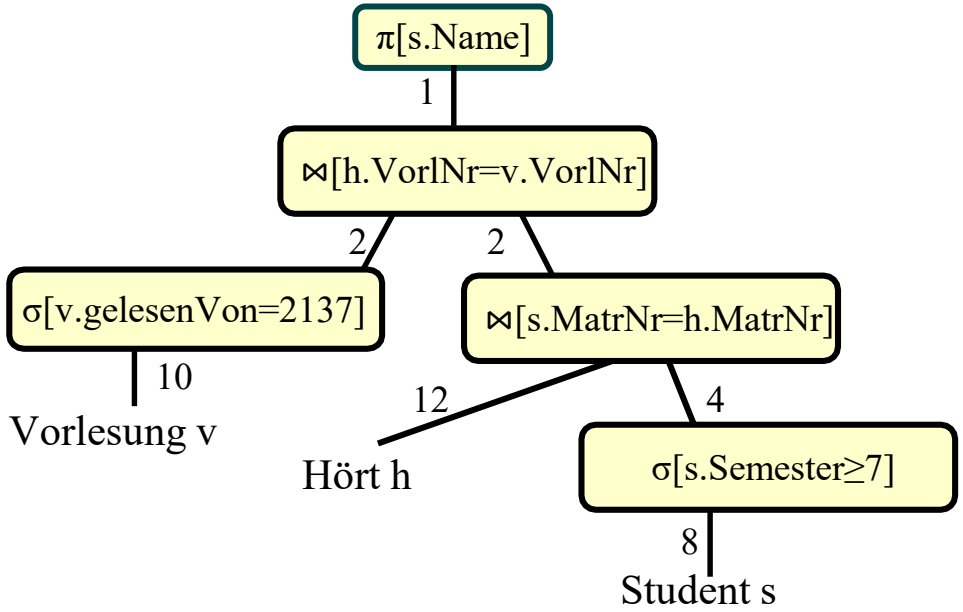


Kosten einer Anfrage: Wie wirkt sich die Join-Reihenfolge aus?

$Student \bowtie (Hört \bowtie Vorlesung))$



$Vorlesung \bowtie (Hört \bowtie Student))$



Betrachtete Tupel: $\Sigma = 40$ $\Sigma = 39$

Die beiden Pläne unterschieden sich nur in der Reihenfolge der ausgeführten Joins. Damit hängt die Performanz von Auswertungsplänen von der Datenverteilung der gespeicherten Informationen ab und der Reihenfolge der Joins ab.

Vom Algebraausdruck zum Ausführungsplan (QEP = Query Execution Plan)

- Ein Algebraausdruck ist noch kein Ausführungsplan
- Zusätzliche Entscheidungen müssen getroffen werden:
  - welche Indexe sollen verwendet werden, z.B. für Selektion oder Join?
  - welche Algorithmen sollen verwendet werde, z.B. Nested-Loop oder Hash Join?
- sollen Zwischenergebnisse materialisiert oder “pipelined” werden? usw.
  - Ausnutzung von Gleichungen der relationalen Algebra
  - Heuristiken auf Basis der Schemainformation
  - Keine Betrachtung der betroffenen Daten
- Für jeden Algebra Ausdruck können mehrere Ausführungspläne erzeugt werden.
- Alle Pläne ergeben dieselbe Relation, unterscheiden sich jedoch in der Ausführungszeit

Logische und physische Anfrageoptimierung

- Die in dem Beispiel angewandte Heuristik, Selektionen möglichst frühzeitig durchzuführen, wird als *push selection* bezeichnet. Weitere wichtige Optimierungen betreffen
  - die Erkennung der Join-Operation aus kartesischem Produkt und Selektion sowie deren Zusammenfassung
  - die Reihenfolge von Join-Operationen bzw. kartesischen Produkten
  - das Erkennen von widersprüchlichen (d.h. leeren) oder redundanten Teilen (gleichen Teilbäumen) in Auswertungsplänen (die nur einmal ausgewertet werden müssen)
- *Logische (algebraische) Anfrageoptimierung :* Optimierungstechniken, die den Auswertungsplan betrachten und "umbauen" (d.h. die Reihenfolge der Operatoren verändern).
- *Physische Anfrageoptimierung*: Die Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation oder die Entscheidung, ob für eine Selektionsoperation ein Index verwendet wird oder nicht und wenn ja, welcher (bei unterschiedlichen Alternativen).

  ⇨ die Auswahl eines geeigneten Algorithmus für jede Operation im Auswertungsplan

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - **Grundlagen: Regelbasierte Anfrageoptimierung**
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Regelbasierte Anfrageoptimierung

- Es gibt zahlreiche Regeln (Heuristiken), um die Reihenfolge der Operatoren im Auswertungsplan zu modifizieren und eine Performanz-Verbesserung zu erreichen, z.B.:
  - Push Selection: Führe Selektionen möglichst frühzeitig (vor Joins) aus
  - Elimination leerer Teilbäume
  - Erkennen gemeinsamer Teilbäume
- Optimierer, die sich ausschließlich nach diesen starren Regeln richten, nennt man *regelbasierte Optimierer*.
- Die Performanz von Auswertungsplänen hängt allerdings auch ganz wesentlich von der Datenverteilung der gespeicherten Informationen ab (siehe nächstes Teilkapitel)

15 Regeln fürs Leben (für die Anfrageoptimierung)

1. Selektionen können aufgebrochen werden: $\sigma_{B1 \wedge B2 \wedge \cdots \wedge Bn}(R) = \sigma_{B1}(\sigma_{B2}(\ldots(\sigma_{Bn}(R))\ldots))$
2. Selektionen sind kommutativ: $\sigma_{Bed_1}(\sigma_{Bed_2}(R)) = \sigma_{Bed_2}(\sigma_{Bed_1}(R))$
3. Geschachtelte Projektionen können eliminiert werden $\pi_X(\pi_Y(\ldots(\pi_Z(R))\ldots)) = \pi_X(R)$ (nur sinnvoll, falls $X \subseteq Y \subseteq \cdots \subseteq Z$)
4. Selektion und Projektion sind vertauschbar, falls die Projektion keine Attribute der Selektionsbedingung entfernt: $\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R))$ falls $attr(B) \subseteq A$
5. Das Kreuzprodukt und Joins sind kommutativ:
$$R \times S = S \times R$$
$$R \bowtie S = S \bowtie R$$
6. Selektion und Join (Kreuzprodukt) können vertauscht werden, falls die Selektion nur Attribute eines der beiden Join-Argumente verwendet: $\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$ sowie $\sigma_B(R \times S) = \sigma_B(R) \times S$ falls $attr(B) \subseteq attr(R)$
7. Projektionen können teilweise in den Join verschoben werden:
$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A_1}(R) \bowtie_B \pi_{A_2}(S))$
wenn $A_1 = attr(R) \cap (A \cup attr(B))$ and $A_2 = attr(S) \cap (A \cup attr(B))$

15 Regeln fürs Leben (für die Anfrageoptimierung)

8. Kommutativität von Vereinigung und Schnitt:

$$R \cup S = S \cup R$$
$$R \cap S = S \cap R$$

9. Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ:

$$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$
$$R \cup (S \cup T) = (R \cup S) \cup T$$
$$R \cap (S \cap T) = (R \cap S) \cap T$$
$$R \times (S \times T) = (R \times S) \times T$$

10. Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden:

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

11. Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden: $\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$

15 Regeln fürs Leben (für die Anfrageoptimierung)

12. Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist (z.B. Equijoin): $\sigma_C\ (R\ \times\ S) =\ R \bowtie_C S$
13. Die Selektion braucht nur in das erste Argument der Mengendifferenz gezogen werden: Es gilt daher (neben Regel 10): $\sigma_B(R) - S$
14. Wenn in einer Vereinigung alle Attribute der Selektion aus einer Relation stammen, braucht die Selektion nur auf diese Relation angewandt werden: $\sigma_B(R \cap\ S)\ =\ \sigma_B(R) \cap S$
15. Wenn S die leere Relation ist, dann ist $R \cup\ S = R.$ Wenn die Selektionsbedingung c für die gesamte Relation R erfüllt ist, dann ist $\sigma_B(R) = R$

Auch an Bedingungen in Selektionen oder Joins können Veränderungen vorgenommen werden, es gelten die üblichen Transformationsregeln, z.B.:

- Kommutativgesetze, Assoziativgesetze, z.B.: $B_1\ \wedge\ B_2\ =\ B_2\ \wedge\ B_1$
- Distributivgesetze, z.B.: $B_1\ \vee\ (B_2\ \wedge\ B_3)\ =\ (B_1 \vee\ B_2)\ \wedge\ (B_1\ \vee\ B_3)$
- De Morgan: $\neg(B_1\ \wedge\ B_2)\ =\ \neg B_1\ \vee\ \neg B_2$

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - **Grundlagen: Kostenbasierte Anfrageoptimierung**
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Kostenbasierte Optimierung

- Reihenfolge Selektions- und Projektionsoperationen durch regelbasierte Optimierung bestimmbar
- Die Reihenfolge der Join-Operationen nicht:
  - *Realen Kosten*
- Die kostenmodellbasierte Anfrageoptimierung behandelt deshalb insbesondere

  die Reihenfolge der Join-Operationen.
- Zwei interessante Fragestellungen:
  - Schätzung der Kosten eines bestimmten Query Execution Planes, insbes. die Schätzung der Größe von Zwischenergebnissen
  - Generierung verschiedener Join-Reihenfolgen bei großen Mengen von Ausgangstabellen (die Anzahl verschiedener QEP ist exponentiell in der Anzahl der Tabellen)

Kostenbasierte Optimierung und Selektivität

- Ziel: Ergebnisse innerhalb kurzer Laufzeit liefern, d.h. (nahezu) optimale QEP
- Ein Kostenmodell ist notwendig, um den besten Auswertungsplan auszuwählen
- Ein Kostenmodell stellt Funktionen zur Verfügung, die den Aufwand, d.h. die Laufzeit, der Operationen der physischen Algebra abschätzen.
- Bei der Aufwandsbestimmung spielt in vielen Fällen eine Rolle, wie viele Tupel sich bei Auswertung einer Bedingung qualifizieren
- Die *Selektivität* (sel) eines Anfrage schätzt die Anzahl der qualifizierenden Tupel relativ zur Gesamtanzahl der Tupel in der Relation.
- Der Anteil der qualifizierenden Tupel heißt Selektivität (sel)
  - Die Selektivität ist kein Kostenmaß.
  - Die Laufzeit einer Operation hängt von der Eingabegröße ab, und damit von der Selektivität der im Operatorbaum darunter liegenden Operationen.

Selektivität (sel)

- Für die Selektion und den Join ist sie folgendermaßen definiert:

  - **Selektion** mit Bedingung *B*: $$sel_B = \frac{|\sigma_B(R)|}{|R|}$$

    (relativer Anteil der Tupel, die Bedingung *B* erfüllen)

  - **Join** von *R* und *S*: $$sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

    (Anteil relativ zur Kardinalität des Kreuzprodukts)

Schätzung der Zwischenergebnisse

Selektion mit Bedingung B : $sel_B = \frac{|\sigma_B(R)|}{|R|}$

Join von R und S : $sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$

Die Selektivität muss geschätzt werden, für Spezialfälle gibt es einfache Methoden:

- Die Selektivität von $\sigma_{R.A = c}$) ,Vergleich mit einer Konstante c) beträgt $1 / |R|$, falls A ein *Schlüssel* ist.
- Falls A kein Schlüssel ist, aber Werte *gleichverteilt* sind, ist

  $sel(\sigma_{R.A=c}) = \frac{1}{I}$) I ist die Anzahl der unterschiedlichen Attributwerte)
- Besitzt bei einem Equijoin $R \bowtie_{R.A = S.B} S$ das Attribut A Schlüsseleigenschaft, kann die Größe des Join-Ergebnisses mit $|S|$ abgeschätzt werden, da jedes Tupel aus S maximal einen (genau einen bei *referential integrity*) Joinpartner findet. Also: $sel_{RS} = 1/|R|$

Schätzung der Zwischenergebnisse

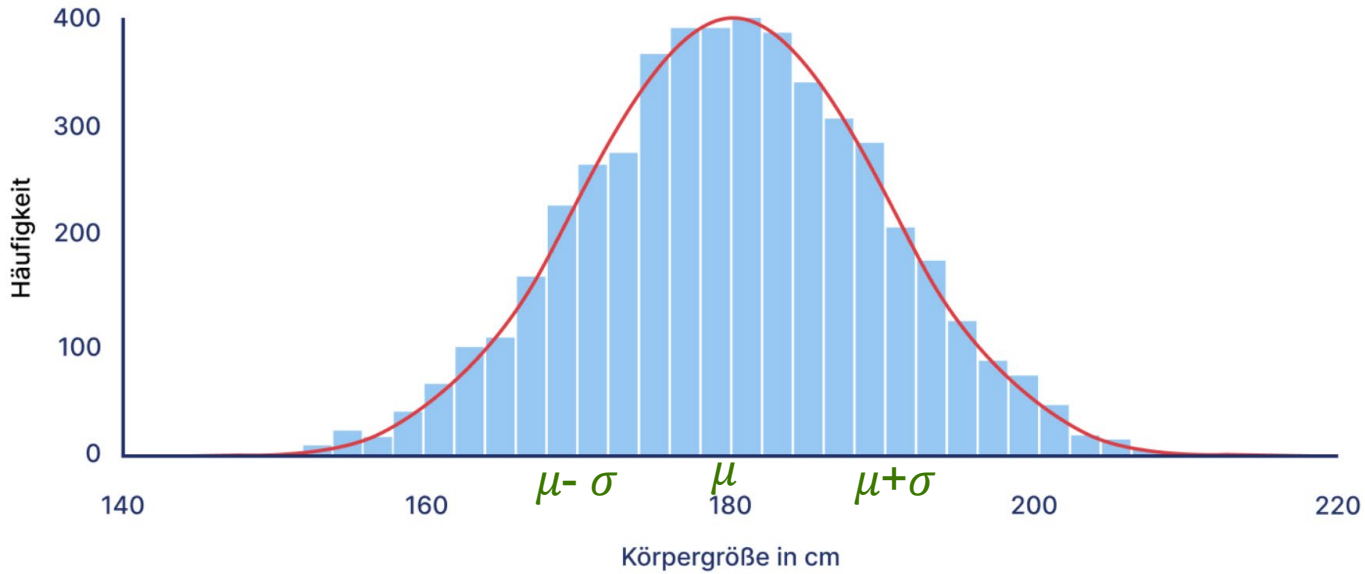
- Boolesche Verknüpfungen von Bedingungen (bei stochastischer Unabhängigkeit der Mengen):
  - logisches UND : $sel(\sigma_{B_1 \wedge B_2}) = sel(\sigma_{B_1}) \cdot sel(\sigma_{B_2})$
  - logisches ODER : $sel(\sigma_{B_1 \vee B_2}) = sel(\sigma_{B_1}) + sel(\sigma_{B_2}) - sel(\sigma_{B_1}) \cdot sel(\sigma_{B_2})$
  - logisches NICHT : $sel(\sigma_{\neg B_1}) = 1 - sel(\sigma_{B_1})$
- Nur in Spezialfällen kann man solche einfachen Rechenregeln anwenden. Im allgemeinen braucht man andere Methoden zur Selektivitätsabschätzung:

- Drei Grundsätzliche Arten von Schätzmethoden:
  1. Parametrische Verteilungen
  2. Histogramme
  3. Stichproben

Selektivität: Parametrische Verteilung

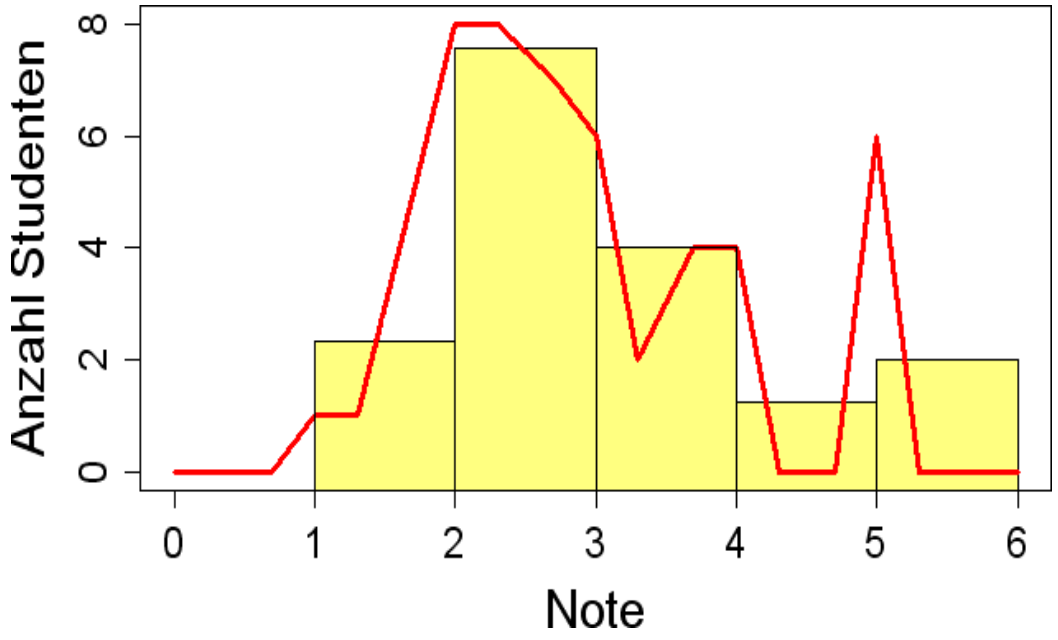
- Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.
- Z.B. Normalverteilung $N(\mu;\sigma$^2) mit den Parametern μ (Mittelwert) und σ^2 (Varianz).

- Probleme: Wahl des Verteilungstyps (Normalverteilung, Exponentialverteilung…) und Wahl der Parameter, besonders bei mehrdimensionalen Anfragen (also z.B. bei Selektionen, die sich auf mehrere Attribute beziehen)



Selektivität: Histogramme

- Unterteile den Wertebereich des Attributs in Intervalle und zähle die Tupel, die in ein bestimmtes Intervall fallen
  - Equi-Width-Histograms: Intervalle gleicher Breite (starker Ungleichverteilung -> viele Unterteilungen in schwach besetzten Bereichen)
  - Equi-Depth-Histograms: Unterteilung so, dass in jedem Intervall gleich viele Tupel sind
- Equi-Depth-Histogramme = bessere Schätzgenauigkeit auf, haben aber auch einen höheren Verwaltungsaufwand.
- Z.B.: ORACLE benutzt Equi-Depth-Histogramme



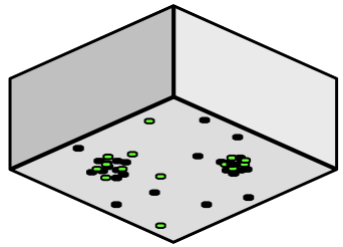
Selektivität: Stichproblem

Stichproben:

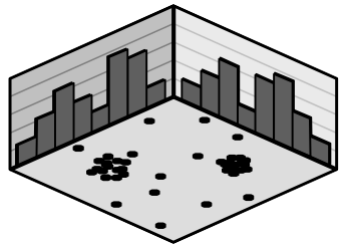
- Sehr einfaches Verfahren
- Ziehe eine zufällige Menge von n Tupeln aus einer Relation, und betrachte deren Verteilung als repräsentativ für die gesamte Relation.
- Problem der Größe des Stichprobenumfangs n:
  - n zu klein: Wenig repräsentative Stichprobe
  - n zu gross: Ziehen der Stichprobe erfordert zu viele „teure“ Zugriffe auf den Hintergrundspeicher

Selektivität: Anfragen über mehrere Attribute (mehr-dimensionale Anfragen)

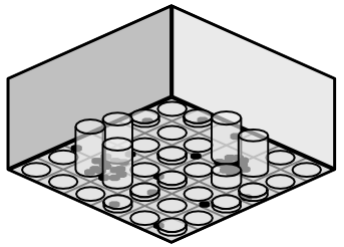
- Stichproblen:
  - Problem: Genauigkeit abhängig von der Samplegröße



- 1d Histogramme
  - Problem: Annahme der Unabhängigkeit zwischen den Attributen
- Multi-d Histogramme
  - Problem: Anzahl der Gridzellen steigt exponentiell mit d



- Parametrische Methoden
  - Problem: nur für max. 2-3 Attribute geeignet



5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - **Join-Reihenfolgen**
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Join Reihenfolgen (1)

- Kostenbasierte Optimierung kann verwendet werden, um die beste Join Reihenfolge herauszufinden.

- Join Reihenfolgen der Relationen entstehen durch:
  - Assoziativgesetz: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
  - Kommutativgesetz: $R \bowtie S = S \bowtie R$

- Die Join Reihenfolge hat große Auswirkung auf Effizienz:
  - Größe der Zwischenergebnisse
  - Auswahlmöglichkeit der Algorithmen (z.B. vorhandene Indexe wiederverwenden)

Join Reihenfolgen (2)

- Wieviele Reihenfolgen gibt es für: $R_1 \bowtie R_2 \bowtie \ ... \bowtie R_m$?
- Assoziativgesetz:
  - Operatorbaum: es gibt C_{m-1} volle binäre Bäume mit m Blättern (es gibt C_{m-1} Klammerungen von m Operanden)
  - dabei ist $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{(n+1)!n!}$, $n > 0$ die Catalan-Zahl:
- Kommutativgesetz:
  - Blätter des Operatorbaums sind die Relationen $R_1, R_2, \ldots, R_m$
  - für jeden Operatorbaum gibt es $\mathrm{m}!$ Permutationen
- Anzahl der Join-Reihenfolgen für m Relationen:

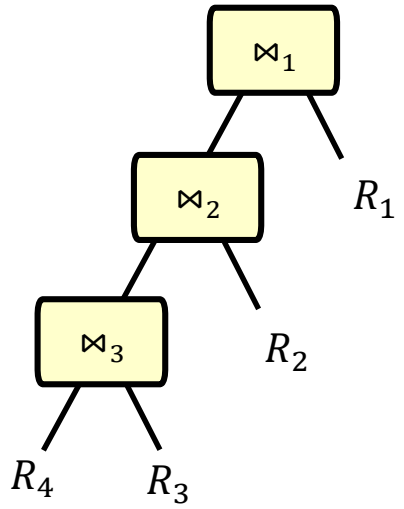
$$m!\, C_{m-1} = \frac{2(m-1)!}{(m-1)!}$$

Join Reihenfolgen (3)

- Anzahl aller Join-Reihenfolgen wächst sehr schnell an:
  - $m = 3$: 12 Reihenfolgen
  - $m = 7$: 665.280 Reihenfolgen
  - $m = 10$: > 17.6 Milliarden Reihenfolgen
- Dynamic Programming Ansatz:
  - Laufzeit Komplexität: $O(3^m)$
  - Speicher Komplexität: $O(2^m)$
- Beispiel: $m = 10$
  - Anzahl der Join-Reihenfolgen: 17.6 × 109
  - Dynamic Programming: $O(3^m)$ = 59049
- Trotz Dynamic Programming bleibt Aufzählung der Join-Reihenfolgen teuer.

Join Reihenfolgen (4)

- Beschränkung auf Left-deep Join Reihenfolgen
  - rechter Join-Operator ist immer eine Relation (nicht Join-Ergebnis)
  - dadurch ergeben sich sog. left-deep Operatorbäume (im Gegensatz zu “bushy”, wenn alle Operatorbäume erlaubt sind)
- Anzahl der left-deep Join Reihenfolgen für m Relationen: $O(m!)$
- Dynamic Programming: Laufzeit $O(m!)$
- Vergleich für 10 Relationen:



	bushy	left-deep	m = 10	bushy	left-deep
#Baumformen	1	C_{m-1}		1	4.862
#Join Reihenfolgen	$\frac{2(m-1)!}{(m-1)!}$	m!		1.76×10^{10}	3.63×10^{6}
Dynamic Programming	$O(3^m)$	$O(m\ 2^m)$		59.049	10.240

- Aber:
  keine Garantie auf optimale Reihenfolge

Greedy Algorithmus für Join Reihenfolgen (1)

- Ansatz: In jedem Schritt wird der Join mit dem kleinsten Zwischenergebnis verwendet.
- Überblick: Greedy Algorithms für Join Reihenfolge:
  - nur left-deep Join Reihenfolgen werden betrachtet
  - Relationen-Paar mit dem kleinsten Join Ergebnis kommt zuerst dran
  - Dann wird immer die Relation ausgewählt, die mit dem vorhanden Operatorbaum das kleinste Join-Ergebnis erzeugt

- Algorithmus: Join Reihenfolge von $S = \{R_1, \dots, R_m\}$
  1. $O \leftarrow R_i \bowtie R_j$, sodass $|R_i \bowtie R_j|$ minimal ist $(i \neq j)$
  2. $S = S - \{R_i, R_j\}$
  3. **while** $S \neq \emptyset$ **do**
     a) wähle $R_i \in S$ sodass $|O \bowtie R_i|$ minimal ist
     b) $O \leftarrow O \bowtie R_i$
     c) $S = S - \{R_i\}$
  4. **return** Operatorbaum O

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - **Ein Algorithmus für die Anfrageoptimierung**
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Ein Algorithmus für die Anfrageoptimierung

1. Zerlege komplexe Selektions -Operationen in eine Kaskade einfacher Selektionen (Regel 1).
2. Verschiebe Selektionen so weit wie möglich den Abfragebaum hinunter (Regeln 2, 4, 6, 10, 13, 14).
3. Ordne Blattknoten nach restriktiven Selektionen und vermeide Kreuzprodukte (Regeln 5 und 9).
4. Kombiniere Kreuzprodukte und Selektionen zu JOINs, wenn möglich (Regel 12).
5. Verschiebe und teile Projektions-Operationen, um nur benötigte Attribute zu behalten (Regeln 3, 4, 7, 11).
6. Identifiziere Teilbäume, die von einem einzigen Auswertungs-Algorithmus ausgeführt werden können (siehe nächsten Abschnitt in der Vorlesung).

Nach Elmasri & Navathe, Fundamentals of Database Systems, 7th Edition)

5. Relationale Anfragebearbeitung

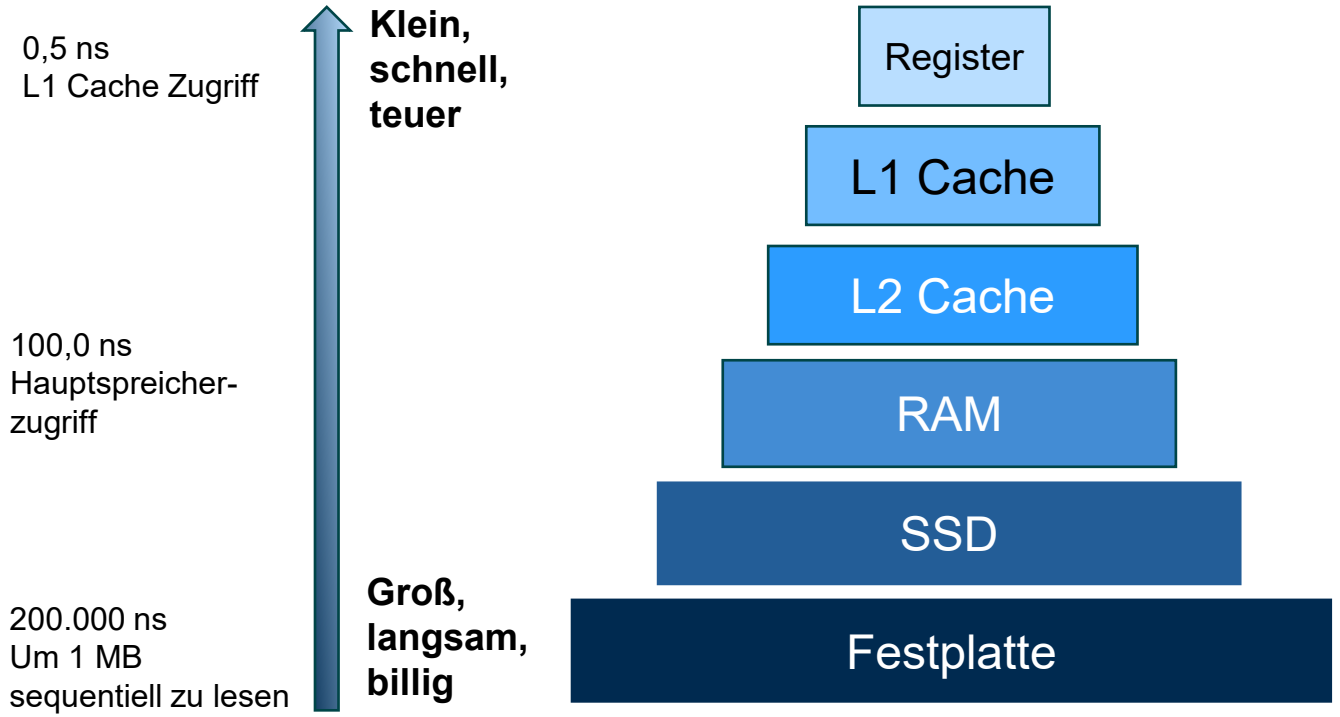
1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. **Algorithmen für Basisoperationen**
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Speicherhierachie

Aufgabe: Finde geeignete Algorithmen & Datenstrukturen für einzelne Operationen
Herausforderungen: Physische Datenorganisation (Cache, RAM, Festplatte, SSD)

Vereinfachung:

- CPU-beschränkt: Das System aus CPU, Arbeitsspeicher und Bus bildet den Hauptengpass
- I/O-beschränkt: Hintergrundspeicher und I/O bilden den Hauptengpass
- Parallelität zwischen CPU und Hintergrundspeicher: Algorithmen zur Anfragebearbeitung “multithreaded” implementiert



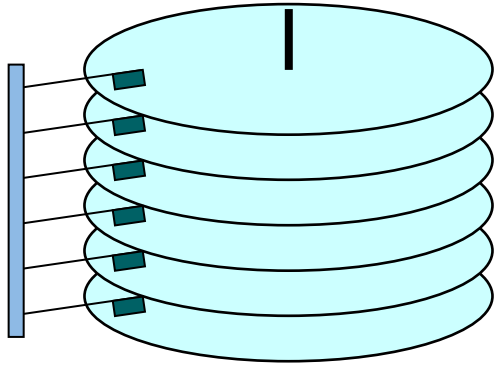
Optimierungsziele

Optimierung auf verschiedenen Ebenen:

- Reduzieren der I/O-Kosten durch
  - gute Ausnutzung eines Puffers
  - Verwendung von Indexen
  - durch vorberechnete Joins (Cluster)
- Reduzieren der Vergleiche (CPU-Kosten, kann insbesondere bei komplexen Datentypen sehr wichtig sein)
- Reduzieren der Kommunikationskosten (besonders wichtig in verteilten DBS)
- Verbesserung der Abarbeitungsreihenfolge in einem Mehrwege-Join

Speicherhierarchie

- Motivation
  - Persistente Speicherung von Daten: „Daten überleben Prozesse“
  - Datenmengen sind größer als der Hauptspeicher: viele GB, TB, PB
  - Gemeinsame Nutzung von Daten: nebenläufiges Arbeiten

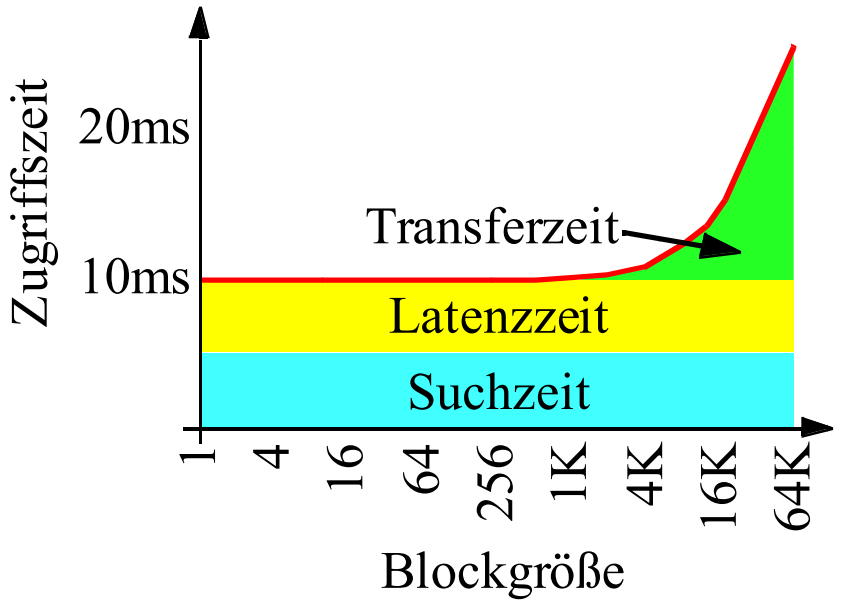


- Festplatten/SSD als gebräuchliche Sekundärspeicher
  - Platten: Übereinanderliegende Platten mit magnetischen / optischen Oberflächen
    - Zur Adressierung sind die Platten in Spuren und Sektoren eingeteilt
    - Mechanische Bewegungen
      - Platten rotieren um gemeinsame Achse
      - Schreib-Lese-Köpfe werden zwischen den Platten synchron in radialer Richtung bewegt
  - SSD:
    - Unterschied zu magnetischen Festplatten: SSDs haben Zellenorganisation
    - Schnelle wahlfreier Zugriff bei SSDs
    - "Wear Leveling" zur Verteilung von Schreibvorgängen und Vermeidung von Verschleiß
    - Feines blockweises lesen (4-8kb Block), grobes schreiben („indirection unit“, z.B., 16 KB)
    - SSDs sind teurer als Festplatten

Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten (nicht für SSDs)
  - Armpositionierung: Suchzeit (ca. 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (ca. 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

- Blockorientierter Zugriff (gilt auch für SSDs)
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB



Algorithmen für Basisoperationen: Selektion und Projektion

Selektion

- Sequentieller Scan oder Verwendung von Indexstrukturen
- Auswahl hängt unter anderem vom Anfragetyp ab
  - Punktanfragen („exact match query“)
    - SELECT * FROM Stud WHERE Matrnr = 123456
  - Bereichsanfragen („range query“)
    - SELECT * FROM Stud WHERE 123456 <= Matrnr AND Matrnr <= 123465

Projektion

- Teiloperationen:

  Projektion auf die Projektionsattribute & Eliminierung von Duplikaten
- Aufwendigere Teiloperation: Eliminierung von Duplikaten
  - ➢ Projektion durch Sortieren
  - ➢ Projektion durch Hashverfahren

Algorithmen für Basisoperationen: Join

Join

- Wichtigste Operation, insbesondere in relationalen DBS:
  - komplexe Benutzeranfragen
  - Normalisierung der Relationen
  - verschiedene Sichten (“views”) auf die Basisrelationen

Beispiele von Join Algorithmen:

1. *Nested Loop Join*:
   - erzeuge alle Tupel des kartesischen Produktes und prüfe die Join-Bedingung
2. *Nested Block Loop Join*
   - *Berücksichtigt die Block-Struktur des verwendeten Speichers*
3. *Indexed Loop Join*:
   - betrachte alle Tupel der einen Relation und greife auf die Joinpartner über einen passenden Index der anderen Relation zu
4. *Hash-Join:*
   - *Join-Partner eines Tupels wird mit Hilfe eines Hash-Verfahrens gesucht*
5. *Sort Merge Join*:
   - sortiere beide Relationen nach dem Joinattribut und filtere passende Paare

Es gibt i.A. kein *bester* Join-Algorithmus! Es ist von der jeweiligen Situation abhängig (Datenverteilung, Existenz von Index, Anfrage usw.), welcher Algorithmus sinnvoll ist.

Annahmen

- Wir schauen uns im folgenden die Join-Operation anhand folgenden Beispiels an:
- Zur Vereinfachung: equi join

```
select *
   from R r, S s
   where r.A = s.B
```

also $R \bowtie_{A=B} S$

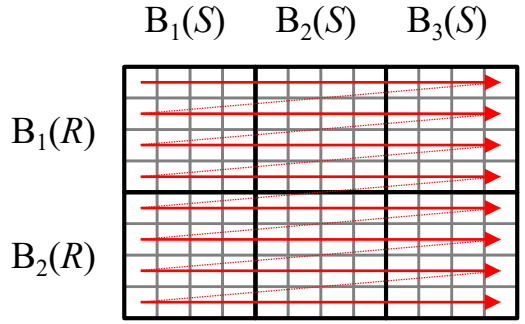
- Die Relationen R und S sind in Blöcken $B_i(R)$ und $B_i(S)$ auf dem Hintergrundspeicher abgelegt.

- Notation:
  - $\otimes$ sei ein Operator, der zwei Tupel zu einem Tupel verknüpft
  - Sei r ein Tupel und A ein Attribut. Dann ist r(A) der Attributwert von A und r-r(A) das Tupel r ohne den Attributwert von A.
  - Sei R Relation und A ein Attribut, dann ist R\A die Relation R ohne das Attribut A

Einfacher Nested Loop Join

for each Tupel $r \in R$ do
    for each Tupel $s \in S$ do
        if $r(A) = s(A)$ then
            $Result := Result \cup \{(r - r(A)) \otimes s\}$

Matrixnotation (S 3 Blöcke, R 2 Blöcke)



- Der einfache Nested Loop Join entspricht der Bildung des kartesischen Produktes in kanonischer Ordnung mit anschließender Selektion.
- Die Laufzeit ist $|S| \cdot |R|$. Relation S wird $|R|$ mal eingelesen: Performanz ist deshalb inakzeptabel
- S wird als *innere* Relation und R als *äußere* Relation bezeichnet

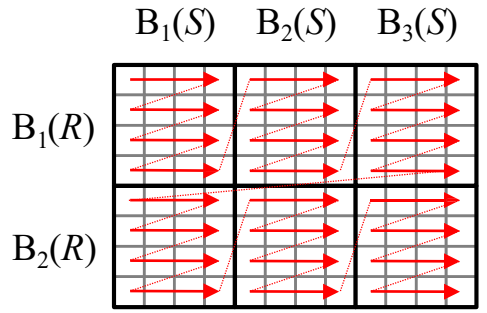
- Matrixdarstellung der Joinoperation (stellt Reihenfolge von Block- und Tupelpaarungen dar)
- Nested Loop Joins sind geeignet für alle Join-Prädikate θ (’=‘, ’<‘, ’>‘, ’≤‘, usw.)

Nested Block Loop Join

- Beobachtung: Die innere Relation S wird $|R|$-mal gelesen (das kann teuer sein).
- Reduktion der I/O-Kosten durch blockweise Verarbeitung der Relationen

for each Block B_R in R do {
    lade Block B_R;
    for each Block B_S in S do {
        lade Block B_S;
        for each Tupel r in B_R do
            for each Tupel s in B_S do
                if $r(A) = s(A)$ then
                    $Result := Result \cup \{(r - r(A)) \otimes s\}$
    }
}

Matrixnotation



Nested Block Loop Join: Beispiel

Relation S

Angesteller	Gehaltsgruppe	
Müller	1	$B_S(1)$
Schneider	2	
Schuster	1	$B_S(2)$
Schmidt	2	
Schütz	1	$B_S(3)$

Relation R

Gehaltsgruppe	Gehalt	
1	10.000	$B_R(1)$
2	20.000	
3	30.000	$B_R(2)$

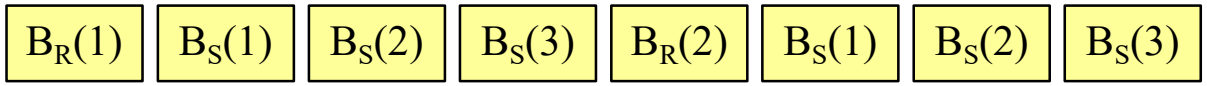
- Blockzugriffe $= b_R + b_S \cdot b_R$ =8, (b_R = Anzahl der Blöcke der Relation R ist)

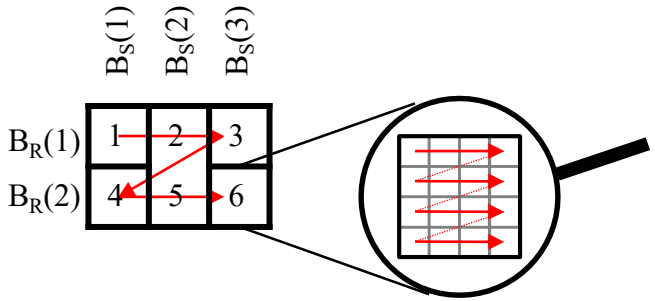
- Empfehlung: Die kleinere Relation sollte die äußere sein (ohne Cache).
- Wenn ein Cache (= Hauptspeicherplatz für viele Blöcke gleichzeitig) zur Verfügung steht, kann u.U. die kleinere Relation ganz im Hauptspeicher gehalten werden
  - Dann nur $b_R + b_S$ Zugriffe, im Beispiel: 5 Zugriffe

Cache Strategien für Nested-Block-Loop-Join

Strategíe 1

- Seiten der inneren Relation (S) im Cache halten
- Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R
- (▭: Zugriff Platte)

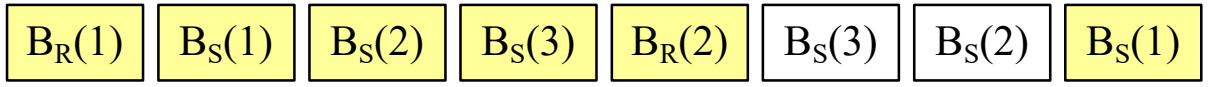


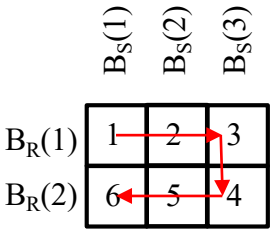


Cache Strategien für Nested-Block-Loop-Join

Strategie 2

- Seiten der inneren Relation (S) im Cache, aber innere Relation jedes zweite mal rückwärts
- Pro Durchlauf der äußeren Schleife werden (|C|-1) Blockzugriffe
- eingespart (ab 2. Durchlauf)
- |C| = Anzahl Blöcke, die in den Cache passen, ein Cache-Block wird jeweils für äußere Relation (R) benötigt
- Blockzugriffe: BR+ BR · (BS - |C| + 1) + |C| - 1
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R





Cache Strategien für Nested-Block-Loop-Join

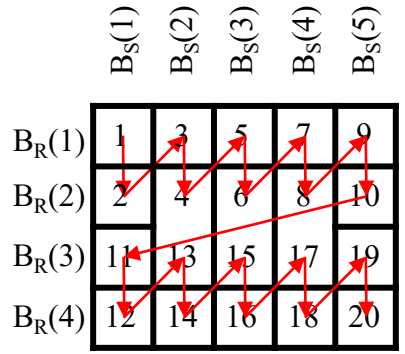
Strategie 3

- |C|-1 Blöcke der äußeren Relation werden in den Cache eingelesen, zu jedem Block der inneren Relation werden diese Blöcke gejoint
- Blockzugriffe:

$$B_R + B_S \cdot \left\lceil \frac{B_R}{|C| - 1} \right\rceil$$

- Beispiel: 2 Seiten Cache für R, 1 Seite Cache für S

$B_R(1)$	$B_R(2)$	$B_S(1)$	$B_S(2)$	$B_S(3)$	$B_S(4)$	$B_S(5)$
$B_R(3)$	$B_R(4)$	$B_S(1)$	$B_S(2)$	$B_S(3)$	$B_S(4)$	$B_S(5)$



Strategie 3 - Algorithmus

```
for i := 1 to B_R step |C| −1 do
  Lade Block B_R(i) ... B_R (i+ |C| −2)
  for each Block B_S ∈ S do
    Lade Block B_S
    for each Tupel r ∈ B_R(i) ... B_R (i+ |C| −2) do
      for each Tupel s ∈ B_S do
        if r.A = s.B then
          result := result ∪ {(r−r(A)) ⊗ s}
```

- Leistung:
  - |R|*|S| Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
  - Effizienteste Ausführung von θ–Joins mit θ ≠ '=' (also allen Joins außer Equi-Joins)

Sort-Merge Join: Beispiel – Annahme R und S sind sortiert

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider
2	20.000	Schmidt

Zweistufiger Algorithmus

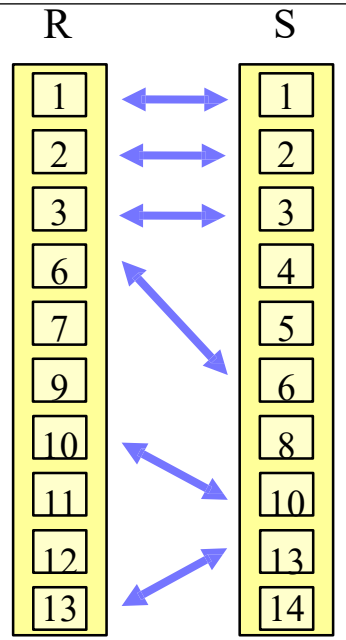
1.Schritt:

sortiere R bzgl. Attribut A

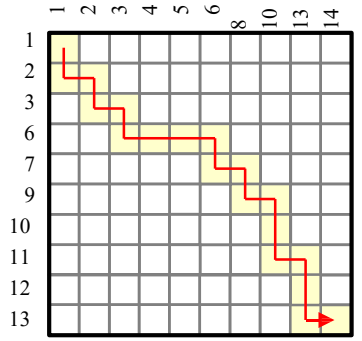
sortiere S bzgl. Attribut B

2.Schritt:

```
Result :=∅;
s:=erstes Tuple in S;
r:=erstes Tuple in R;
while (r≠null ∧s≠null)
        while (s≠null∧ r(A) ≥s(A))
                if ( r(A)=s(A)) Result :=Result∪{(r−r(A)⊗s}
                s:=nächstes Tuple in S nach s;
        r:=nächstes Tuple in R nach r;
return Result;
```



Matrixnotation



- Achtung: Dieser Algorithmus funktioniert nur, falls R auf dem Joinattribut A keine Duplikate enthalten.
- Wie muss der Algorithmus erweitert werden um Duplikate zu erfassen?

Sort-Merge Join (Duplikate möglich in beiden Relationen)

1. Sort-Phase (falls Relationen nicht schon sortiert)
   - Sortiere Relation R bzgl. Attribut A;
   - Sortiere Relation S bzgl. Attribut A;

2. Merge-Phase: Paralleles Durchlaufen der Relationen

$Result := \emptyset;$
$Rvalue, Rtuple, Rgroup := \text{nextGroup}(R, \text{erstes Tupel in R});$
$Svalue, Stuple, Sgroup := \text{nextGroup}(S, \text{erstes Tupel in S});$
while $(Rgroup \neq \emptyset \wedge \text{Sgroup} \neq \emptyset)$
  if $(Rvalue = Svalue)$
    $Result = Result \cup (Rgroup \setminus A) \times Sgroup)$
    $Rvalue, Rgroup := \text{nextGroup}(R, Rtuple);$
    $Svalue, Sgroup := \text{nextGroup}(S, Stuple);$
  else if $(Rvalue < Svalue)$
    $Rvalue, Rtuple, Rgroup := \text{nextGroup}(R, Rtuple);$
  else $Svalue, Stuple, Sgroup := \text{nextGroup}(S, Stuple);$
return $Result;$

Hilfsfunktion zur Bestimmung der nächsten Gruppe:

function $value, t, group$ $\text{nextGroup}(T, tuple)$
  $group := \emptyset;$
  $t := tuple;$
  $value := t\ (A)\ ;$
  while $(t \neq null \wedge \text{value} = t\ (A))$
    $group := group \cup \{t\};$
    $t := nächstes\ Tuple\ in\ \ T\ nach\ t;$
  return $value, t, group;$

Analyse:

- Sortieren der Relationen kostet $O(|R| \log |R| + |S| \log |S|)$
- Sortieren ist nicht notwendig, wenn bereits Index existiert
- Wenn beide Relationen keine Duplikate in den Join-Attributen habe wird jede Relation genau einmal durchlaufen: $O(|R| + |S|)$ Vergleiche
- Bei Duplikaten in beiden Relationen bestimmt das kartesische Produkt die worst case Performanz.
- Weitere Optimierungen: Sortierung und Merging kombinieren, Blöcke berücksichtigen

Index Join

- Grundidee: Ersetzt innere Schleife durch Suche in einer Indexstruktur
- Index-Join kann in folgendem Fall verwendet werden:
  - Equi-Join zwischen R und S bezüglich eines (ggf. nicht-eindeutigen) Attributs B.
  - S hat einen *Index* auf dem Attribut A.
  - R kann ohne Index gespeichert sein.
- Folgende Schleife berechnet den Join:

  for each Tupel r in R do

  find join partner s in index on *S(A)*

  $$Result := Result \cup ((r - r(A)) \otimes s)$$

- Kostenschätzung:
  - $|R| \cdot cost\left(Indexzugriff\ auf\ S(\hat{A})\right)$
  - Also z.B. $|R| \cdot \log|S|$ im Fall der Verwendung von B-Bäumen (siehe Indexstrukturen)

Einfacher Hash-Join

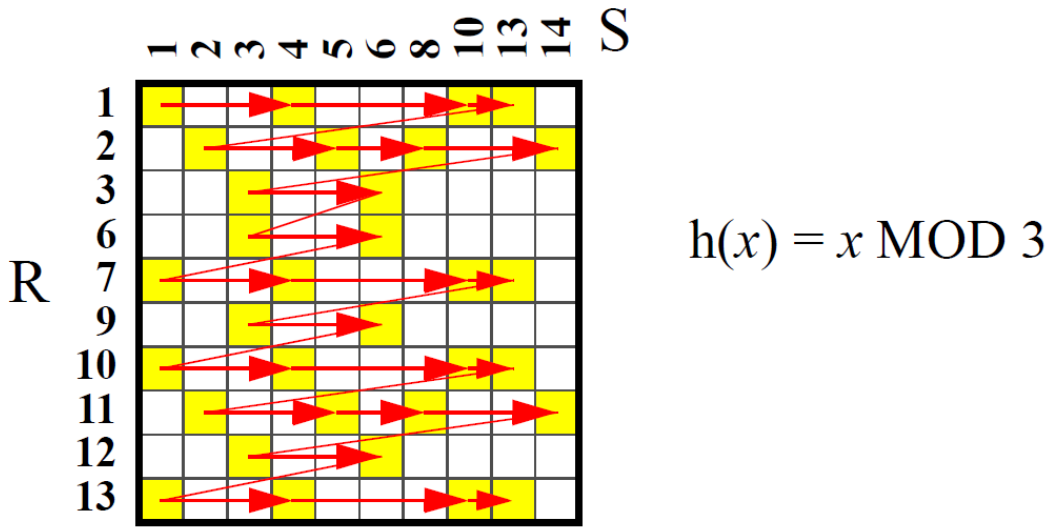
Weitere Idee: Erzeugung eines Index zur Laufzeit

Reduktion des CPU-Aufwandes bei der Join-Berechnung:

- Der Join-Partner eines R-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, statt sequentiell mit jedem Tupel der S-Relation zu vergleichen
- Zu diesem Zweck wird die S-Relation gehasht, d.h. zu allen Tupeln der Hash-Key bestimmt und die Tupel in einer Tabelle unter diesem Key eingetragen
- Nicht alle S-Tupel, die den passenden Hash-Key haben, sind Join-Partner eines R-Tupels, aber alle Join-Partner haben denselben Hashkey (**Bedingung der Hashfunktion!**)
- Im Idealfall soll der Join im Hauptspeicher ablaufen: die Hashtabelle soll für die kleinere Relation erzeugt werden.
- Hash-Join Verfahren können nur für Equi-Join und Natürlichen Joint effizient genutzt werden.

Einfacher Hash-Join

for each Tupel s in S do { /* Erzeugen der Hashtabelle HT */
    berechne adr = Hash(s);
    speichere das Tupel s in $HT[adr]$
}
for each Tupel r in R do { /* Prüfen in der Hashtabelle HT */
    berechne adr = Hash(r);
    for each Tupel s in $HT[adr]$ do {
        if $r(A) = s(A)$ then
            $Result := Result \cup \{(r - r(A) \otimes s\}$
    }
}



$h(x) = x \text{ MOD } 3$

Leistung

- hängt stark ab von der Güte der Hashfunktion: O(|R| + |S|) im Idealfall
- verschlechtert sich, wenn Werte ungleichmäßig belegt sind
- Modifikation ist notwendig, wenn Hauptspeicher zu klein (kleiner als R)

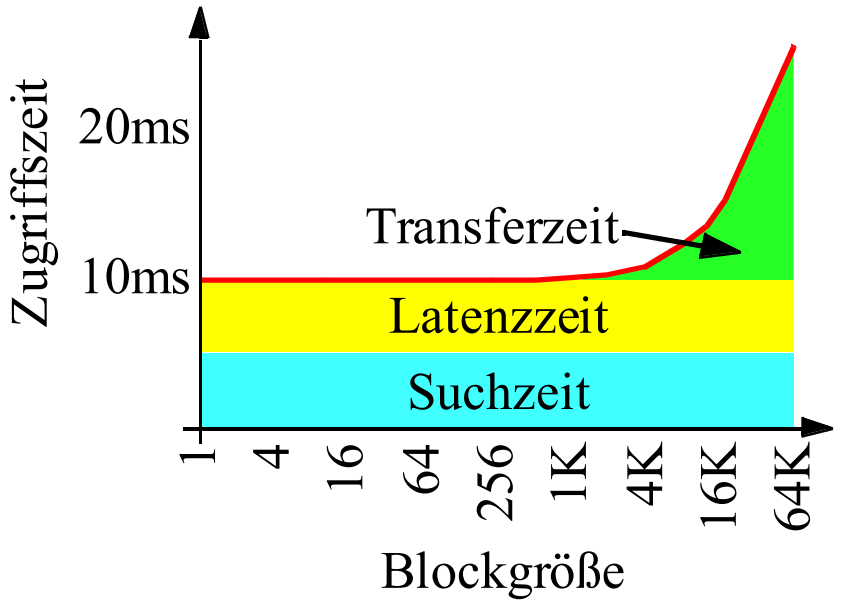
5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - **Indexstrukturen für eindimensionale Daten**
   - Indexstrukturen für mehrdimensionale Daten

Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten (nicht für SSDs)
  - Armpositionierung: Suchzeit (ca. 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (ca. 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

- Blockorientierter Zugriff (gilt auch für SSDs)
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB



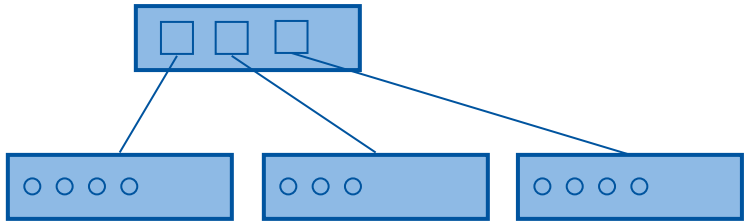
Verwendungsarten von Indexen

- Zielsetzung: Effiziente Unterstützung von Selektionen
- Beispiel: Primärindex
  - Die Tupel sind eindeutig durch einen Primärschlüssel oder einen Schlüsselkandidaten bestimmt.
  - Verwendung eines Clusterindex möglich, d.h. die Daten sind gemäß dem Schlüssel geordnet gespeichert minimale Such- und Latenzzeit auf Platten!
- Beispiel: Sekundärindex
  - Indexe dürfen auch über anderen Attributen angelegt werden.
  - In diesem Fall können in den Attributwerten auch Duplikate auftreten.
- Speicherhierarchie impliziert wichtige Nebenbedingungen:
  - Vorhersagbarer Suchaufwand: AVL-Binärbäume sinnlos balancierte Mehrwegbäume
  - Möglichst wenig I/O : Ausnutzen der Blockstruktur B-Baum-Familie als Standard

Blockweise Speicherung von Daten

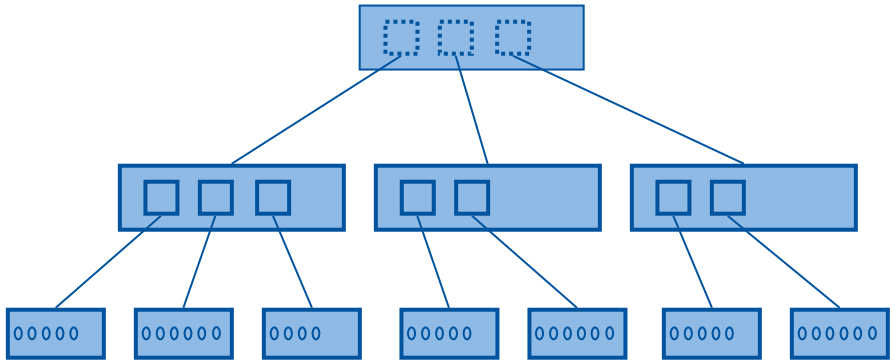
- Block- oder seitenweise Speicherung
  - Speicherung vieler Datensätze auf ein- und derselben Seite
  - Für effiziente Suche: Speichere ähnliche Werte auf derselben Seite
  - Bei Überlauf einer Seite: Aufspaltung auf mehrere Seiten

- Hierarchische Organisation
  - Übergeordnete Knoten zur Erschließung der Datenblöcke (Directory)



Mehrstufige Hierarchien (Bäume)

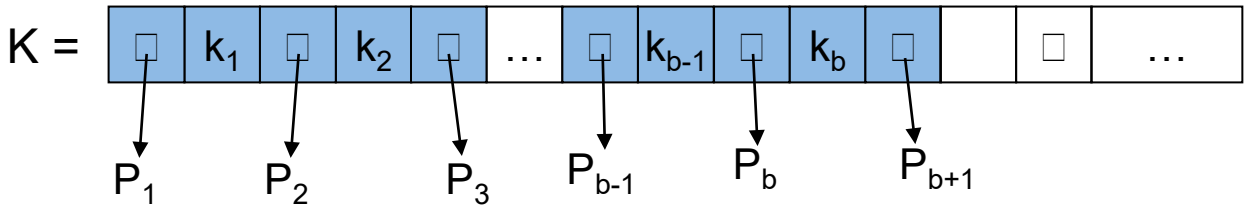
- Rekursive Aufspaltung liefert Baumstruktur



- Eigenschaften der Datenstruktur
  - Blattknoten enthalten Datensätze
  - Innere Knoten enthalten Knotenbeschreibungen und Zeiger
  - Alle Blätter haben dieselbe Entfernung von der Wurzel
  - Jeder Knoten hat höchstens M viele Einträge
  - Jeder Knoten (außer Wurzel) hat mindestens m ≥ M/2 Einträge

Mehrweg-Bäume

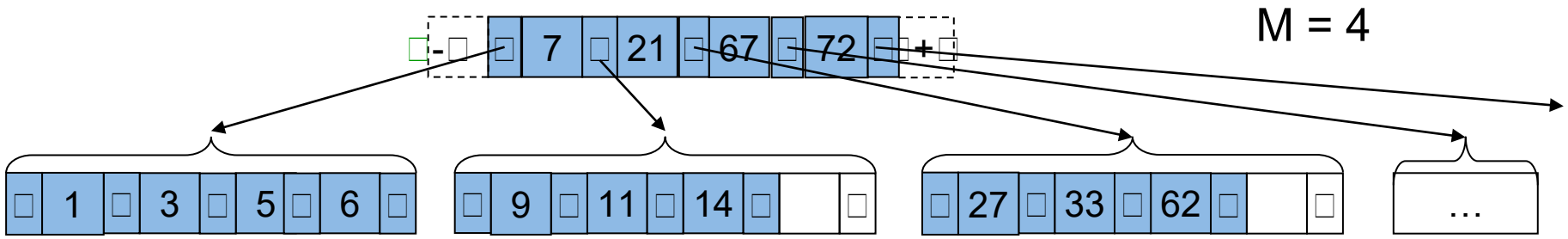
- Charakterisierung
  - Knoten haben bis zu M+1 viele Nachfolger
  - Blockorientierte Speicherung der Bäume: Speichere Knoten auf Plattenseiten
  - Damit ergibt sich M aus Seitengröße und Größe der Datensätze + Zeiger



- Knoten K eines (M+1)-Wege Suchbaums besteht aus:
  - Verzweigungsgrad b+1 = Grad(K) □ M+1
  - Datensätze mit Schlüsseln k_i (1 □ i □ b)
  - Zeiger P_i auf die Unterbäume (1 □ i □ b+1)

Mehrweg-Suchbäume

- Suchbaumeigenschaft für Mehrwegebäume
  - Die Schlüssel $k_1, k_2, \ldots, k_b$ in einem Knoten K sind geordnet, d.h. für i = 1, ..., b–1 gilt:
    $$k_i \square k_{i+1}$$
  - Für alle Schlüssel k‘ im Teilbaum, der zwischen den Schlüsseln k_{p-1} und k_p liegt, gilt (setze $k_0 := -\square$ und $k_b := +\square$):
    $$k_{p-1} < k' \square k_p$$
- Beispiel



B-Baum

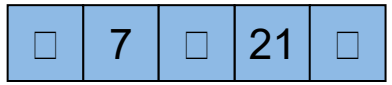
- Definition:
  Ein B-Baum ist ein (M+1)-Wege Suchbaum (für eine gerade Zahl M)

- Für einen nicht leeren B-Baum gilt:
  1. Jeder Knoten enthält höchstens M Schlüssel
  2. Die Wurzel enthält mindestens einen Schlüssel
  3. Jeder Knoten außer der Wurzel enthält mindestens m = M/2 Schlüssel
  4. Ein innerer Knoten mit b Schlüsseln hat genau b+1 Kinder
  5. Alle Blätter befinden sich auf demselben Level (Balanciertheit)

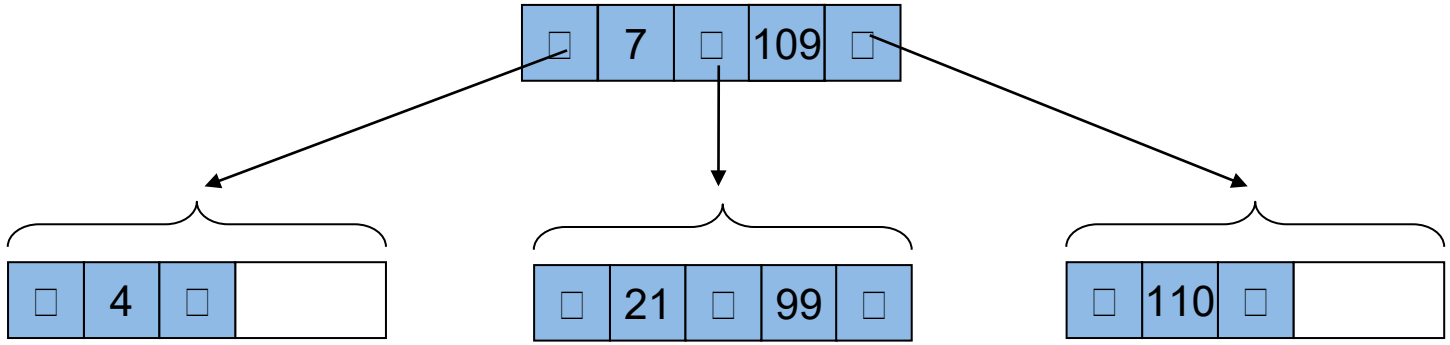
- Bedeutung des „B“:
  - **B**alanced Tree, **B**locked Tree (technische Beschreibung)
  - **B**ushy Tree, **B**road Tree (Hinweis auf hohen Verzweigungsgrad)
  - Prof. Dr. Rudolf **B**ayer (mit Ed McCreight Erfinder der B-Bäume)
  - The **B**oeing Company (Bayer arbeitete in deren Forschungslabor)
  - **B**arbara (Vorname von Bayers Ehefrau)
  - **B**anyan Tree (australischer Baum, wächst durch Wurzelteilung)
  - **B**inary Tree (falsch, da Mehrwegebaum; richtig, da binäre Suche)

Beispiele für B-Bäume

- Beispiele für B-Bäume mit M = 2 (d.h. maximal 3 Nachfolger)
  - Bsp. Höhe 1



  - Bsp. Höhe 2



Suchen im B-Baum

- Suche nach einem Datensatz
  - Beginne in der Wurzel und suche binär auf dem jeweiligen Knoten
  - Falls nicht gefunden: Suche im entsprechenden Teilbaum rekursiv weiter

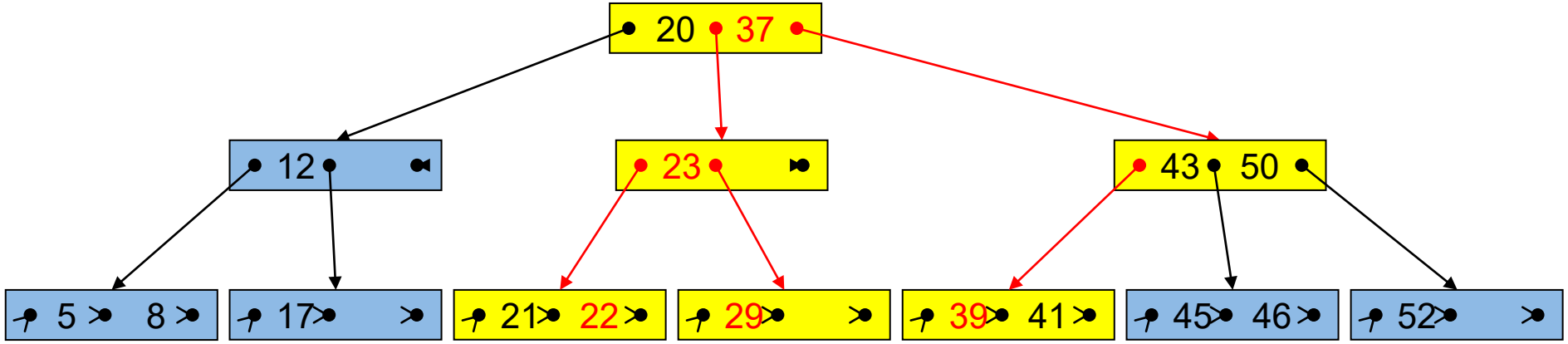
- Komplexität der Suche
  - In einem B-Baum der Höhe h werden maximal h viele Knoten besucht
  - Die Höhe eines B-Baums mit N Objekten ist maximal $h = \log_m N$ (s.später)
  - Die binäre Suche auf einem Knoten benötigt maximal $\log_2 M$ Vergleiche
  - Anzahl der Vergleiche insgesamt: höchstens $\log_2 M \cdot \log_m N \square \log_2 N$
  - Anzahl der Plattenzugriffe (jeweils ca. 10ms): höchstens $\log_m N$

- Beispiel
  N = 1 Mio, M = 100 gibt 20 Vergleiche, 3 Plattenzugriffe (Wurzel im Cache)

Bereichsanfrage im B-Baum

- Suche Objekte, die in einen Bereich (min, max) fallen
  - Suche rekursiv jeweils binär den kleinsten Eintrag e ≥ min
  - Gehe von e aus mit Inorder-Durchlauf bis zum größten Eintrag e‘ ≤ max
  - Sei r die Anzahl der dabei gefundenen Elemente
  - Anzahl Vergleiche: $O(r + \log_2 N)$
  - Anzahl Plattenzugriffe: $O(r/m + \log_m N)$

- Beispiel: (22, 40)

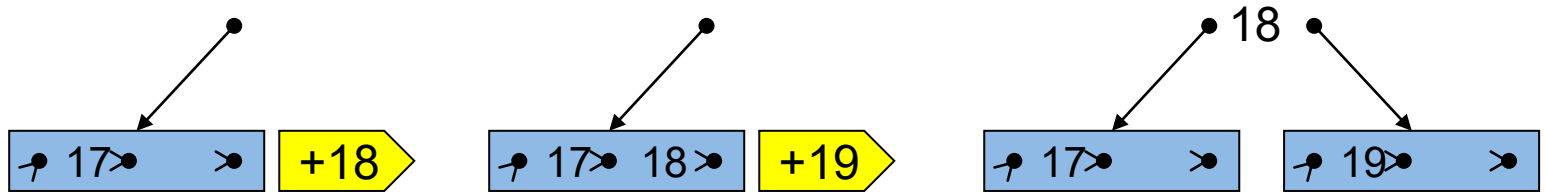


Einfügen im B-Baum

- Grundidee
  - Neue Objekte werden nur in Blättern eingefügt
  - Bei Überlauf eines Blatts wird ein neues Blatt erzeugt;
    die Einträge werden zwischen den beiden Nachbarknoten verteilt
  - Der Baum wächst nicht in die Tiefe, sondern in die Höhe

- Algorithmus: Einfügen eines neuen Objekts
  - Suche das Blatt, in welches das neue Objekt gehört
  - Füge das Objekt sortiert in das Blatt ein
  - Wenn hierdurch der Blattknoten überläuft, spalte ihn auf

- Beispiel

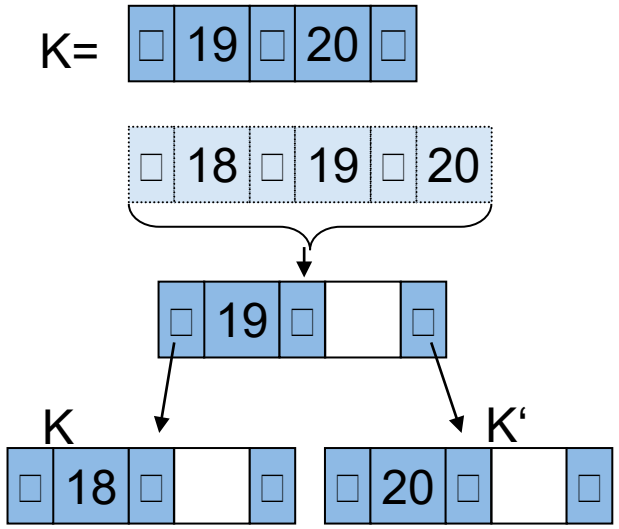


Einfügen im B-Baum: Split

- Überlauf eines Knotens
  - Knoten K kann M+1 Objekte (o_1, o_2, …, o_{M+1}) nicht fassen
  - Erzeuge einen Nachbarknoten K‘
  - Verteile die M+1 Objekte auf die beiden Knoten
    K = (o_1, o_2, …, o_m) und K‘ = (o_{m+2}, …, o_{M+1})
  - Das mittlere Objekt o_{m+1} wird dem Vorgängerknoten hinzugefügt

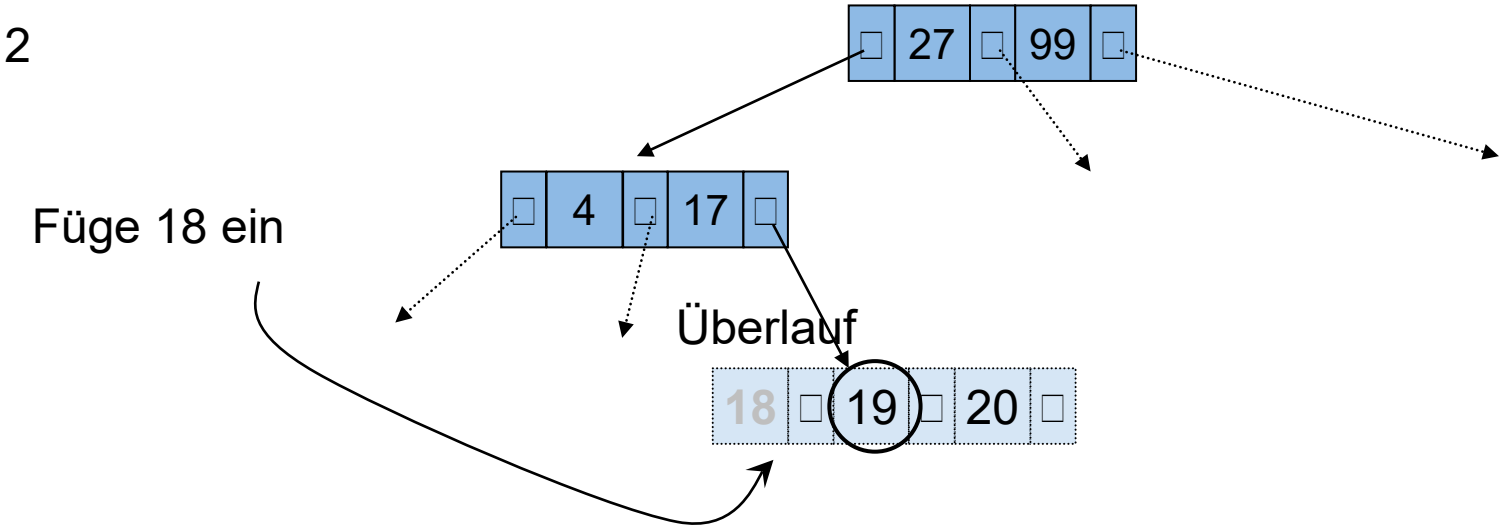
- Falls Vorgänger nicht existiert
  - Knoten war die Wurzel: Schaffe neue Wurzel
  - Die Höhe wächst um Eins

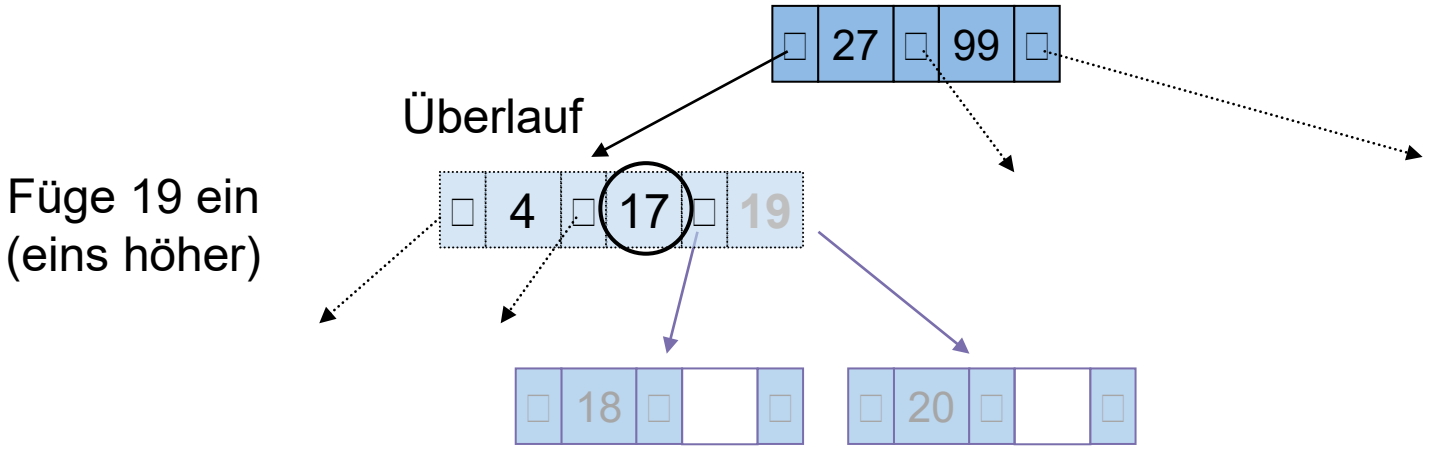
- Falls Vorgänger überläuft
  - Wende denselben Split-Algorithmus an
  - Split kann rekursiv bis zur Wurzel laufen
  - Komplexität des Einfügens: $O(\log_m N)$



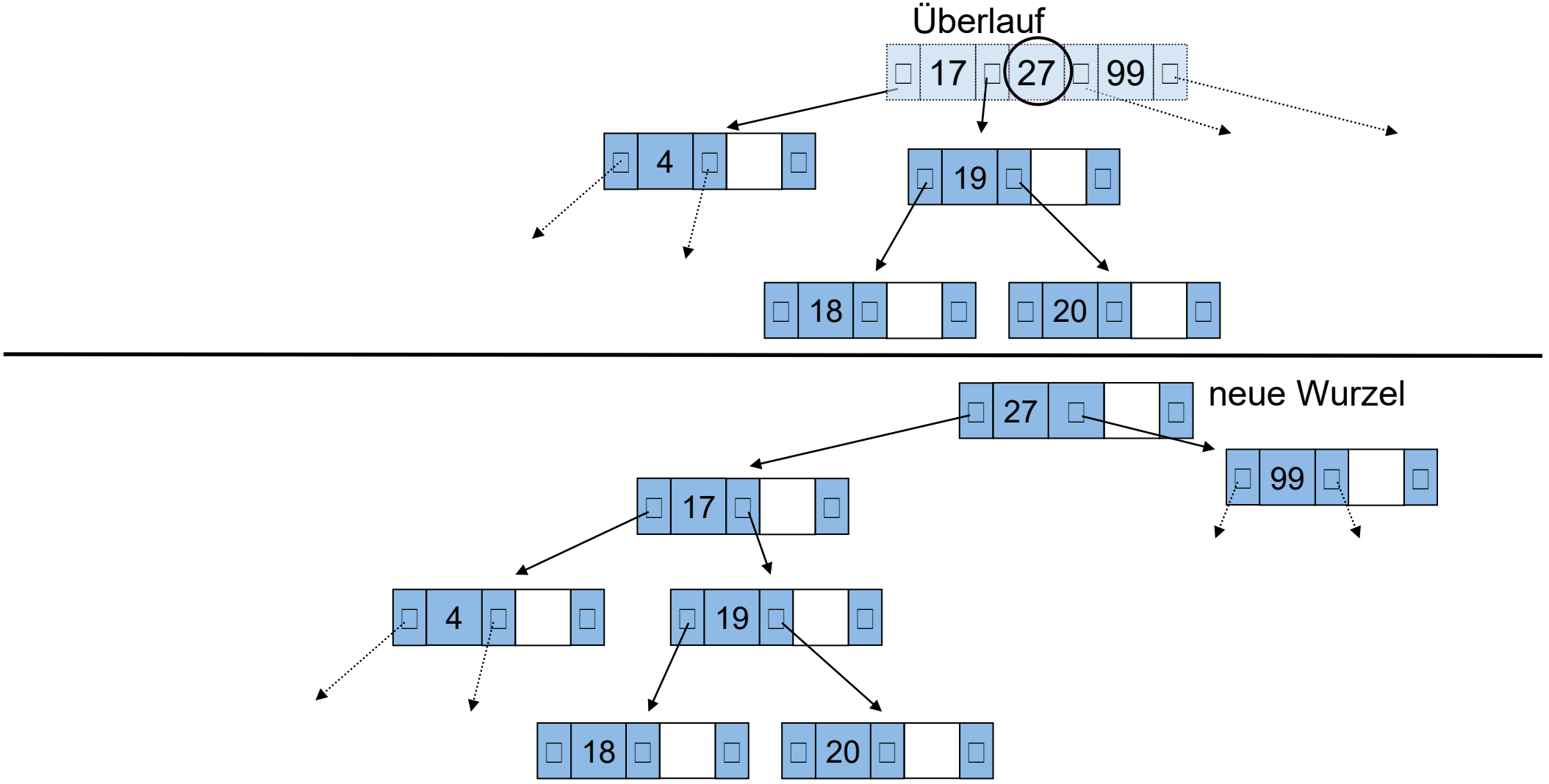
Beispiel für Einfügen im B-Baum

- Beispiel: M = 2





Beispiel (2)



Löschen im B-Baum

- Suche den Knoten K, der den zu löschenden Schlüssel o enthält

- Falls K ein Blatt ist: Lösche den Schlüssel o aus dem Blatt
  - Es ist möglich, dass K nun weniger als m = M/2 Schlüssel beinhaltet
  - Reorganisation unter Einbeziehung der Nachbarknoten

- Falls K ein innerer Knoten ist
  - Suche den größten Schlüssel o‘ im Teilbaum links von Schlüssel o
  - Ersetze o im Knoten K durch o‘
  - Lösche o‘ aus seinem ursprünglichen Knoten (das ist ein Blatt)

- Falls K die Wurzel ist
  - Die Wurzel hat keine Nachbarn und darf weniger als m = M/2 Schlüssel beinhalten

Löschen im B-Baum (Ausgleich)

Entferne Schlüssel o_i aus dem Knoten K = (o_1, …, o_b) eines B-Baums:

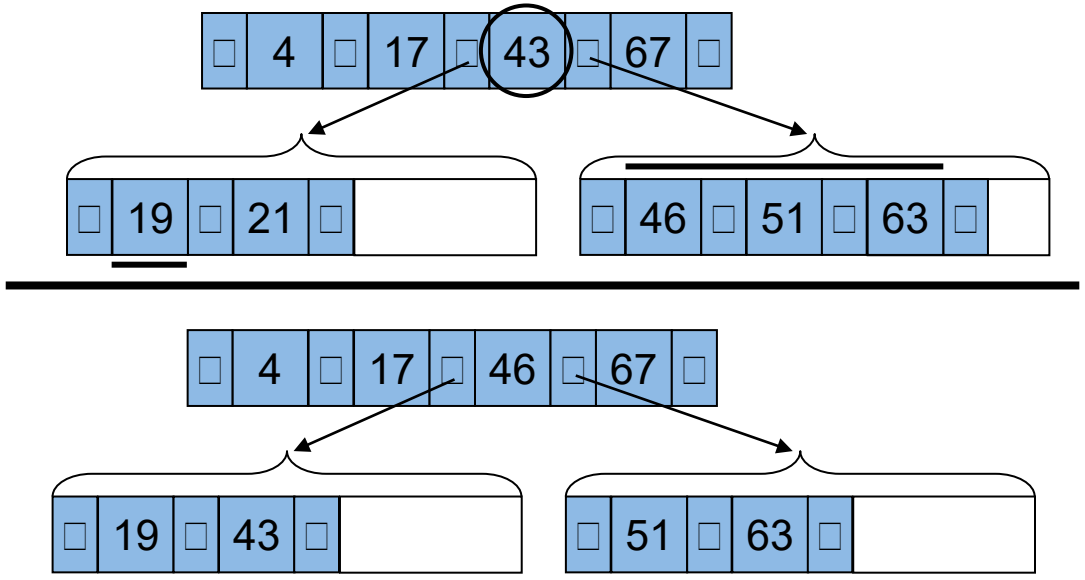
- Falls es einen Nachbarknoten K‘ = (o‘$_1$,…,o‘$_n$) mit **mehr** als *m* = M/2 Schlüsseln gibt, kann ein Ausgleich durchgeführt werden:
  - O.B.d.A. sei K‘ rechts von K, und p der Trennschlüssel im Vorgänger
  - Verteile die Schlüssel o_1 … o_b, p, o‘$_1$ … o‘$_n$ auf die Knoten K und K‘, und ersetze den Schlüssel p im Vorgänger durch den mittleren Schlüssel
  - K und K‘ haben nun jeweils mindestens *m* = M/2 Schlüssel

Beispiel:

B-Baum mit M = 4

Lösche Schlüssel 21

Ausgleich(19 ,43 , 46 , 51 , 63)



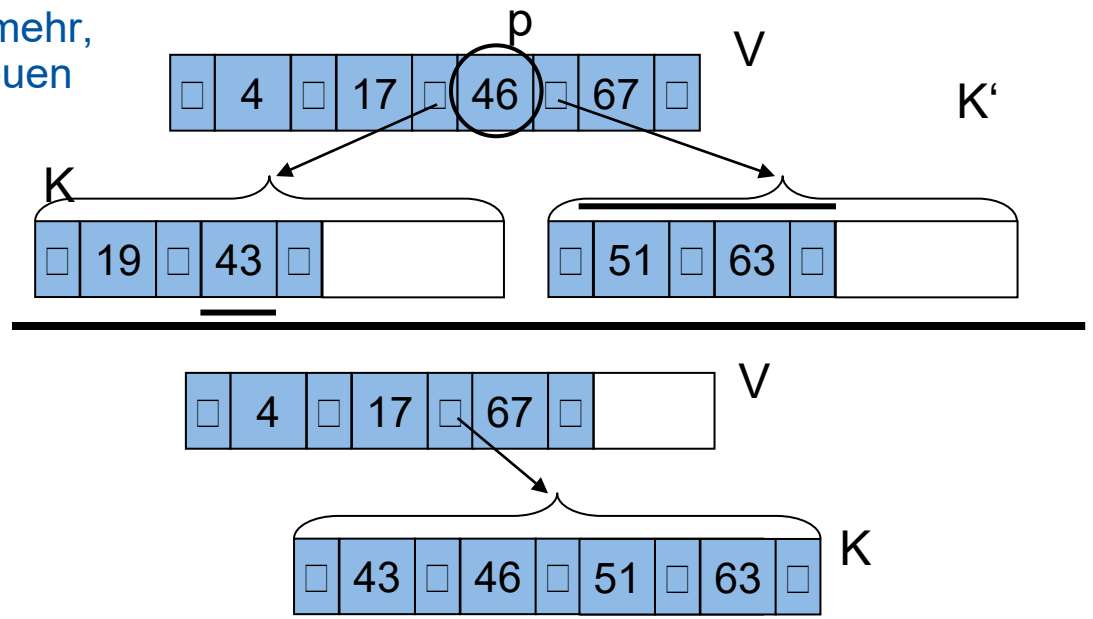
Löschen im B-Baum (Verschmelzen)

Falls es keinen Nachbarknoten mit mehr als m = *M*/2 Elementen gibt, so existiert mindestens ein Nachbarknoten K‘ = (o‘$_1$, …, o‘$_m$) mit **genau** *m* Schlüsseln:

- O.B.d.A. sei K‘ rechts von K, und *p* der Trennschlüssel im Vorgänger V
- Verschmelze die Knoten K‘ und K zu K, füge *p* in K hinzu und lösche K‘
- Entferne *p* sowie den Verweis auf K‘ aus dem Vorgänger V
- Ggf. rekursiv bis zur Wurzel (enthält diese danach keine Schlüssel mehr, so wird das einzige Kind zur neuen Wurzel)

Beispiel:
B-Baum mit M = 4
Lösche Schlüssel 19

Verschmelze (43 , 46, 51 , 63)
Entferne (p=46)



Höhenabschätzung für B-Bäume

- Schlüsselanzahl N in einem Baum der Höhe h
  - Minimal:

$$\begin{aligned} N &\geq 1 + 2m + 2(m+1)\cdot m + 2(m+1)^2 \cdot m + \ldots \\ &= 1 + 2m \cdot \sum_{i=0}^{h-2} (m+1)^i = 2(m+1)^{h-1} - 1 \end{aligned}$$

  - Maximal:

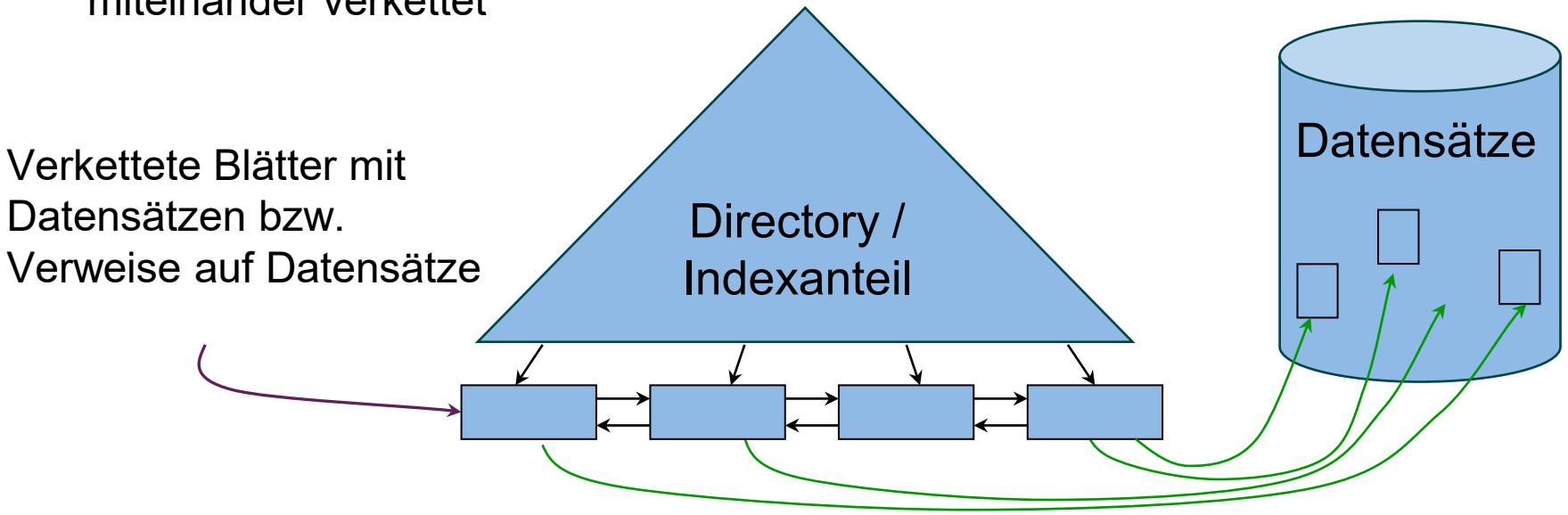
$$\begin{aligned} N &\leq M + (M+1)\cdot M + (M+1)^2 \cdot M + \ldots \\ &= M \cdot \sum_{i=0}^{h-1} (M+1)^i = (M+1)^h - 1 \end{aligned}$$

$$\log_{M+1}(N+1) \leq h \leq \log_{m+1}\left(\frac{N+1}{2}\right) + 1$$

- Auflösen nach h ergibt die Höhenabschätzung:
- Betrachtung der Schranken
  - Die Höhe eines B-Baumes ist durch den Logarithmus zur Basis der maximalen bzw. minimalen Anzahl Nachfolger eines Knotens beschränkt

Wichtige Variante B⁺-Baum

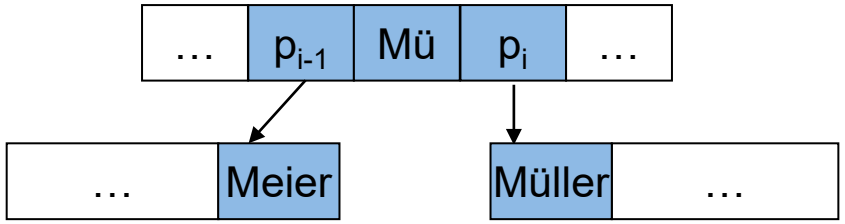
- Ein B⁺-Baum ist eine B-Baum-Variante mit zwei Knotentypen
  - Blätter enthalten Schlüssel mit Datensätzen oder Schlüssel mit Verweisen auf Datensätze
  - Innere Knoten enthalten keine Datensätze, nur Trennschlüssel
  - Als Trennschlüssel (Separatoren, Wegweiser) nutzt man z.B. die Schlüssel selbst oder geeignete Präfixe (bei Strings)
  - Für ein effizientes Durchlaufen großer Bereiche der Daten sind die Blätter miteinander verkettet



Vergleich B⁺-Baum und B-Baum

- Da in den inneren Knoten nur Schlüssel ohne Daten gespeichert werden, haben auf einer B⁺-Baum-Seite mehr Einträge Platz

- Schlüssel dienen nur als Wegweiser und können deswegen oft verkürzt werden:
  - Verwende als Trennschlüssel k_i (Wegweiser) z.B. das kürzeste Präfix des ersten Schlüssels im rechten Teilbaum p_i von k_i, das größer ist als der größte Schlüssel im linken Teilbaum p_{i-1} von k_i



- Dadurch B+-Baum in der Regel breiter und weniger hoch als B-Baum
- In der Praxis werden wegen dieser Vorteile überwiegend nur noch Varianten von B⁺-Bäumen eingesetzt.

5. Relationale Anfragebearbeitung

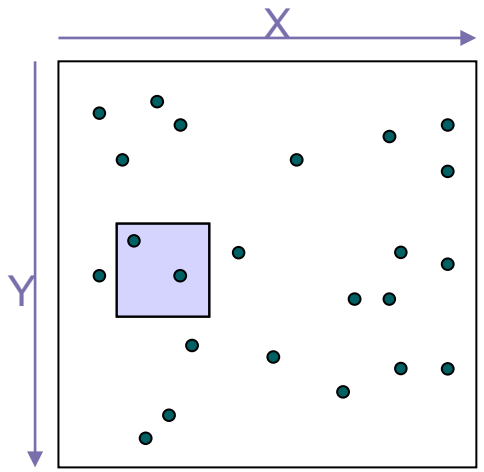
1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - **Indexstrukturen für mehrdimensionale Daten**

Mehrdimensionale Daten

- Problemstellung:
  - Gesucht wird anhand mehrdimensionaler Schlüssel $K = (a_1, \ldots, a_n)$ basierend auf verschiedenen Attributen $A_1, \ldots, A_n$
  - Gesuchte Attribute $A_1, \ldots, A_n$ sind gleichwertig
- Beispiel:

Matrikel-nummer	Nebenfach	Semester
171283	BWL	9
184238	Medizin	7
191373	Elektrotechnik	5
…	…	…

```
SELECT *
FROM Studenten
WHERE Nebenfach = „BWL“
      AND Semester >= 7
      AND   Semester <= 9
```

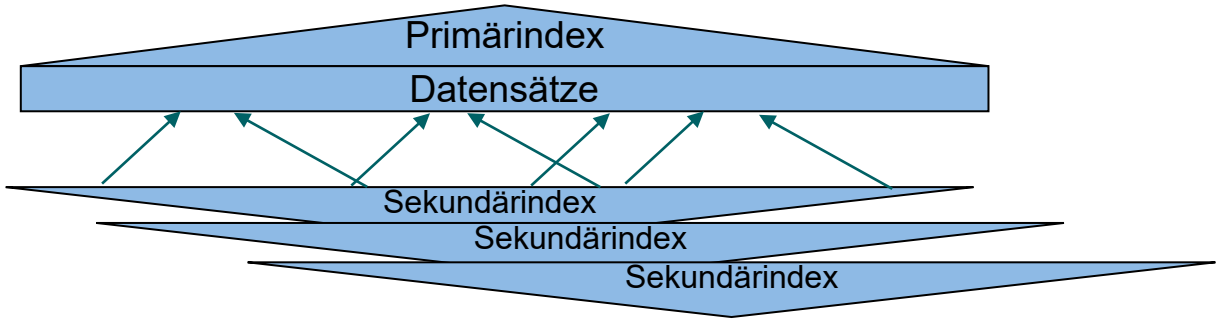



```
SELECT *
FROM Geometry
WHERE 2.5 <= x AND x <= 4.1
      AND 3.8 <= y AND y <= 6.2
```

Invertierte Listen

Ziel: Unterstützung von Anfragen über mehrere Attribute

- Multiattributssuche anhand *invertierter Listen*
  - Für jedes Attribut gibt es einen (Sekundär-) Index
  - Suche in allen Indexen unabhängig von den anderen
  - Kombiniere Ergebnis über Durchschnittsbildung
- Indexgefüge
  - *Primärindex:* Index über den Primärschlüssel
  - *Sekundärindex:* Index über ein Attribut, das kein Primärschlüssel ist
    - Im Gegensatz zu einem Primärindex beeinflusst der Sekundärindex den Ort der Speicherung eines Datensatzes nicht. Es werden nur Verweise gespeichert.



Invertierte Listen (2)

Konzept der invertierten Listen:

- Für anfragerelevante Attribute werden Sekundärindexe ***(invertierte Listen)*** angelegt.
- Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

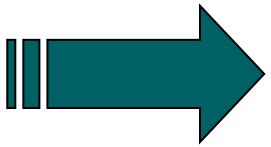
Multiattributsuche für invertierte Listen:

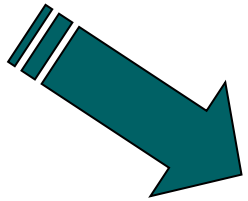
- Eine Anfrage spezifiziere die Attribute A_1, ... ,A_m:
  - *m Anfragen über m Indexstrukturen*
- Ergebnis:
  m Listen mit Verweisen auf die entsprechenden Antwortkandidaten in der Datei.
- *Mengentheoretische Verknüpfung* (z.B. Durchschnitt) der m Listen gemäß der Anfrage

Invertierte Listen (Beispiel)

Primärindex (über Name)

Speicher-adresse	Name	Stadt	Alter
1	*Adams*	Athen	30
2	*Blake*	Paris	30
3	*Clark*	London	50
4	*Hart*	Chicago	40
5	*James*	Athen	30
6	*Jones*	Paris	40
7	*Parker*	New York	40
8	*Smith*	London	30





invertierte Listen:

Stadt	Speicher-Adresse
Athen	1, 5
Chicago	4
London	3, 8
New York	7
Paris	2, 6

Alter	Speicher-Adresse
30	1, 2, 5, 8
40	4, 6, 7
50	3

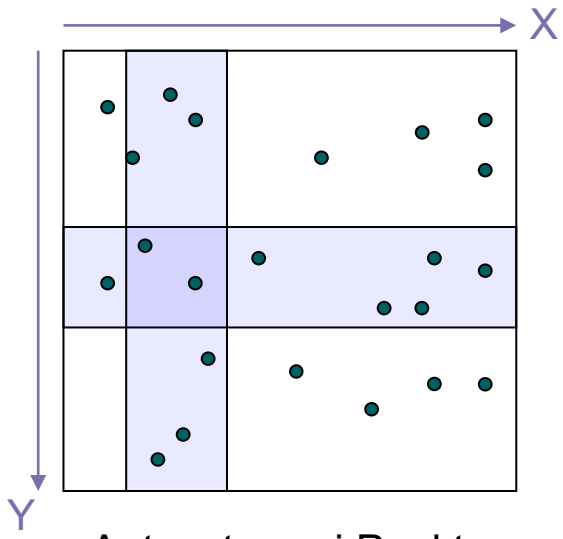
```
SELECT *
FROM ….
WHERE Stadt = „Athen“
    AND Alter = 30
```

„Athen“ „30“
{1,5} ∩ {1,2,5,8} = {1,5}

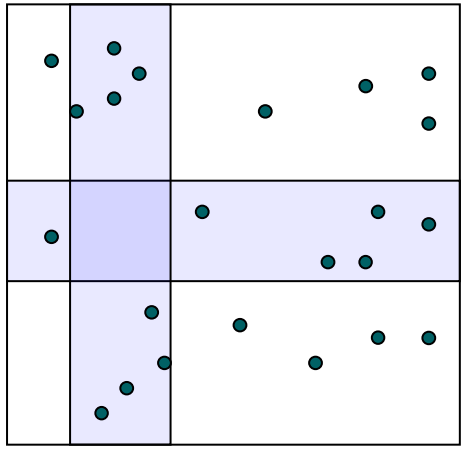
Invertierte Listen: Punktdaten

Bei 2-dimensionalen Punktdaten:

- Speicherung der X- und Y-Werte in jeweils einem Sekundärindex
- bei Suchen von Punkten in Bereichen (Rechtecken):
  - Bilde Punktmenge, deren X-Koordinaten im Anfragebereich liegen
  - Bilde Punktmenge, deren Y-Koordinaten im Anfragebereich liegen
  - Bilde die Schnittmenge beider Mengen



Antwort: zwei Punkte



„worst-case“

Invertierte Listen: Eigenschaften

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Die Suche dauert umso länger, je mehr Attribute spezifiziert sind.
- *Ursache für beide Beobachtungen:*
  Die Attributwerte eines Datensatzes sind nicht in einer Struktur miteinander verbunden.
- Invertierte Listen sind einigermaßen effizient, wenn die Antwortlisten sehr klein sind.
- Invertierte Listen haben hohe Kosten für Update-Operationen.
- Sekundärindexe beeinflussen die physische Speicherung der Datensätze nicht.
  - Ordnungserhaltung über den Sekundärschlüssel nicht möglich.
  - schlechtes Leistungsverhalten von invertierten Listen.

Bitmap Index

- Bitmap-Indizes sind ein spezieller Indextyp, der für effiziente Abfragen mit mehreren Schlüsseln konzipiert ist.
- Sehr effektiv bei Attributen, die eine relativ kleine Anzahl *unterschiedlicher Werte* annehmen
  - z.B. Geschlecht, Land, Staat, ...
  - Z.B. Einkommensstufe (Einkommen unterteilt in eine kleine Anzahl von Stufen wie 0-9999, 10000-19999, 20000-50000, 50000-unendlich)
- Eine Bitmap ist einfach ein Bit-Array
  - Jedem Geschlecht wird eine Bitmap zugeordnet, wobei jedes Bit angibt, ob der entsprechende Datensatz dieses Geschlecht hat oder nicht.

Bitmap Index (2)

- In seiner einfachsten Form hat ein Bitmap-Index für ein Attribut eine Bitmap für jeden Wert des Attributs
  - Eine Bitmap hat so viele Bits wie die Relation Tupel
  - In einer Bitmap für den Wert v ist das Bit für einen Datensatz 1, wenn der Datensatz den Wert v für das Attribut hat, und sonst 0

Tupel	Name	Geschlecht	Adresse	Einkommenstufe
0	Klaus	m	Aachen	L1
1	Diana	w	Köln	L2
2	Maria	w	Bonn	L1
3	Peter	m	Köln	L4
4	Johannes	w	Aachen	L3

Bitmaps für Geschlecht

m	1 0 0 1 0
f	0 1 1 0 1

Bitmaps für Einkommenstufe

L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

Bitmap Index

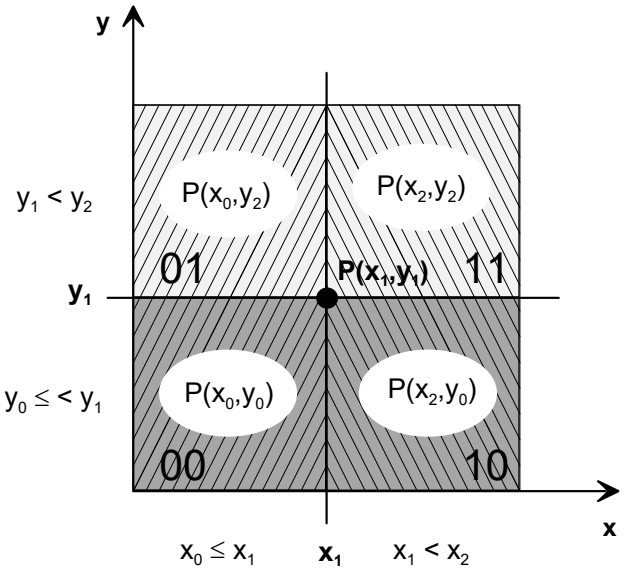
- Bitmap-Indizes sind nützlich für Abfragen über mehrere Attribute
  - nicht besonders nützlich für Abfragen über ein einzelnes Attribut
- Abfragen werden mit Bitmap-Operationen beantwortet
  - Schnittmenge (und)
  - Vereinigung (oder)
  - Komplement (nicht)
- Jede Operation nimmt zwei Bitmaps der gleichen Größe und wendet die Operation auf die entsprechenden Bits an, um die Ergebnis-Bitmap zu erhalten
  - Z.B. 100110 AND 110011 = 100010
    100110 ODER 110011 = 110111
    NICHT 100110 = 011001
  - Männer mit Einkommensstufe L1:
  - Und-Verknüpfung der Bitmap Männer mit der Bitmap Einkommensstufe L1
  - 10010 UND 10100 = 10000
- Kann dann die gewünschten Tupel abrufen.
- Das Zählen der Anzahl der übereinstimmenden Tupel ist noch schneller

Bitmap Index

- Bitmap-Indizes sind im Allgemeinen sehr klein im Vergleich zur Größe der Beziehung
  - Wenn z.B. ein Datensatz 100 Bytes groß ist, beträgt der Platz für eine einzelne Bitmap 1/800 des von der Relation belegten Platzes.
  - Wenn die Anzahl der unterschiedlichen Attributwerte 8 beträgt, ist die Bitmap nur 1% der Größe der Beziehung.
- Das Löschen muss richtig gehandhabt werden
  - Existenz-Bitmap, um zu vermerken, ob ein gültiger Datensatz an einer Datensatzposition vorhanden ist
  - Erforderlich für die Komplementierung
    - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Sollte Bitmaps für alle Werte behalten, auch für Nullwerte
  - Um die SQL-Null-Semantik für NOT(A=v) korrekt zu behandeln:
  - obiges Ergebnis mit (NOT bitmap-A-Null) schneiden

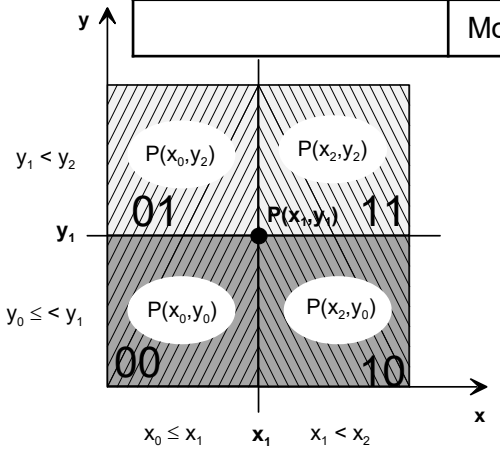
Quadtree (2D)

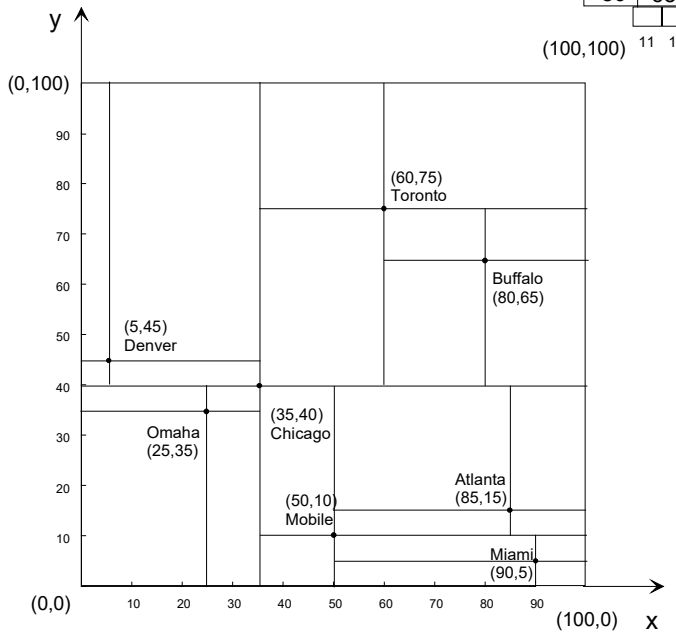
- nicht-balancierte Datenstruktur für mehrdimensionale Punkt-Objekte (hier: 2-dimensionale)
- Ansatz: Jeder Knoten hat sowohl für die x- als auch für die y-Koordinate je zwei Kindknoten: x (<=,>) und y (<=,>) (je Knoten also vier Kindknoten „*Quad*“-Tree)
- Aufbau des Baums:
  - erster Punkt bildet Wurzel
  - bei jedem weiteren Punkt:
    - Bestimmung des Quadranten, in dem Punkt liegt
    - Abstieg in entsprechenden Kindknoten
    - falls kein Kindknoten für diesen Quadranten existiert: Hinzufügen eines Kindknotens
- hier: Hauptspeicherstruktur (Verzweigungsgrad = 2^d für d Dimensionen)
- später: Quadrant = Bucket zur Verwendung als Sekundärspeicherstruktur



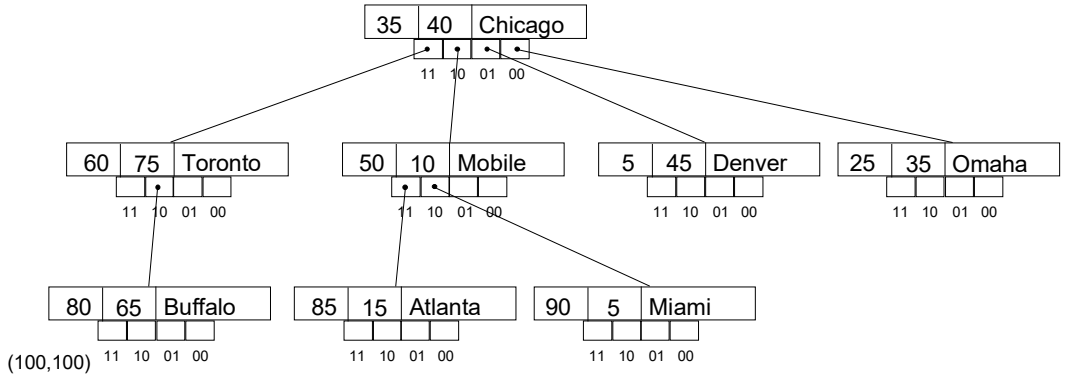
Quadtree (Beispiel)

einzufügen:	Vergleichs-knoten	$D_{binär}$
Chicago (35,40)	-	
Denver (5,45)	Chicago (35,40)	01
Mobile (50,10)	Chicago (35,40)	10
Toronto (60,75)	Chicago (35,40)	11
Buffalo (80,65)	Chicago (35,40)	11
	Toronto (60,75)	10
Miami (90,5)	Chicago (35,40)	10
	Mobile (50,10)	10
Omaha (25,35)	Chicago (35,40)	00
Atlanta (85,15)	Chicago (35,40)	10
	Mobile (50,10)	11





Nach allen Einfügungen:



R-Baum

(Guttman A.: 'R-trees: A Dynamic Index Structure for Spatial Searching', Proc. ACM SIGMOD Int. Conf. on Management of Data,1984, pp. 47-57.)

R (Rectangle)-Baum: höhenbalancierter Baum zur Speicherung von Punkt- und Rechteckdaten

Idee:

- basiert auf der Technik überlappender Seitenregionen
- Approximation der Objekte durch minimale umgebende Rechtecke
  (Abk. MUR, engl. MBR "minimum bounding rectangle")
- verallgemeinert die Idee des B^+-Baums auf den mehrdimensionalen Raum

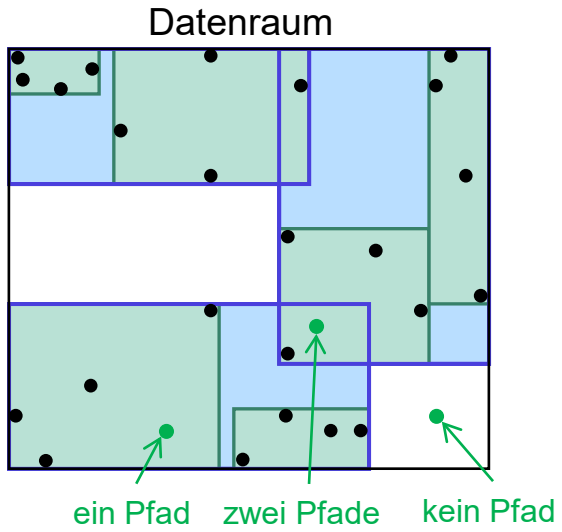
Aufbau einer Seite:

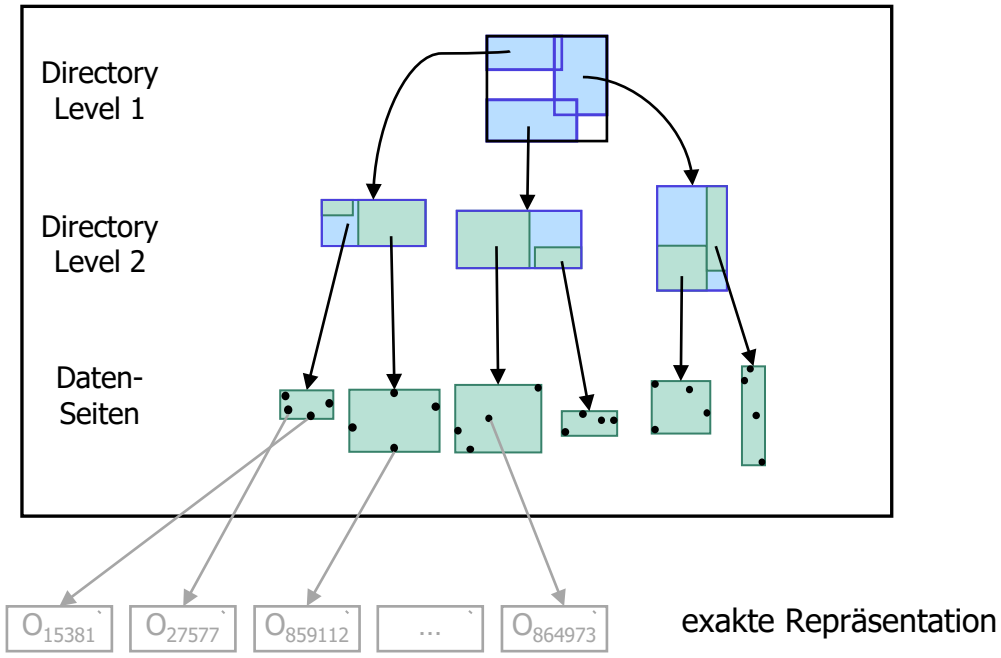
- Seite besteht aus mehreren Einträgen
- Einträge in Directory-Seiten bestehen aus MURs und Verweisen auf andere Seiten
- Einträge in Datenseiten bestehen aus MURs und Verweisen auf die exakte Objekt-Repräsentation, bzw. einfach aus Punkten

R-Baum (2)

“Partitionierung“ des Datenraums:

- jedes Rechteck in einer Directoryseite umfasst als MUR alle Rechtecke in allen Directory- oder Datenseiten, die im zugehörigen Teilbaum liegen
- nicht disjunkt: die Rechtecke einer Seite können sich überlappen
- „Partitionierung“ des Datenraumes der Directoryseite muss nicht vollständig sein, d.h. es existiert „leerer“ Raum





R-Baum: Eigenschaften

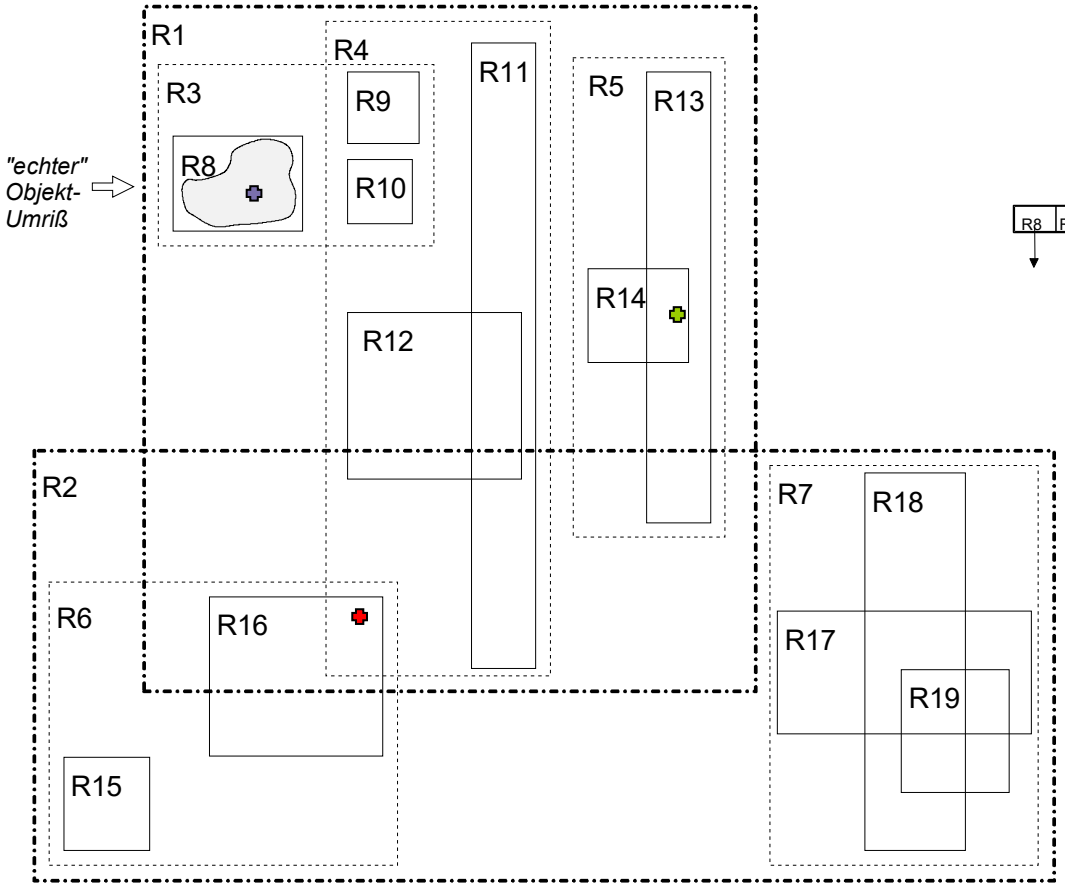
Parameter:

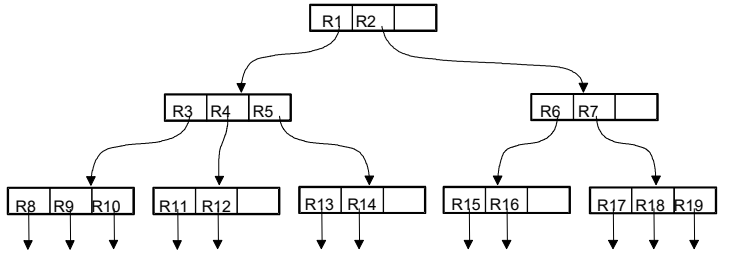
- M : maximale Anzahl von Einträgen pro Knoten (abhängig von Blockgröße)
- $m \leq M/2$: Mindestbelegung pro Knoten (z.B. $m = 40\ \% \cdot M$)

Eigenschaften:

- Anzahl der Index-Einträge pro Blatt-Knoten zwischen m und M
- Anzahl der Kindknoten von Nichtblatt-Knoten (Directory-Knoten) zwischen m und M
- in inneren Knoten ist das kleinste Rechteck gespeichert, welches Rechtecke der Kindknoten umfasst (MUR: **M**inimal **U**mgebendes **R**echteck, engl. MBR: **M**inimum **B**ounding **R**ectangle)
- in Blatt-Knoten (Datenknoten) ist Verweis auf Objekt und sein kleinstes umschließendes Rechteck gespeichert
- höhenbalanciert (Blattknoten auf derselben Höhe)
- Zerlegung des Datenraums nicht disjunkt (also überlappende Regionen möglich)
- Höhe des Baums ≤ $\lceil \log_m N \rceil - 1$ (bei N gespeicherten Objekten)

R-Baum: Punktanfrage



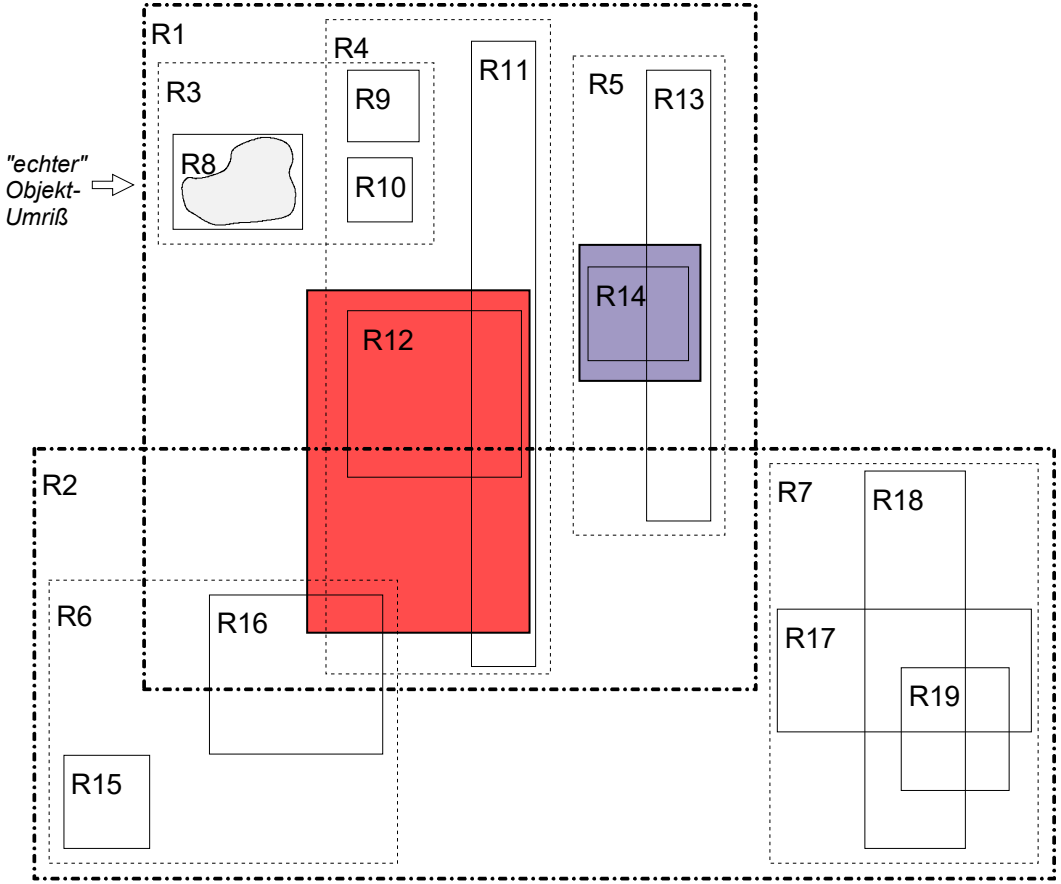


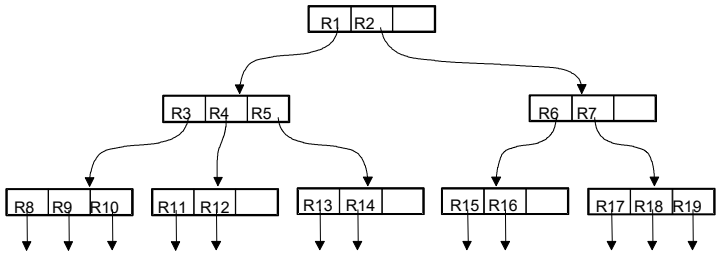
Eingabe: *Punkt p*

1. Starten bei Wurzel
2. Tiefensuche im R-Baum
3. Untersuche jeweils Kindknoten von Nichtblattknoten, deren Rechteck den Punkt p enthält
4. Überprüfe in Blattknoten, ob eines der Rechtecke den Punkt p enthält

- untersuche R1 □ R3 □ **R8** □ R9 □ R10 □ R4 □ R5 □ R2
- untersuche R1 □ R3 □ R4 □ R5 □ **R13** □ **R14** □ R2
- untersuche R1 □ R3 □ R4 □ R11 □ R12 □ R5 □ R2 □ R6 □ R15 □ **R16** □ R7

R-Baum: Rechteckanfrage (Schnitt)





Eingabe: Rechteck r

1. Starten bei Wurzel
2. Tiefensuche im R-Baum
3. Untersuche jeweils Kindknoten von Nichtblattknoten, deren Rechteck das Rechteck R schneidet
4. Überprüfe in Blattknoten, ob eines der Rechtecke das Anfragerechteck schneidet

R1 □ R3 □ R4 □ R5 □ **R13** □ **R14** □ R2

R1 □ R3 □ R4 □ **R11** □ **R12** □ R5 □ R2 □ R6 □ R15 □ **R16** □ R7

R-Baum: Einfügen

- ähnlich wie im B^+-Baum
- Einfügungen erfolgen stets in den Blattknoten
- im Gegensatz zum B-Baum kommen hier i. a. mehrere Blattknoten in Frage (Überlappungen von minimal umgebenden Rechtecken)
- Wahl des Blattknotens/Teilbaumes mit minimaler Vergrößerung der MURs
- Durch Einfügen eines neuen Elements kann ein Knoten überlaufen:
  Verschiedene Heuristiken:
  - Quadratischer Split
    - Laufzeitkomplexität ist quadratisch in der Anzahl der Rechtecke
    - Verteile Einträge auf zwei Knoten, so dass die Flächenvergrößerung des minimal umgebenden Rechteck am geringsten ist
  - Linearer Split
    - Laufzeitkomplexität ist linear in der Anzahl der Rechtecke
    - Basierend auf größter normalisierter Separierung in den Dimensionen

R-Baum: Löschen

- Beginnend bei der Wurzel, durchsuche alle Teilbäume, in denen der zu löschende Eintrag sein könnte, bis Eintrag gefunden ist.
- Entferne Objekt aus Plattenblock.
- Passe minimal umgebende Rechtecke auf dem Pfad zurück zur Wurzel an (falls nötig).
- Zwei Strategien, falls Unterlauf in Blattknoten auftritt:
  - Unterlauf behandeln: Verschmelze Nachbarknoten, propagiere Entfernung nach oben.
  - Unterlauf ignorieren: Beliebte Vorgehensweise; falls mehr Einfügungen als Entfernungen auftreten, wird die Seite vermutlich schon bald wieder weiter belegt. Verschmelzung und erneuter Split kann eingespart werden.

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. Armstrong-Kalkül
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Einführung

Bis jetzt:

- Nutzen- und Anforderungsanalys (Pflichtenheft)
- Entity-Relationship-Entwurf
- relationales Schema

Zu tun:

- Feintuning des erstellten Schemas auf der Basis von intrarelationalen Abhängigkeiten
  - Funktionale Abhängigkeiten
  - Kriterien für gute Schemata, schlechte Schemata
  - Normalformen
  - Algorithmen zur Normalisierung

Was ist faul mit diesem Schema?

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

Änderungs-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Angenommen Sokrates soll von Raum 226 nach Raum 233 umziehen
- Die Information ‘Raum’ existiert in diesem Fall mehrfach
- Lösung: Änderung aller Einträge gleichzeitig
  - hoher Speicherbedarf durch Redundanz
  - erhöhter Zeitbedarf bei Änderungen

Einfüge-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Schema kombiniert Informationen verschiedener unpassender Entitytypen
  - Hinzufügen eines Professors ohne Vorlesung

    ⇒ NULL-Werte in VorlNr, Titel und SWS
  - Analog: Hinzufügen einer Vorlesung zu der noch kein Dozent festgelegt wurde

    ⇒ NULL-Werte in PersNr, Name, Rang und Raum

Lösch-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Schema kombiniert Informationen verschiedener unpassender Entitytypen
  - Löschen von Elementen eines Entitytyps kann Verlust eines anderen Entitytyps bewirken
    - Löschen des Eintrags zu „Der Wiener Kreis“ (die einzige Vorlesung von Popper) würde auch Informationen zu Popper löschen
    - Alternative: Prüfen der *gesamten* Datenbank, ob dieser Eintrag die einzige Vorlesung von Popper ist. In diesem Fall durch NULL-Werte ersetzen

6. Relationale Entwurfstheorie

1. **Funktionale Abhängigkeiten**
2. Armstrong-Kalkül
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Funktionale Abhängigkeiten

- Das zentrale Konzept der relationalen Entwurfstheorie

- Sei X die Attributmenge eines Relationenschemas $\mathcal{R}$. Die funktionalen Abhängigkeiten über X bilden eine zweistellige Relation „→“ auf den Attributmengen aus X:

$$\alpha \rightarrow \beta, \qquad \text{für } \alpha, \beta \subseteq X.$$

(gesprochen: von Alpha nach Beta)

- In Worten: β ist funktional abhängig von α
  oder die α-Werte bestimmen die β-Werte funktional (d.h. eindeutig)

- Für zwei Attributmengen $\alpha, \beta \subseteq X$ und eine Relation R sagen wir R *erfüllt* die funktionale Abhängigkeit $\alpha \rightarrow \beta$, wenn gilt:

$$r.\alpha = t.\alpha \text{ impliziert } r.\beta = t.\beta \text{ für alle } r, t \in R.$$

Funktionale Abhängigkeiten

- Verallgemeinerung der Schlüsseleigenschaft
  - Eindeutigkeitseigenschaft der Schlüssel als funktionale Abhängigkeit:

$$\alpha \rightarrow X$$

- Eine funktionale Abhängigkeit $\alpha \rightarrow \beta$ lässt sich ebenfalls als intrarelationale Abhängigkeit auffassen:

$$\sigma_{\alpha \rightarrow \beta}: \mathrm{Rel}(X) \rightarrow \{true, false\}, \qquad R \longmapsto \begin{cases} true, & \text{falls } \alpha \rightarrow \beta \text{ in } R \text{ gilt} \\ false, & \text{sonst} \end{cases}$$

- Ist $\beta \subseteq \alpha$, so heißt $\alpha \rightarrow \beta$ eine *triviale Abhängigkeit*

- Funktionale Abhängigkeiten werden auch als FDs (functional dependencies) abgekürzt

Funktionale Abhängigkeiten – Beispiel

- Betrachte Schema $\mathcal{R}$ mit Attributmenge $\{A, B, C, D\}$ und FD $\{A\} \rightarrow \{B\}$.

  Die Ausprägung r erfüllt diese FD:

  nur für die Tupel t_2, t_3 gilt $t_2.A = t_3.A\ (= a_1)$

  und für diese gilt ebenfalls $t_2.B = t_3.B\ (= b_1)$

  – diese Ausprägung erfüllt auch die FDs
  - $\{A\} \rightarrow \{C\}$
  - $\{A, B\} \rightarrow \{C\}$
  - $\{C, D\} \rightarrow \{B\}$

  nicht aber die FDs
  - $\{B\} \rightarrow \{C\}$
  - $\{A, B, C\} \rightarrow \{D\}$

	r			
	A	B	C	D
t_1	a_4	b_2	c_4	d_3
t_2	a_1	b_1	c_1	d_1
t_3	a_1	b_1	c_1	d_2
t_4	a_2	b_2	c_3	d_2
t_5	a_3	b_2	c_4	d_3

Funktionale Abhängigkeiten – Beispiel

- Wichtig:
  - funktionale Abhängigkeiten beschreiben die Menge aller gültigen Relationen (wenn nichts anderes gesagt wird)
  - üblicherweise wird gefragt: welche zusätzlichen FDs lassen sich aus den gegebenen FDs ableiten
  - es wird nicht gefragt: welche zusätzlichen FDs erfüllt diese konkrete Ausprägung

	r			
	A	B	C	D
t_1	a_4	b_2	c_4	d_3
t_2	a_1	b_1	c_1	d_1
t_3	a_1	b_1	c_1	d_2
t_4	a_2	b_2	c_3	d_2
t_5	a_3	b_2	c_4	d_3

- Übrigens: die Notation

$$\{A, B, C\} \rightarrow \{D\}$$

wird häufig auch abgekürzt, z.B. durch

$$A, B, C \rightarrow D.$$

Funktionale Abhängigkeiten – Beispiel

Student			
MatrNr: *Inte*	Name: *String*	#Semester: *Int*	Status: *Status*
1234	Michael	6	eingeschrieben
5678	Andrea	4	eingeschrieben
4711	Sabine	8	beurlaubt
815	Franz	12	exmatrikuliert

$\sigma_{\alpha \to \beta}$	Wert in der Ausprägung	Wert in allen Ausprägungen
$MatrNr \rightarrow Name, Semester, Status$		
$Name \rightarrow Semester, Status, MatrNr$		
$Semester \rightarrow Status$		
$Status \rightarrow Semester$		
$MatrNr, Name \rightarrow Semester, Status$		

Funktionale Abhängigkeiten – Beispiel

Student			
MatrNr: *Inte*	Name: *String*	#Semester: *Int*	Status: *Status*
1234	Michael	6	eingeschrieben
5678	Andrea	4	eingeschrieben
4711	Sabine	8	beurlaubt
815	Franz	12	exmatrikuliert

$\sigma_{\alpha \to \beta}$	Wert in der Ausprägung	Wert in allen Ausprägungen
$MatrNr \rightarrow Name, Semester, Status$	true	true
$Name \rightarrow Semester, Status, MatrNr$	true	false
$Semester \rightarrow Status$	true	false
$Status \rightarrow Semester$	false	false
$MatrNr, Name \rightarrow Semester, Status$	true	true

Überprüfen funktionaler Abhängigkeiten

- Ein einfacher Algorithmus zur Überprüfung einer FD:
  - Eingabe: eine Relation R und eine FD $\alpha \rightarrow \beta$
  - Ausgabe: *ja* genau dann, wenn $\alpha \rightarrow \beta$ in R erfüllt ist
  - Algorithmus:
    - sortiere R nach den α-Werten
    - falls alle Gruppen, bestehend aus Tupeln mit gleichen α-Werten, auch gleiche β-Werte aufweisen: *ja*; sonst: *nein*

- Die Laufzeit dieses Algorithmus wird durch die Sortierung dominiert
  - Komplexität $O\big(n \log(n)\big)$

Schlüssel

- Präzisierung des Schlüsselbegriffs
  - Dazu sei $\mathcal{R}$ ein Relationenschema mit Attributmenge X und funktionalen Abhängigkeiten F
- Superschlüssel (Eindeutigkeit) :
  - $\alpha \subseteq X$ heißt *Superschlüssel*, falls $\alpha \rightarrow X$ gilt
  - α bestimmt also alle anderen Attributwerte
  - X selbst ist stets auch ein Superschlüssel, da trivialerweise $X \rightarrow X$ gilt
- voll funktional abhängig (Minimalität):
  - $\beta \subseteq X$ heißt *voll funktional abhängig* von α, falls
    - $\alpha \rightarrow \beta$ gilt
    - $\alpha - \{A\} \nrightarrow \beta$ für alle $A \in \alpha$ gilt, d.h. α kann nicht „verkleinert“ werden

Schlüssel

- Präzisierung des Schlüsselbegriffs
  - Dazu sei $\mathcal{R}$ ein Relationenschema mit Attributmenge X und funktionalen Abhängigkeiten F
- Schlüsselkandidat:
  - Eine Attributmenge $\alpha \subseteq X$ heißt *Schlüsselkandidat*, falls X voll funktional abhängig von α ist
- Primärschlüssel:
  - In einem Relationenschema wird einer der Schlüsselkandidaten als *Primärschlüssel* ausgewählt
  - Fremdschlüssel sollten z.B. immer nur auf den Primärschlüssel verweisen

Schlüssel – Beispiel

Orte			
Name	BLand	Vorwahl	EW
Frankfurt	Hessen	069	690.000
Frankfurt	Brandenburg	0335	60.000
München	Bayern	089	1.378.000.000
Passau	Bayern	0851	50.000

- Annahme: Ortsnamen sind innerhalb eines Bundeslandes eindeutig
- Schlüsselkandidaten:

$$\{\,\text{Name}, \text{BLand}\,\}, \qquad \{\,\text{Name}, \text{Vorwahl}\,\}$$

  beide sind minimal:
  - Städte in unterschiedlichen Bundesländern können denselben Namen besitzen
  - kleine Dörfer können sich dieselbe Vorwahl teilen

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. **Armstrong-Kalkül**
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Bestimmung aller funktionalen Abhängigkeiten

- Frage: Ausgehend von einer Menge funktionaler Abhängigkeiten F (beim Datenbank-Entwurf erstellt), welche zusätzlichen funktionalen Abhängigkeiten sind implizit immer erfüllt?

- *Beispiel*:
  - Erweiterung von Universitäts-Beispiel um Adressen
  - erster Entwurf für Professoren und Adressen:

ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

Bestimmung aller funktionalen Abhängigkeiten – Beispiel

ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

- Funktionale Abhängigkeiten F:
  - PersNr → PersNr, Name, Rang, … , Vorwahl, Bundesland
  - Raum → PersNr
  - Ort, Bundesland → Vorwahl
  - Ort, Bundesland, Straße, Hausnummer → PLZ
  - PLZ → Ort, Bundesland
- implizierte funktionale Abhängigkeiten:
  - Raum → PersNr, Name, Rang, … , Vorwahl, Bundesland
  - PLZ → Vorwahl
  - …

Implizierte funktionale Abhängigkeiten (Formalisierung)

- Seien X eine Attributmenge und F eine Menge von funktionalen Abhängigkeiten über X.
- Semantische Grundlagen:
  - Eine Relationsausprägung R *erfüllt* F, falls R jede funktionale Abhängigkeit $f \in F$ erfüllt
    (man sagt auch: R ist ein Modell von F)
  - Schreibweise:
    $$R \vDash F \text{ oder } R \vDash f$$
  - Die Menge aller *gültigen Ausprägungen* des Schemas $\mathcal{R}$ ist
    $$\text{Sat}(F) = \{ R \in \text{Rel}(X) \mid R \vDash F \}$$
  - Zwei Mengen F, G von funktionalen Abhängigkeiten über X heißen *äquivalent*, falls sie die gleichen gültigen Relationen definieren:
    $$\text{Sat}(F) = \text{Sat}(G)$$

Implizierte funktionale Abhängigkeiten (Definition)

- Seien X eine Attributmenge und F eine Menge von funktionalen Abhängigkeiten über X.
- *Implikation*:
  - Wir sagen, dass die Menge funktionaler Abhängigkeiten F die funktionale Abhängigkeit f impliziert, falls alle F erfüllenden Relationenausprägungen $R \in \mathrm{Sat}(F)$ auch f erfüllen.
  - Schreibweise:

$$F \models f$$

$F \models f$ ist eine semantische Beziehung

- nicht praktikabel: Überprüfe jede gültige Ausprägung $R \in \mathrm{Sat}(F)$, ob f gilt
- stattdessen: Armstrong-Kalkül

Armstrong-Kalkül

- Algorithmische Bestimmung aller implizierten funktionalen Abhängigkeiten
- Hilfsmittel: Armstrong-Axiome

  Seien $\alpha, \beta, \gamma \subseteq X$ Attributmengen.

  – Reflexivität (A_1): Ist $\beta \subseteq \alpha$ eine Teilmenge von α, so gilt auch $\alpha \rightarrow \beta$.

  – Verstärkung (A_2): Falls $\alpha \rightarrow \beta$ gilt, so gilt auch $\alpha\gamma \rightarrow \beta\gamma$. Wobei hier $\alpha\gamma \coloneqq \alpha \cup \gamma$.

  – Transitivität (A_3): Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, so gilt auch $\alpha \rightarrow \gamma$.

Armstrong-Kalkül

- Algorithmische Bestimmung aller implizierten funktionalen Abhängigkeiten
- *Ableitbar* :
  - Wir sagen, dass die funktionale Abhängigkeit f aus der Menge der funktionalen Abhängigkeiten F *ableitbar* ist, falls:

    Es gibt eine endliche Folge $f_1, \ldots, f_{n-1}, f_n = f$, sodass für jedes $1 \leq i \leq n$ gilt:

    f_i erhält man aus $F \cup \{f_1, \ldots, f_{i-1}\}$ durch Anwendung der Axiome A_1, A_2 oder A_3

  - Schreibweise

$$F \vdash f$$

$F \vdash f$ ist eine syntaktische Beziehung

Armstrong-Kalkül – Beispiel

- *Ableitbar* :

  ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

- Funktionale Abhängigkeiten F:
  - $\text{Ort}, \text{Bundesland} \rightarrow \text{Vorwahl}$
  - $\text{PLZ} \rightarrow \text{Ort}, \text{Bundesland}$
  - ...
- Mithilfe der Transitivitätsregel (A_3) :Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, so gilt auch $\alpha \rightarrow \gamma$ erhält man aus

  $$\text{PLZ} \rightarrow \text{Ort}, \text{Bundesland},$$

  $$\text{Ort}, \text{Bundesland} \rightarrow \text{Vorwahl}$$

  die funktionale Abhängigkeit $\text{PLZ} \rightarrow \text{Vorwahl}$.

  $$F \vdash (\text{PLZ} \rightarrow \text{Vorwahl})$$

Armstrong-Kalkül – Korrektheit und Vollständigkeit

- Liefert das Armstrong-Kalkül alle „gültigen“ bzw. von F implizierten FDs?

  Sei X eine Attributmenge und F eine Menge von FDs über X

  – Zu F nennen wir $F^+ = \{ f \mid F \vdash f \}$ die *(geschlossene) Hülle* von F

  – Gilt also

$$F^+ = \{ f \mid F \models f \}\,?$$

- Ja, der Armstrong-Kalkül ist *korrekt* und *vollständig*.

  – *korrekt*: Für jede funktionale Abhängigkeit f mit $F \vdash f$ gilt auch $F \models f$

  (es lassen sich nur „gültige“ funktionale Abhängigkeiten ableiten, „$\subseteq$“)

  – *vollständig*: Jede von F implizierte funktionale Abhängigkeit f (also $F \models f$) lässt sich mithilfe des Armstrong-Kalküls ableiten, d.h. $F \vdash f$ („$\supseteq$“)

  (Beweis der Korrektheit jetzt, Beweis der Vollständigkeit später)

Armstrong-Kalkül – Korrektheit

- Korrektheit der Reflexivitätsregel (A_1): Für $\beta \subseteq \alpha$ gilt $\alpha \rightarrow \beta$.
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationsausprägung, $\alpha \subseteq X$ und $\beta \subseteq \alpha$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha] = t_2[\alpha]$. Dann gilt auch $t_1[\beta] = t_2[\beta]$.

    Insgesamt: $\alpha \rightarrow \beta$ gilt auch in R.

Armstrong-Kalkül – Korrektheit

- Korrektheit der Verstärkungsregel (A_2): Für $\alpha \rightarrow \beta$ gilt auch $\alpha\gamma \rightarrow \beta\gamma$.
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationsausprägung, $\alpha, \beta, \gamma \subseteq X$ und $\alpha \rightarrow \beta \in F$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha \cup \gamma] = t_2[\alpha \cup \gamma]$. Dann gilt auch $t_1[\beta] = t_2[\beta]$ und $t_1[\gamma] = t_2[\gamma]$. Daraus folgt $t_1[\beta \cup \gamma] = \mathrm{t}_2[\beta \cup \gamma]$.
    Insgesamt: $\alpha \cup \gamma \rightarrow \beta \cup \gamma$ gilt auch in R.

Armstrong-Kalkül – Korrektheit

- Korrektheit der Transitivitätsregel (A_3): Für $\alpha \to \beta, \beta \to \gamma$ gilt auch $\alpha \to \gamma$
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationenausprägung, $\alpha, \beta, \gamma \subseteq X$ und $\alpha \to \beta,\ \beta \to \gamma \in F$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha] = t_2[\alpha]$. Wegen $\alpha \to \beta$, gilt also auch $t_1[\beta] = t_2[\beta]$. Wegen $\beta \to \gamma$, gilt dann auch $t_1[\gamma] = t_2[\gamma]$. Insgesamt: $\alpha \to \gamma$ gilt auch in R.

Armstrong-Kalkül – Erweiterung

Es ist für den Herleitungsprozess komfortabel, weitere Regeln hinzuzunehmen

- Erweiterung der Armstrong-Axiome um drei Regeln:

  Seien $\alpha, \beta, \gamma, \delta \subseteq X$ Attributmengen.

  - Vereinigung (A_4): Gelten $\alpha \to \beta$ und $\alpha \to \gamma$, so gilt auch $\alpha \to \beta\gamma$.
  - Dekomposition (A_5): Falls $\alpha \to \beta\gamma$ gilt, so gelten auch $\alpha \to \beta$ und $\alpha \to \gamma$.
  - Pseudotransitivität (A_6): Falls $\alpha \to \beta$ und $\beta\gamma \to \delta$ gilt, so gilt auch $\alpha\gamma \to \delta$.

Armstrong-Kalkül – Erweiterung

- Ableitung der Vereinigungsregel (A_4) :Gelten $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$, so gilt auch $\alpha \rightarrow \beta\gamma$.
  - Seien $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma \in F$. Über die Grundregeln erhalten wir

$$(A_2): \text{Da } \alpha \rightarrow \beta \text{ gilt } \alpha\gamma \rightarrow \beta\gamma$$

$$(A_2): \text{Da } \alpha \rightarrow \gamma \text{ gilt } \alpha \rightarrow \alpha\gamma$$

$$(A_3): \text{Da } \alpha \rightarrow \alpha\gamma \text{ und } \alpha\gamma \rightarrow \beta\gamma \text{ gilt } \alpha \rightarrow \beta\gamma$$

Armstrong-Kalkül – Erweiterung

- Ableitung der Dekompositionsregel ($A_{5)}$):

  Falls $\alpha \rightarrow \beta\gamma$ gilt, so gelten auch $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$.

  – Sei $\alpha \rightarrow \beta\gamma \in F$. Über die Grundregeln erhalten wir

$$(A_1) : \text{Da } \beta\gamma := \beta \cup \gamma \text{ gilt } \beta\gamma \rightarrow \beta$$

$$(A_1) : \text{Da } \beta\gamma := \beta \cup \gamma \text{ gilt } \beta\gamma \rightarrow \gamma$$

$$(A_3) : \text{Da } \alpha \rightarrow \beta\gamma \text{ und } \beta\gamma \rightarrow \beta \text{ gilt } \alpha \rightarrow \beta$$

$$(A_3) : \text{Da } \alpha \rightarrow \beta\gamma \text{ und } \beta\gamma \rightarrow \gamma \text{ gilt } \alpha \rightarrow \gamma$$

Armstrong-Kalkül – Erweiterung

- Ableitung der Pseudotransitivitätsregel (A_6) :

  Falls $\alpha \to \beta$ und $\beta\gamma \to \delta$ gilt, so gilt auch $\alpha\gamma \to \delta$

  Sei $\alpha \to \beta$ und $\beta\gamma \to \delta \in F$. Über die Grundregeln erhalten wir

  $$(A_2) : \text{Da } \alpha \to \beta \text{ gilt } \alpha\gamma \to \beta\gamma$$

  $$(A_3) : \text{Da } \alpha\gamma \to \beta\gamma \text{ und } \beta\gamma \to \delta \text{ gilt } \alpha\gamma \to \delta$$

Armstrong-Kalkül – Vereinigung

ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

- Funktionale Abhängigkeiten F:
  - Ort, Bundesland → Vorwahl
  - Ort, Bundesland, Straße, Hausnummer → PLZ
  - …

- Beispiel: f = Ort, Bundesland, Straße, Hausnummer → PLZ, Vorwahl ist ebenfalls aus F ableitbar
  - $(A_1) : f_1 = (\text{Ort, Bundesland, Straße, Hausnummer} \rightarrow \text{Ort, Bundesland})$
  - $(A_3) : f_2 = (\text{Ort, Bundesland, Straße, Hausnummer} \rightarrow \text{Vorwahl})$
  - $(\boldsymbol{A_4}) : \boldsymbol{f = f_3} = (\textbf{Ort, Bundesland, Strasse, Hausnummer} \rightarrow \textbf{PLZ, Vorwahl})$

Attributhülle

- Oft ist man nicht an der gesamten Hülle F^+ interessiert:
  - Welche Attribute sind unter einer gegebenen Menge von FDs F von einer bestimmten Attributmenge α funktional bestimmt?
  - Man nennt α^+ die Attributhülle von α unter F

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \rightarrow \beta \in F^+ \text{ mit } x \in \beta \}$$

  - Algorithmus zur Bestimmung von $\text{AttrHülle}(F, \alpha)$:
    - $Erg := \alpha$

      while (Änderungen an Erg) **do**

        foreach FD $\beta \rightarrow \gamma$ **in** F **do**

          if $\beta \subseteq Erg$ **then** $Erg := Erg \cup \gamma$;

      Ausgabe $\alpha^+ = Erg$;

$\kappa \subseteq X$ ist genau dann ein Superschlüssel, falls gilt:
$\kappa^+ = X$

Attributhülle – Beispiel

- Attributhülle

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \rightarrow \beta \in F^+ \text{ mit } x \in \beta \}$$

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

  - AttrHülle(F, {B}):

    $Erg = \{B\}$

    Durchlaufe F:

    $A \rightarrow C$: $\{B\}$, $B \rightarrow A$: $\{A, B\}$, $AB \rightarrow C$: $\{A, B, C\}$

    Durchlaufe F nochmal:

    keine Änderung

    Ausgabe $B^+ = \{A, B, C\}$

Attributhülle

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \to \beta \in F^+ \text{ mit } x \in \beta \}$$

- Lemma **L1**: Die Attributhülle besitzt folgende Eigenschaft.

  Für jede Teilmenge $V \subseteq \alpha^+$ gilt auch $\alpha \to V \in F^+$.

- Beweis (Skizze):

  Wir beschränken uns auf den Fall $V = \{a_1, a_2\}$. Da $V \subseteq \alpha^+$ ist, gibt es $\beta_1 = \{a_1, \dots\}$ und $\beta_2 = \{a_2, \dots\}$ mit $\alpha \to \beta_1,\ \alpha \to \beta_2 \in F^+$. Mit der Reflexivitätsregel (A_1) sind auch $\beta_1 \to \{a_1\}, \beta_2 \to \{a_2\} \in F^+$. Mit der Transitivitätsregel (A_3) und

  $$\alpha \to \beta_1, \qquad \beta_1 \to \{a_1\} \alpha \to \beta_2, \qquad \beta_2 \to \{a_2\}$$

  sind auch $\alpha \to \{a_1\}, \alpha \to \{a_2\} \in F^+$. Mit der Vereinigungsregel (A_4) erhalten wir

  $$\alpha \to \{a_1, a_2\} \in F^+.$$

Armstrong-Kalkül – Vollständigkeit

- Das Armstrong-Kalkül ist vollständig, d.h. jede von funktionalen Abhängigkeiten F implizierte FD f lässt sich mithilfe des Armstrong-Kalküls ableiten. Es gilt also

$$F \models f \Rightarrow F \vdash f$$

- Beweis (durch Kontraposition):
  - Wir zeigen die Aussage $F \nvdash f \Rightarrow F \not\models f$. D.h. ist eine funktionale Abhängigkeit f nicht ableitbar, so wird sie auch nicht von F impliziert. Sei dazu $f = \alpha \rightarrow \beta$ eine FD mit $F \nvdash f$ und X die Menge aller Attribute aus F und f.
  - Um zu zeigen, dass $F \not\models f$ gilt, konstruieren wir eine Relation R, in der F gilt, aber f nicht.

  Konstruktion:

  Seien $\alpha^+ = \{a_1, \ldots, a_n\}$ und $X \setminus \alpha^+ = \{b_1, \ldots, b_m\}$.

  Konstruiere R wie rechts.

R	$a_1, \ldots, a_n$	$b_1, \ldots, b_m$
r	$1, \ldots, 1$	$1, \ldots, 1$
t	$1, \ldots, 1$	$0, \ldots, 0$

Armstrong-Kalkül – Vollständigkeit

- Zeige für Vollständigkeit: $F \nvdash f \Rightarrow F \nvDash f$
- Hilfslemma 1:

  Es gilt $R \vDash F$, d.h. alle FDs in F werden von R erfüllt.
- Beweis (durch Widerspruch):

  Angenommen es existiert $V \rightarrow W \in F$ mit $R \nvDash V \rightarrow W$. Dann muss für die beiden Tupel $r, t \in R$ gelten, dass $r[V] = t[V]$ und $r[W] \neq t[W]$ ist. Nach Konstruktion von R kann $r[V] = t[V]$ nur gelten, wenn $V \subseteq \{a_1, \ldots, a_n\} = \alpha^+$ ist. Ebenso impliziert $r[W] \neq t[W]$, dass $W \not\subseteq \alpha^+$ ist.

  Nach dem vorigen Lemma **L1** folgt aus $V \subseteq \alpha^+$, dass $\alpha \rightarrow V \in F^+$ ist. Mit der Transivititätsregel (A_3) erhalten wir aus $\alpha \rightarrow V,\ V \rightarrow W \in F^+$, dass auch $\alpha \rightarrow W \in F^+$ ist.

  Dies liefert einen Widerspruch zu $W \not\subseteq \alpha^+$.

Armstrong-Kalkül – Vollständigkeit

- Zeige für Vollständigkeit: $F \nvdash f \Rightarrow F \nvDash f$
- Hilfslemma 2:

  Es gilt $R \nvDash f$, d.h. f wird von R nicht erfüllt.
- Beweis:

  Da f nicht aus F ableitbar ist $(F \nvdash f)$ gilt insbesondere $\beta \not\subseteq \alpha^+$. Also existiert ein $b_k \in \beta$ mit $b_k \in X \setminus \alpha^+$. Direkt aus der Konstruktion von R folgt dann, dass $r[\alpha] = t[\alpha]$ ist, aber $r[b_k] = 1 \neq 0 = t[b_k]$.

  Insgesamt erhalten wir, dass $f = \alpha \rightarrow \beta$ von R nicht erfüllt wird.
- Es gilt also $R \vDash F$, aber $R \nvDash f$. Darauf folgt $F \nvDash f$.

Kanonische Überdeckung – Motivation

- Um für zwei Mengen F und G zu entscheiden, ob sie äquivalent sind ($\mathrm{Sat}(F) = \mathrm{Sat}(G)$), reicht es $F^+ = G^+$ zu überprüfen.

Warum?
Freiwillige Übung

- Im Allgemeinen ist die Hülle F^+ einer Menge von FDs sehr groß
- Vor allem bei Datenbankmodifikationen:
  - Überprüfen der Konsistenz anhand von F^+ sehr aufwändig (auch viele triviale Abhängigkeiten)
  - minimale Menge von „erzeugenden“ funktionalen Abhängigkeiten wünschenswert

- Statt der Hülle F^+: kanonische Überdeckung

Kanonische Überdeckung (Definition)

- Sei F eine Menge von funktionalen Abhängigkeiten über einer Attributmenge X. Dann heißt F_c eine *kanonische Überdeckung* von F, falls gilt:
  1. $F_c^+ = F^+$, d.h. $\mathrm{Sat}(F_c) = \mathrm{Sat}(F)$ (äquivalent)
  2. In F_c existieren keine FDs $\alpha \rightarrow \beta$ bei denen α oder β überflüssige Attribute enthalten. D.h.
     - Für alle $A \in \alpha$: $\left(F_c \setminus (\alpha \rightarrow \beta)\right) \cup (\alpha \setminus A \rightarrow \beta) \not\equiv F_c$ (nicht äquivalent)
     - Für alle $B \in \beta$: $\left(F_c \setminus (\alpha \rightarrow \beta)\right) \cup (\alpha \rightarrow \beta \setminus B) \not\equiv F_c$
  3. Jede linke Seite einer FD ist einzigartig in F_c (sonst ersetze durch Vereinigungen)

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

Eine kanonische Überdeckung ist $\mathrm{F_c} = \{A \rightarrow C, B \rightarrow A\}$.
(Algorithmus: nächste Folie)

Kanonische Überdeckung – Algorithmus

- Eingabe: Menge von FDs F, Ausgabe: Kanonische Überdeckung F_c

1. Setze $F_c = F$
2. Führe für jede FD $\alpha \rightarrow \beta \in F_c$ eine Linksreduktion durch:
   - Überprüfe für alle $A \in \alpha$, ob A überflüssig ist. D.h. ob gilt: $\beta \subseteq \text{AttrHülle}(F_c, \alpha \setminus A)$

     Falls ja: ersetze $\alpha \rightarrow \beta$ durch $\alpha \setminus A \rightarrow \beta$.
3. Führe für jede FD $\alpha \rightarrow \beta \in F_c$ eine Rechtsreduktion durch:
   - Überprüfe für alle $B \in \beta$, ob B überflüssig ist. D.h. ob gilt:

     $$B \in \text{AttrHülle}(\big(F_c \setminus (\alpha \rightarrow \beta)\big) \cup (\alpha \rightarrow \beta \setminus B), \alpha)$$

     Falls ja: ersetze $\alpha \rightarrow \beta$ durch $\alpha \rightarrow \beta \setminus B$.
4. Entferne die im 3. Schritt entstandenen FDs der Form $\alpha \rightarrow \emptyset$
5. Fasse über die Vereinigungsregel FDs der Form $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$

   $$\alpha \rightarrow \beta_1 \cup \cdots \cup \beta_n$$

Kanonische Überdeckung – Beispiel

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

1. Linksreduktion:
   - $A \rightarrow C$: C ist nicht in $(\{A\} \setminus \{A\})^+ = \emptyset^+$. Ok
   - $B \rightarrow A$: Ok
   - $AB \rightarrow C$: Überprüfe A. Ist $C \in (\{A, B\} \setminus \{A\})^+$? Ja, denn

$$B^+ = \{ \quad \}$$

   Ersetze $AB \rightarrow C$ durch $B \rightarrow C$.

   B muss nun in $B \rightarrow C$ nicht mehr überprüft werden.

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow C\}$$

Kanonische Überdeckung – Beispiel

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

1. Linksreduktion:
   - $A \rightarrow C$: C ist nicht in $(\{A\} \setminus \{A\})^+ = \emptyset^+$. Ok
   - $B \rightarrow A$: Ok
   - $AB \rightarrow C$: Überprüfe A. Ist $C \in (\{A, B\} \setminus \{A\})^+$? Ja, denn

$$B^+ = \{B, A, C\}$$

     Ersetze $AB \rightarrow C$ durch $B \rightarrow C$.

     B muss nun in $B \rightarrow C$ nicht mehr überprüft werden.

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow C\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\},\ A)$

     $= \{$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\},\ B)$

     $= \{$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\},\ A)$

$$= \{A\}$$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\},\ B)$

$$= \{$$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\},\ A)$

$$= \{A\}$$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\},\ B)$

$$= \{B, A, C\}$$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow \emptyset\}$$

3. Entferne $\alpha \rightarrow \emptyset$:
   - Entferne die Abhängigkeit $B \rightarrow \emptyset$
4. Vereinigen:
   - Hier ist nichts zu tun.

- Ergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A\}$$

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. Armstrong-Kalkül
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Einführung

Bis jetzt:

- Nutzen- und Anforderungsanalys (Pflichtenheft)
- Entity-Relationship-Entwurf
- relationales Schema

Zu tun:

- Feintuning des erstellten Schemas auf der Basis von intrarelationalen Abhängigkeiten
  - Funktionale Abhängigkeiten
  - Kriterien für gute Schemata, schlechte Schemata
  - Normalformen
  - Algorithmen zur Normalisierung

Was ist faul mit diesem Schema?

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

Änderungs-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Angenommen Sokrates soll von Raum 226 nach Raum 233 umziehen
- Die Information ‘Raum’ existiert in diesem Fall mehrfach
- Lösung: Änderung aller Einträge gleichzeitig
  - hoher Speicherbedarf durch Redundanz
  - erhöhter Zeitbedarf bei Änderungen

Einfüge-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Schema kombiniert Informationen verschiedener unpassender Entitytypen
  - Hinzufügen eines Professors ohne Vorlesung

    ⇒ NULL-Werte in VorlNr, Titel und SWS
  - Analog: Hinzufügen einer Vorlesung zu der noch kein Dozent festgelegt wurde

    ⇒ NULL-Werte in PersNr, Name, Rang und Raum

Lösch-Anomalie

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	W2	52	5259	Der Wiener Kreis	2
2137	Kant	W3	7	4630	Die 3 Kritiken	4

- Schema kombiniert Informationen verschiedener unpassender Entitytypen
  - Löschen von Elementen eines Entitytyps kann Verlust eines anderen Entitytyps bewirken
    - Löschen des Eintrags zu „Der Wiener Kreis“ (die einzige Vorlesung von Popper) würde auch Informationen zu Popper löschen
    - Alternative: Prüfen der *gesamten* Datenbank, ob dieser Eintrag die einzige Vorlesung von Popper ist. In diesem Fall durch NULL-Werte ersetzen

6. Relationale Entwurfstheorie

1. **Funktionale Abhängigkeiten**
2. Armstrong-Kalkül
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Funktionale Abhängigkeiten

- Das zentrale Konzept der relationalen Entwurfstheorie

- Sei X die Attributmenge eines Relationenschemas $\mathcal{R}$. Die funktionalen Abhängigkeiten über X bilden eine zweistellige Relation „→“ auf den Attributmengen aus X:

$$\alpha \rightarrow \beta, \qquad \text{für } \alpha, \beta \subseteq X.$$

(gesprochen: von Alpha nach Beta)

- In Worten: β ist funktional abhängig von α
  oder die α-Werte bestimmen die β-Werte funktional (d.h. eindeutig)

- Für zwei Attributmengen $\alpha, \beta \subseteq X$ und eine Relation R sagen wir R *erfüllt* die funktionale Abhängigkeit $\alpha \rightarrow \beta$, wenn gilt:

$$r.\alpha = t.\alpha \text{ impliziert } r.\beta = t.\beta \text{ für alle } r, t \in R.$$

Funktionale Abhängigkeiten

- Verallgemeinerung der Schlüsseleigenschaft
  - Eindeutigkeitseigenschaft der Schlüssel als funktionale Abhängigkeit:

$$\alpha \rightarrow X$$

- Eine funktionale Abhängigkeit $\alpha \rightarrow \beta$ lässt sich ebenfalls als intrarelationale Abhängigkeit auffassen:

$$\sigma_{\alpha \rightarrow \beta}: \mathrm{Rel}(X) \rightarrow \{true, false\}, \qquad R \longmapsto \begin{cases} true, & \text{falls } \alpha \rightarrow \beta \text{ in } R \text{ gilt} \\ false, & \text{sonst} \end{cases}$$

- Ist $\beta \subseteq \alpha$, so heißt $\alpha \rightarrow \beta$ eine *triviale Abhängigkeit*

- Funktionale Abhängigkeiten werden auch als FDs (functional dependencies) abgekürzt

Funktionale Abhängigkeiten – Beispiel

- Betrachte Schema $\mathcal{R}$ mit Attributmenge $\{A, B, C, D\}$ und FD $\{A\} \rightarrow \{B\}$.

  Die Ausprägung r erfüllt diese FD:

  nur für die Tupel t_2, t_3 gilt $t_2.A = t_3.A\ (= a_1)$
  und für diese gilt ebenfalls $t_2.B = t_3.B\ (= b_1)$

  – diese Ausprägung erfüllt auch die FDs
  - $\{A\} \rightarrow \{C\}$
  - $\{A, B\} \rightarrow \{C\}$
  - $\{C, D\} \rightarrow \{B\}$

  nicht aber die FDs
  - $\{B\} \rightarrow \{C\}$
  - $\{A, B, C\} \rightarrow \{D\}$

	r			
	A	B	C	D
t_1	a_4	b_2	c_4	d_3
t_2	a_1	b_1	c_1	d_1
t_3	a_1	b_1	c_1	d_2
t_4	a_2	b_2	c_3	d_2
t_5	a_3	b_2	c_4	d_3

Funktionale Abhängigkeiten – Beispiel

- Wichtig:
  - funktionale Abhängigkeiten beschreiben die Menge aller gültigen Relationen (wenn nichts anderes gesagt wird)
  - üblicherweise wird gefragt: welche zusätzlichen FDs lassen sich aus den gegebenen FDs ableiten
  - es wird nicht gefragt: welche zusätzlichen FDs erfüllt diese konkrete Ausprägung

	r			
	A	B	C	D
t_1	a_4	b_2	c_4	d_3
t_2	a_1	b_1	c_1	d_1
t_3	a_1	b_1	c_1	d_2
t_4	a_2	b_2	c_3	d_2
t_5	a_3	b_2	c_4	d_3

- Übrigens: die Notation

$$\{A, B, C\} \rightarrow \{D\}$$

wird häufig auch abgekürzt, z.B. durch

$$A, B, C \rightarrow D.$$

Funktionale Abhängigkeiten – Beispiel

Student			
MatrNr: *Inte*	Name: *String*	#Semester: *Int*	Status: *Status*
1234	Michael	6	eingeschrieben
5678	Andrea	4	eingeschrieben
4711	Sabine	8	beurlaubt
815	Franz	12	exmatrikuliert

$\sigma_{\alpha \to \beta}$	Wert in der Ausprägung	Wert in allen Ausprägungen
$MatrNr \rightarrow Name, Semester, Status$		
$Name \rightarrow Semester, Status, MatrNr$		
$Semester \rightarrow Status$		
$Status \rightarrow Semester$		
$MatrNr, Name \rightarrow Semester, Status$		

Überprüfen funktionaler Abhängigkeiten

- Ein einfacher Algorithmus zur Überprüfung einer FD:
  - Eingabe: eine Relation R und eine FD $\alpha \rightarrow \beta$
  - Ausgabe: *ja* genau dann, wenn $\alpha \rightarrow \beta$ in R erfüllt ist
  - Algorithmus:
    - sortiere R nach den α-Werten
    - falls alle Gruppen, bestehend aus Tupeln mit gleichen α-Werten, auch gleiche β-Werte aufweisen: *ja*; sonst: *nein*

- Die Laufzeit dieses Algorithmus wird durch die Sortierung dominiert
  - Komplexität $O\big(n \log(n)\big)$

Schlüssel

- Präzisierung des Schlüsselbegriffs
  - Dazu sei $\mathcal{R}$ ein Relationenschema mit Attributmenge X und funktionalen Abhängigkeiten F
- Superschlüssel (Eindeutigkeit) :
  - $\alpha \subseteq X$ heißt *Superschlüssel*, falls $\alpha \rightarrow X$ gilt
  - α bestimmt also alle anderen Attributwerte
  - X selbst ist stets auch ein Superschlüssel, da trivialerweise $X \rightarrow X$ gilt
- voll funktional abhängig (Minimalität):
  - $\beta \subseteq X$ heißt *voll funktional abhängig* von α, falls
    - $\alpha \rightarrow \beta$ gilt
    - $\alpha - \{A\} \nrightarrow \beta$ für alle $A \in \alpha$ gilt, d.h. α kann nicht „verkleinert“ werden

Schlüssel

- Präzisierung des Schlüsselbegriffs
  - Dazu sei $\mathcal{R}$ ein Relationenschema mit Attributmenge X und funktionalen Abhängigkeiten F
- Schlüsselkandidat:
  - Eine Attributmenge $\alpha \subseteq X$ heißt *Schlüsselkandidat*, falls X voll funktional abhängig von α ist
- Primärschlüssel:
  - In einem Relationenschema wird einer der Schlüsselkandidaten als *Primärschlüssel* ausgewählt
  - Fremdschlüssel sollten z.B. immer nur auf den Primärschlüssel verweisen

Schlüssel – Beispiel

Orte			
Name	BLand	Vorwahl	EW
Frankfurt	Hessen	069	690.000
Frankfurt	Brandenburg	0335	60.000
München	Bayern	089	1.378.000.000
Passau	Bayern	0851	50.000

- Annahme: Ortsnamen sind innerhalb eines Bundeslandes eindeutig
- Schlüsselkandidaten:

$$\{\,\text{Name}, \text{BLand}\,\}, \qquad \{\,\text{Name}, \text{Vorwahl}\,\}$$

  beide sind minimal:
  - Städte in unterschiedlichen Bundesländern können denselben Namen besitzen
  - kleine Dörfer können sich dieselbe Vorwahl teilen

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. **Armstrong-Kalkül**
3. Zerlegung von Relationen
4. Normalformen und Normalisierungen

Bestimmung aller funktionalen Abhängigkeiten

- Frage: Ausgehend von einer Menge funktionaler Abhängigkeiten F (beim Datenbank-Entwurf erstellt), welche zusätzlichen funktionalen Abhängigkeiten sind implizit immer erfüllt?

- *Beispiel*:
  - Erweiterung von Universitäts-Beispiel um Adressen
  - erster Entwurf für Professoren und Adressen:

$$\text{ProfessorenAdressen}(\text{PersNr}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnummer}, \text{PLZ}, \text{Vorwahl}, \text{Bundesland})$$

Bestimmung aller funktionalen Abhängigkeiten – Beispiel

ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

- Funktionale Abhängigkeiten F:
  - PersNr → PersNr, Name, Rang, … , Vorwahl, Bundesland
  - Raum → PersNr
  - Ort, Bundesland → Vorwahl
  - Ort, Bundesland, Straße, Hausnummer → PLZ
  - PLZ → Ort, Bundesland
- implizierte funktionale Abhängigkeiten:
  - Raum → PersNr, Name, Rang, … , Vorwahl, Bundesland
  - PLZ → Vorwahl
  - …

Implizierte funktionale Abhängigkeiten (Formalisierung)

- Seien X eine Attributmenge und F eine Menge von funktionalen Abhängigkeiten über X.
- Semantische Grundlagen:
  - Eine Relationsausprägung R *erfüllt* F, falls R jede funktionale Abhängigkeit $f \in F$ erfüllt

    (man sagt auch: R ist ein Modell von F)
  - Schreibweise:

$$R \vDash F \text{ oder } R \vDash f$$

  - Die Menge aller *gültigen Ausprägungen* des Schemas $\mathcal{R}$ ist

$$\text{Sat}(F) = \{ R \in \text{Rel}(X) \mid R \vDash F \}$$

  - Zwei Mengen F, G von funktionalen Abhängigkeiten über X heißen *äquivalent*, falls sie die gleichen gültigen Relationen definieren:

$$\text{Sat}(F) = \text{Sat}(G)$$

Implizierte funktionale Abhängigkeiten (Definition)

- Seien X eine Attributmenge und F eine Menge von funktionalen Abhängigkeiten über X.
- *Implikation*:
  - Wir sagen, dass die Menge funktionaler Abhängigkeiten F die funktionale Abhängigkeit f impliziert, falls alle F erfüllenden Relationenausprägungen $R \in \mathrm{Sat}(F)$ auch f erfüllen.
  - Schreibweise:

$$F \models f$$

$F \models f$ ist eine semantische Beziehung

- nicht praktikabel: Überprüfe jede gültige Ausprägung $R \in \mathrm{Sat}(F)$, ob f gilt
- stattdessen: Armstrong-Kalkül

Armstrong-Kalkül

- Algorithmische Bestimmung aller implizierten funktionalen Abhängigkeiten
- Hilfsmittel: Armstrong-Axiome

  Seien $\alpha, \beta, \gamma \subseteq X$ Attributmengen.

  – Reflexivität (A_1): Ist $\beta \subseteq \alpha$ eine Teilmenge von α, so gilt auch $\alpha \rightarrow \beta$.

  – Verstärkung (A_2): Falls $\alpha \rightarrow \beta$ gilt, so gilt auch $\alpha\gamma \rightarrow \beta\gamma$. Wobei hier $\alpha\gamma \coloneqq \alpha \cup \gamma$.

  – Transitivität (A_3): Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, so gilt auch $\alpha \rightarrow \gamma$.

Armstrong-Kalkül

- Algorithmische Bestimmung aller implizierten funktionalen Abhängigkeiten
- *Ableitbar* :
  - Wir sagen, dass die funktionale Abhängigkeit f aus der Menge der funktionalen Abhängigkeiten F *ableitbar* ist, falls:

    Es gibt eine endliche Folge $f_1, \ldots, f_{n-1}, f_n = f$, sodass für jedes $1 \leq i \leq n$ gilt:

    f_i erhält man aus $F \cup \{f_1, \ldots, f_{i-1}\}$ durch Anwendung der Axiome A_1, A_2 oder A_3

  - Schreibweise

$$F \vdash f$$

$F \vdash f$ ist eine syntaktische Beziehung

Armstrong-Kalkül – Beispiel

- *Ableitbar* :

  $\text{ProfessorenAdressen}(\text{PersNr}, \text{Name}, \text{Rang}, \text{Raum}, \quad \text{Ort}, \text{Straße}, \text{Hausnummer}, \text{PLZ}, \text{Vorwahl}, \text{Bundesland})$

- Funktionale Abhängigkeiten F:
  - $\text{Ort}, \text{Bundesland} \rightarrow \text{Vorwahl}$
  - $\text{PLZ} \rightarrow \text{Ort}, \text{Bundesland}$
  - ...
- Mithilfe der Transitivitätsregel (A_3) :Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, so gilt auch $\alpha \rightarrow \gamma$ erhält man aus

  $$\text{PLZ} \rightarrow \text{Ort}, \text{Bundesland},$$

  $$\text{Ort}, \text{Bundesland} \rightarrow \text{Vorwahl}$$

  die funktionale Abhängigkeit $\text{PLZ} \rightarrow \text{Vorwahl}$.

  $$F \vdash (\text{PLZ} \rightarrow \text{Vorwahl})$$

Armstrong-Kalkül – Korrektheit und Vollständigkeit

- Liefert das Armstrong-Kalkül alle „gültigen“ bzw. von F implizierten FDs?

  Sei X eine Attributmenge und F eine Menge von FDs über X

  – Zu F nennen wir $F^+ = \{ f \mid F \vdash f \}$ die *(geschlossene) Hülle* von F

  – Gilt also

$$F^+ = \{ f \mid F \models f \}\,?$$

- Ja, der Armstrong-Kalkül ist *korrekt* und *vollständig*.

  – *korrekt*: Für jede funktionale Abhängigkeit f mit $F \vdash f$ gilt auch $F \models f$

  (es lassen sich nur „gültige“ funktionale Abhängigkeiten ableiten, „$\subseteq$“)

  – *vollständig*: Jede von F implizierte funktionale Abhängigkeit f (also $F \models f$) lässt sich mithilfe des Armstrong-Kalküls ableiten, d.h. $F \vdash f$ („$\supseteq$“)

  (Beweis der Korrektheit jetzt, Beweis der Vollständigkeit später)

Armstrong-Kalkül – Korrektheit

- Korrektheit der Reflexivitätsregel (A_1): Für $\beta \subseteq \alpha$ gilt $\alpha \rightarrow \beta$.
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationsausprägung, $\alpha \subseteq X$ und $\beta \subseteq \alpha$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha] = t_2[\alpha]$. Dann gilt auch $t_1[\beta] = t_2[\beta]$.

    Insgesamt: $\alpha \rightarrow \beta$ gilt auch in R.

Armstrong-Kalkül – Korrektheit

- Korrektheit der Verstärkungsregel (A_2): Für $\alpha \rightarrow \beta$ gilt auch $\alpha\gamma \rightarrow \beta\gamma$.
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationsausprägung, $\alpha, \beta, \gamma \subseteq X$ und $\alpha \rightarrow \beta \in F$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha \cup \gamma] = t_2[\alpha \cup \gamma]$. Dann gilt auch $t_1[\beta] = t_2[\beta]$ und $t_1[\gamma] = t_2[\gamma]$. Daraus folgt $t_1[\beta \cup \gamma] = \mathrm{t}_2[\beta \cup \gamma]$.

    Insgesamt: $\alpha \cup \gamma \rightarrow \beta \cup \gamma$ gilt auch in R.

Armstrong-Kalkül – Korrektheit

- Korrektheit der Transitivitätsregel (A_3): Für $\alpha \to \beta, \beta \to \gamma$ gilt auch $\alpha \to \gamma$
  - Seien $R \in \mathrm{Sat}(F)$ eine gültige Relationenausprägung, $\alpha, \beta, \gamma \subseteq X$ und $\alpha \to \beta,\ \beta \to \gamma \in F$. Außerdem seien $t_1, t_2 \in R$ zwei beliebige Tupel mit $t_1[\alpha] = t_2[\alpha]$. Wegen $\alpha \to \beta$, gilt also auch $t_1[\beta] = t_2[\beta]$. Wegen $\beta \to \gamma$, gilt dann auch $t_1[\gamma] = t_2[\gamma]$. Insgesamt: $\alpha \to \gamma$ gilt auch in R.

Armstrong-Kalkül – Erweiterung

Es ist für den Herleitungsprozess komfortabel, weitere Regeln hinzuzunehmen

- Erweiterung der Armstrong-Axiome um drei Regeln:

  Seien $\alpha, \beta, \gamma, \delta \subseteq X$ Attributmengen.

  – Vereinigung (A_4): Gelten $\alpha \to \beta$ und $\alpha \to \gamma$, so gilt auch $\alpha \to \beta\gamma$.

  – Dekomposition (A_5): Falls $\alpha \to \beta\gamma$ gilt, so gelten auch $\alpha \to \beta$ und $\alpha \to \gamma$.

  – Pseudotransitivität (A_6): Falls $\alpha \to \beta$ und $\beta\gamma \to \delta$ gilt, so gilt auch $\alpha\gamma \to \delta$.

Armstrong-Kalkül – Erweiterung

- Ableitung der Vereinigungsregel (A_4) :Gelten $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$, so gilt auch $\alpha \rightarrow \beta\gamma$.
  - Seien $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma \in F$. Über die Grundregeln erhalten wir

$$(A_2): \text{Da } \alpha \rightarrow \beta \text{ gilt } \alpha\gamma \rightarrow \beta\gamma$$

$$(A_2): \text{Da } \alpha \rightarrow \gamma \text{ gilt } \alpha \rightarrow \alpha\gamma$$

$$(A_3): \text{Da } \alpha \rightarrow \alpha\gamma \text{ und } \alpha\gamma \rightarrow \beta\gamma \text{ gilt } \alpha \rightarrow \beta\gamma$$

Armstrong-Kalkül – Erweiterung

- Ableitung der Dekompositionsregel ($A_{5)}$):

  Falls $\alpha \rightarrow \beta\gamma$ gilt, so gelten auch $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$.

  – Sei $\alpha \rightarrow \beta\gamma \in F$. Über die Grundregeln erhalten wir

$$(A_1) : \text{Da } \beta\gamma \coloneqq \beta \cup \gamma \text{ gilt } \beta\gamma \rightarrow \beta$$

$$(A_1) : \text{Da } \beta\gamma \coloneqq \beta \cup \gamma \text{ gilt } \beta\gamma \rightarrow \gamma$$

$$(A_3) : \text{Da } \alpha \rightarrow \beta\gamma \text{ und } \beta\gamma \rightarrow \beta \text{ gilt } \alpha \rightarrow \beta$$

$$(A_3) : \text{Da } \alpha \rightarrow \beta\gamma \text{ und } \beta\gamma \rightarrow \gamma \text{ gilt } \alpha \rightarrow \gamma$$

Armstrong-Kalkül – Erweiterung

- Ableitung der Pseudotransitivitätsregel (A_6) :

  Falls $\alpha \to \beta$ und $\beta\gamma \to \delta$ gilt, so gilt auch $\alpha\gamma \to \delta$

  Sei $\alpha \to \beta$ und $\beta\gamma \to \delta \in F$. Über die Grundregeln erhalten wir

  $$(A_2) : \text{Da } \alpha \to \beta \text{ gilt } \alpha\gamma \to \beta\gamma$$

  $$(A_3) : \text{Da } \alpha\gamma \to \beta\gamma \text{ und } \beta\gamma \to \delta \text{ gilt } \alpha\gamma \to \delta$$

Armstrong-Kalkül – Vereinigung

ProfessorenAdressen(PersNr, Name, Rang, Raum, Ort, Straße, Hausnummer, PLZ, Vorwahl, Bundesland)

- Funktionale Abhängigkeiten F:
  - Ort, Bundesland → Vorwahl
  - Ort, Bundesland, Straße, Hausnummer → PLZ
  - …

- Beispiel: f = Ort, Bundesland, Straße, Hausnummer → PLZ, Vorwahl ist ebenfalls aus F ableitbar
  - $(A_1) : f_1 = (\text{Ort, Bundesland, Straße, Hausnummer} \rightarrow \text{Ort, Bundesland})$
  - $(A_3) : f_2 = (\text{Ort, Bundesland, Straße, Hausnummer} \rightarrow \text{Vorwahl})$
  - $(\boldsymbol{A_4}) : \boldsymbol{f = f_3} = (\textbf{Ort, Bundesland, Strasse, Hausnummer} \rightarrow \textbf{PLZ, Vorwahl})$

Attributhülle

- Oft ist man nicht an der gesamten Hülle F^+ interessiert:
  - Welche Attribute sind unter einer gegebenen Menge von FDs F von einer bestimmten Attributmenge α funktional bestimmt?
  - Man nennt α^+ die Attributhülle von α unter F

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \rightarrow \beta \in F^+ \text{ mit } x \in \beta \}$$

  - Algorithmus zur Bestimmung von $\text{AttrHülle}(F, \alpha)$:
    - $Erg := \alpha$

      while (Änderungen an Erg) **do**

        foreach FD $\beta \rightarrow \gamma$ **in** F **do**

          if $\beta \subseteq Erg$ **then** $Erg := Erg \cup \gamma$;

      Ausgabe $\alpha^+ = Erg$;

$\kappa \subseteq X$ ist genau dann ein Superschlüssel, falls gilt:
$\kappa^+ = X$

Attributhülle – Beispiel

- Attributhülle

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \to \beta \in F^+ \text{ mit } x \in \beta \}$$

- Beispiel:

$$F = \{A \to C, B \to A, AB \to C\}$$

  - $\text{AttrHülle}(\text{F}, \{\text{B}\})$:

    $Erg = \{B\}$

    Durchlaufe F:

    $A \to C$: $\{B\}$, $B \to A$: $\{A, B\}$, $AB \to C$: $\{A, B, C\}$

    Durchlaufe F nochmal:

    keine Änderung

    Ausgabe $B^+ = \{A, B, C\}$

Attributhülle

$$\alpha^+ = \{ x \mid \text{es gibt } \alpha \to \beta \in F^+ \text{ mit } x \in \beta \}$$

- Lemma **L1**: Die Attributhülle besitzt folgende Eigenschaft.

  Für jede Teilmenge $V \subseteq \alpha^+$ gilt auch $\alpha \to V \in F^+$.

- Beweis (Skizze):

  Wir beschränken uns auf den Fall $V = \{a_1, a_2\}$. Da $V \subseteq \alpha^+$ ist, gibt es $\beta_1 = \{a_1, \dots\}$ und $\beta_2 = \{a_2, \dots\}$ mit $\alpha \to \beta_1,\ \alpha \to \beta_2 \in F^+$. Mit der Reflexivitätsregel (A_1) sind auch $\beta_1 \to \{a_1\}, \beta_2 \to \{a_2\} \in F^+$. Mit der Transitivitätsregel (A_3) und

  $$\alpha \to \beta_1, \qquad \beta_1 \to \{a_1\} \alpha \to \beta_2, \qquad \beta_2 \to \{a_2\}$$

  sind auch $\alpha \to \{a_1\}, \alpha \to \{a_2\} \in F^+$. Mit der Vereinigungsregel (A_4) erhalten wir

  $$\alpha \to \{a_1, a_2\} \in F^+.$$

Armstrong-Kalkül – Vollständigkeit

- Das Armstrong-Kalkül ist vollständig, d.h. jede von funktionalen Abhängigkeiten F implizierte FD f lässt sich mithilfe des Armstrong-Kalküls ableiten. Es gilt also

$$F \models f \Rightarrow F \vdash f$$

- Beweis (durch Kontraposition):
  - Wir zeigen die Aussage $\mathrm{F} \nvdash f \Rightarrow F \not\models f$. D.h. ist eine funktionale Abhängigkeit f nicht ableitbar, so wird sie auch nicht von F impliziert. Sei dazu $f = \alpha \rightarrow \beta$ eine FD mit $F \nvdash f$ und X die Menge aller Attribute aus F und f.
  - Um zu zeigen, dass $F \not\models f$ gilt, konstruieren wir eine Relation R, in der F gilt, aber f nicht.

    <u>Konstruktion:</u>

    Seien $\alpha^+ = \{a_1, \ldots, a_n\}$ und $X \setminus \alpha^+ = \{b_1, \ldots, \mathrm{b}_m\}$.

    Konstruiere R wie rechts.

R	$a_1, \ldots, a_n$	$b_1, \ldots, b_m$
r	$1, \ldots, 1$	$1, \ldots, 1$
t	$1, \ldots, 1$	$0, \ldots, 0$

Armstrong-Kalkül – Vollständigkeit

- Zeige für Vollständigkeit: $F \nvdash f \Rightarrow F \nvDash f$
- Hilfslemma 1:

  Es gilt $R \vDash F$, d.h. alle FDs in F werden von R erfüllt.
- Beweis (durch Widerspruch):

  Angenommen es existiert $V \rightarrow W \in F$ mit $R \nvDash V \rightarrow W$. Dann muss für die beiden Tupel $r, t \in R$ gelten, dass $r[V] = t[V]$ und $r[W] \neq t[W]$ ist. Nach Konstruktion von R kann $r[V] = t[V]$ nur gelten, wenn $V \subseteq \{a_1, \ldots, a_n\} = \alpha^+$ ist. Ebenso impliziert $r[W] \neq t[W]$, dass $W \not\subseteq \alpha^+$ ist.

  Nach dem vorigen Lemma **L1** folgt aus $V \subseteq \alpha^+$, dass $\alpha \rightarrow V \in F^+$ ist. Mit der Transivititätsregel (A_3) erhalten wir aus $\alpha \rightarrow V,\ V \rightarrow W \in F^+$, dass auch $\alpha \rightarrow W \in F^+$ ist.

  Dies liefert einen Widerspruch zu $W \not\subseteq \alpha^+$.

Armstrong-Kalkül – Vollständigkeit

- Zeige für Vollständigkeit: $F \nvdash f \Rightarrow F \nvDash f$
- Hilfslemma 2:

  Es gilt $R \nvDash f$, d.h. f wird von R nicht erfüllt.
- Beweis:

  Da f nicht aus F ableitbar ist $(F \nvdash f)$ gilt insbesondere $\beta \not\subseteq \alpha^+$. Also existiert ein $b_k \in \beta$

  mit $b_k \in X \setminus \alpha^+$. Direkt aus der Konstruktion von R folgt dann, dass $r[\alpha] = t[\alpha]$ ist, aber $r[b_k] = 1 \neq 0 = t[b_k]$.

  Insgesamt erhalten wir, dass $f = \alpha \rightarrow \beta$ von R nicht erfüllt wird.
- Es gilt also $R \vDash F$, aber $R \nvDash f$. Darauf folgt $F \nvDash f$.

Kanonische Überdeckung – Motivation

- Um für zwei Mengen F und G zu entscheiden, ob sie äquivalent sind ($\mathrm{Sat}(F) = \mathrm{Sat}(G)$), reicht es $F^+ = G^+$ zu überprüfen.

Warum?
Freiwillige Übung

- Im Allgemeinen ist die Hülle F^+ einer Menge von FDs sehr groß
- Vor allem bei Datenbankmodifikationen:
  - Überprüfen der Konsistenz anhand von F^+ sehr aufwändig (auch viele triviale Abhängigkeiten)
  - minimale Menge von „erzeugenden“ funktionalen Abhängigkeiten wünschenswert

- Statt der Hülle F^+: kanonische Überdeckung

Kanonische Überdeckung (Definition)

- Sei F eine Menge von funktionalen Abhängigkeiten über einer Attributmenge X. Dann heißt F_c eine *kanonische Überdeckung* von F, falls gilt:
  1. $F_c^+ = F^+$, d.h. $\mathrm{Sat}(F_c) = \mathrm{Sat}(F)$ (äquivalent)
  2. In F_c existieren keine FDs $\alpha \rightarrow \beta$ bei denen α oder β überflüssige Attribute enthalten. D.h.
     - Für alle $A \in \alpha$: $\big(F_c \setminus (\alpha \rightarrow \beta)\big) \cup (\alpha \setminus A \rightarrow \beta) \not\equiv F_c$ (nicht äquivalent)
     - Für alle $B \in \beta$: $\big(F_c \setminus (\alpha \rightarrow \beta)\big) \cup (\alpha \rightarrow \beta \setminus B) \not\equiv F_c$
  3. Jede linke Seite einer FD ist einzigartig in F_c (sonst ersetze durch Vereinigungen)

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

  Eine kanonische Überdeckung ist $\mathrm{F_c} = \{A \rightarrow C, B \rightarrow A\}$.
  (Algorithmus: nächste Folie)

Kanonische Überdeckung – Algorithmus

- Eingabe: Menge von FDs F, Ausgabe: Kanonische Überdeckung F_c

1. Setze $F_c = F$
2. Führe für jede FD $\alpha \rightarrow \beta \in F_c$ eine Linksreduktion durch:
   - Überprüfe für alle $A \in \alpha$, ob A überflüssig ist. D.h. ob gilt: $\beta \subseteq \text{AttrHülle}(F_c, \alpha \setminus A)$

     Falls ja: ersetze $\alpha \rightarrow \beta$ durch $\alpha \setminus A \rightarrow \beta$.
3. Führe für jede FD $\alpha \rightarrow \beta \in F_c$ eine Rechtsreduktion durch:
   - Überprüfe für alle $B \in \beta$, ob B überflüssig ist. D.h. ob gilt:

     $$B \in \text{AttrHülle}\big(\big(F_c \setminus (\alpha \rightarrow \beta)\big) \cup (\alpha \rightarrow \beta \setminus B), \alpha\big)$$

     Falls ja: ersetze $\alpha \rightarrow \beta$ durch $\alpha \rightarrow \beta \setminus B$.
4. Entferne die im 3. Schritt entstandenen FDs der Form $\alpha \rightarrow \emptyset$
5. Fasse über die Vereinigungsregel FDs der Form $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$

   $$\alpha \rightarrow \beta_1 \cup \cdots \cup \beta_n$$

Kanonische Überdeckung – Beispiel

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

1. Linksreduktion:
   - $A \rightarrow C$: C ist nicht in $(\{A\} \setminus \{A\})^+ = \emptyset^+$. Ok
   - $B \rightarrow A$: Ok
   - $AB \rightarrow C$: Überprüfe A. Ist $C \in (\{A, B\} \setminus \{A\})^+$? Ja, denn

$$B^+ = \{ \quad \}$$

   Ersetze $AB \rightarrow C$ durch $B \rightarrow C$.

   B muss nun in $B \rightarrow C$ nicht mehr überprüft werden.

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow C\}$$

Kanonische Überdeckung – Beispiel

- Beispiel:

$$F = \{A \rightarrow C, B \rightarrow A, AB \rightarrow C\}$$

1. Linksreduktion:
   - $A \rightarrow C$: C ist nicht in $(\{A\} \setminus \{A\})^+ = \emptyset^+$. Ok
   - $B \rightarrow A$: Ok
   - $AB \rightarrow C$: Überprüfe A. Ist $C \in (\{A, B\} \setminus \{A\})^+$? Ja, denn

$$B^+ = \{B, A, C\}$$

   Ersetze $AB \rightarrow C$ durch $B \rightarrow C$.

   B muss nun in $B \rightarrow C$ nicht mehr überprüft werden.

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow C\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\}, A)$

     $= \{$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\}, B)$

     $= \{$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\}, A)$

     $= \{A\}$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\}, B)$

     $= \{$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to C\}$$

2. Rechtsreduktion:
   - $A \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to \emptyset, B \to A, B \to C\},\ A)$

$$= \{A\}$$

     Also ist C rechts nicht überflüssig.
   - $B \to A$: Analog: A ist rechts nicht überflüssig.
   - $B \to C$: Überprüfe C. $\text{AttrHülle}(\{A \to C, B \to A, B \to \emptyset\},\ B)$

$$= \{B, A, C\}$$

     Also ist C auf der rechten Seite überflüssig. Ersetze $B \to C$ durch $B \to \emptyset$.

- Neues Zwischenergebnis:

$$F_c = \{A \to C, B \to A, B \to \emptyset\}$$

Kanonische Überdeckung – Beispiel

- Zwischenergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A, B \rightarrow \emptyset\}$$

3. Entferne $\alpha \rightarrow \emptyset$:
   - Entferne die Abhängigkeit $B \rightarrow \emptyset$
4. Vereinigen:
   - Hier ist nichts zu tun.

- Ergebnis:

$$F_c = \{A \rightarrow C, B \rightarrow A\}$$

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. Armstrong-Kalkül
3. **Zerlegung von Relationen**
4. Normalformen und Normalisierungen

Zerlegung

- funktionale Abhängigkeiten: mächtiges Werkzeug um Konsistenzbedingungen zu modellieren
- In der Sprache der FDs kann nun ausgedrückt werden, welche Schemata „gut“ und welche „schlecht“ sind (siehe „Normalformen und Normalisierungen“)
- „schlechte“ Schemata können durch Zerlegung/Dekomposition in die Normalformen überführt werden.
- Welche Anomalien können bei einem schlechtem Relationenschema auftreten?

Zerlegung

- nicht zusammenpassende Informationen
- Basis jeder Normalisierung:

  Zerlege das Relationenschema $\mathcal{R}$ in Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$

- Kriterien für eine korrekte Zerlegung?

  1. Verlustlosigkeit:
     - Die Informationen einer Relationsausprägung R von $\mathcal{R}$ müssen aus den resultierenden Ausprägungen $R_1, \ldots, R_n$ wieder komplett rekonstruiert werden können.
  2. Abhängigkeitserhaltung:
     - Die für $\mathcal{R}$ geltenden funktionalen Abhängigkeiten müssen sich auf $\mathcal{R}_1, \ldots, \mathcal{R}_n$ übertragen lassen

  (Formalisierung auf den nächsten Folien)

Gültig, Verlustlos

Betrachte die Zerlegung von $\mathcal{R}$ mit FDs F in $\mathcal{R}_1, \mathcal{R}_2$.
Seien X, X_1 und X_2 die entsprechenden Attributmengen.

- *Gültigkeit*:

  Die Zerlegung von $\mathcal{R}$ heißt *gültig*, falls gilt

  $$X = X_1 \cup X_2.$$

- *Verlustlosigkeit*:

  Sei $R \in \mathrm{Sat}(F)$ eine beliebige gültige Ausprägung von $\mathcal{R}$. Definiere

  $$R_1 \coloneqq \Pi_{X_1}(R), \qquad R_2 \coloneqq \Pi_{X_2}(R).$$

  Die Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1, \mathcal{R}_2$ heißt *verlustlos*, falls für alle $R \in \mathrm{Sat}(F)$ gilt

  $$R = R_1 \bowtie R_2$$

Verlustlos – Beispiel

- Verlust von Informationen:

  Relationsschema $\text{Biertrinker}(\text{Kneipe}, \text{Gast}, \text{Bier})$ mit FD $\{\text{Kneipe}, \text{Gast} \rightarrow \text{Bier}\}$

Biertrinker		
Kneipe	Gast	Bier
Domkeller	Kemper	Pils
Domkeller	Eickler	Hefeweizen
Die Kiste	Eickler	Pils

  (Kneipe, Gast und Bier werden eindeutig durch ihren Namen bestimmt)

  – Welches Getränk ein Gast trinkt hängt von der besuchten Kneipe ab

  – Zerlegung in $\text{Besucht}(\text{Kneipe}, \text{Gast})$, $\text{Trinkt}(\text{Gast}, \text{Bier})$

Besucht	
Kneipe	Gast
Domkeller	Kemper
Domkeller	Eickler
Die Kiste	Eickler

Trinkt	
Gast	Bier
Kemper	Pils
Eickler	Hefeweizen
Eickler	Pils

Nicht verlustlose Zerlegung – ein Beispiel

Besucht	
Kneipe	Gast
Domkeller	Kemper
Domkeller	Eickler
Die Kiste	Eickler

Trinkt	
Gast	Bier
Kemper	Pils
Eickler	Hefeweizen
Eickler	Pils

Biertrinker ≠ Besucht ⋈ Trinkt

Besucht ⋈ Trinkt		
Kneipe	Gast	Bier
Domkeller	Kemper	Pils
Domkeller	*Eickler*	*Hefeweizen*
Domkeller	Eickler	Pils
Die Kiste	*Eickler*	*Hefeweizen*
Die Kiste	Eickler	Pils

Ein Mehr an Tupeln bedeutet einen Verlust an Informationen!

Kriterien für Verlustlosigkeit

- Finde Bedingungen unter denen eine verlustlose Zerlegung garantiert ist

  Seien $\mathcal{R}, \mathcal{R}_1, \mathcal{R}_2$ Relationenschemata mit den Attributmengen X, X_1 und X_2 wie vorher definiert.

- Hinreichend:

  Die Zerlegung von $\mathcal{R}$ ist verlustlos, falls gilt

$$(X_1 \cap X_2) \rightarrow X_1 \in F^+ \quad \text{oder} \quad (X_1 \cap X_2) \rightarrow X_2 \in F^+$$

- In anderen Worten:

  Die Zerlegung von $\mathcal{R}$ ist verlustlos, falls gilt

$$X_1 \subseteq (X_1 \cap X_2)^+ \quad \text{oder} \quad X_2 \subseteq (X_1 \cap X_2)^+$$

- Intutition: Joinattribut bestimmt eines der beiden Teilschemata

- Im Biertrinker-Beispiel: einzige nicht-triviale Abhängigkeit war

$$\text{Kneipe}, \text{Gast} \rightarrow \text{Bier}$$

Verlustlos – Beispiel

- Verlustlose Zerlegung:

Relationsschema $\text{Eltern}(\text{Vater}, \text{Mutter}, \text{Kind})$ mit FDs $\{\text{Kind} \rightarrow \text{Vater}, \text{Kind} \rightarrow \text{Mutter}\}$

(biol.) Eltern		
Vater	Mutter	Kind
Johann	Martha	Else
Johann	Maria	Theo
Heinz	Martha	Cleo

(Personen sind durch ihren Vornamen eindeutig bestimmt)

Verlustlose Zerlegung

$\text{Väter}(\text{Vater}, \text{Kind})$

$\text{Mütter}(\text{Mutter}, \text{Kind})$

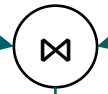
Väter	
Vater	Kind
Johann	Else
Johann	Theo
Heinz	Cleo

Mütter	
Mutter	Kind
Martha	Else
Maria	Theo
Martha	Cleo

Verlustlos – Beispiel

Väter	
Vater	Kind
Johann	Else
Johann	Theo
Heinz	Cleo

Mütter	
Mutter	Kind
Martha	Else
Maria	Theo
Martha	Cleo



$\text{Kind} \rightarrow \text{Vater},$
$\text{Kind} \rightarrow \text{Mutter}$

Väter ⋈ Mütter		
Vater	Mutter	Kind
Johann	Martha	Else
Johann	Maria	Theo
Heinz	Martha	Cleo

Abhängigkeitserhaltung

- Gegeben Zerlegung von $\mathcal{R}$ mit FDs F in die Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ mit entsprechenden Attributmengen $X, X_1, \ldots, X_n$

  Die Überprüfung der Konsistenzbedinungen bei Einfügen, Löschen und Updaten auf

  $$\mathcal{R}_1 \bowtie \cdots \bowtie \mathcal{R}_n$$

  benötigt Joins, falls funktionale Abhängigkeiten sich nach der Zerlegung auf Attribute verschiedenen Relationenschemata beziehen (Überprüfung bei jeder Transaktion). Ineffzient!
- Wunsch: alle FDs, die für das Schema $\mathcal{R}$ gelten, sollen *lokal* (ohne Joins) auf den einzelnen Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ überprüfbar sein.
- Formal*: Abhängigkeitserhaltend*:

  Sei $F_{\mathcal{R}_i}$ die Menge der FDs $\alpha \to \beta \in F^+$ mit $\alpha, \beta \subseteq X_i$, (also deren Attribute aus X_i sind) .

  Die Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1, \ldots, \mathcal{R}_n$ heißt *abhängigkeitserhaltend*, falls $F^+ = \left(F_{\mathcal{R}_1} \cup \cdots \cup F_{\mathcal{R}_n}\right)^+$

  Diese Eigenschaft wird auch *Hüllentreue* der Zerlegung genannt.

Abhängigkeitserhaltung – Beispiel

- Beispiel: Verlustlose aber nicht abhängigkeitserhaltende Zerlegung

  Betrachte das Relationsschema

  $$\text{PLZVerzeichnis}(\text{BLand}, \text{Ort}, \text{Straße}, \text{PLZ})$$

  – Annahme:
    - Ortsnamen sind innerhalb eines Bundeslandes eindeutig
    - PLZ'n ändern sich nicht innerhalb einer Straße
  – Funktionale Abhängigkeiten:
    - $\text{PLZ} \rightarrow \text{Ort}, \text{BLand}$
    - $\text{BLand}, \text{Ort}, \text{Straße} \rightarrow \text{PLZ}$
  – Betrachte die Zerlegung in

  $$\text{Straßen}(\text{PLZ}, \text{Straße})$$

  $$\text{Orte}(\text{PLZ}, \text{Ort}, \text{BLand})$$

Abhängigkeitserhaltung – Beispiel

PLZVerzeichnis			
Ort	BLand	Straße	PLZ
Frankfurt	Hessen	Goethestraße	60313
Frankfurt	Hessen	Galgenstraße	60437
Frankfurt	Brandenburg	Goethestraße	15234

Straßen	
PLZ	Straße
60313	Goethestraße
60437	Galgenstraße
15234	Goethestraße

Orte		
Ort	Bland	PLZ
Frankfurt	Hessen	60313
Frankfurt	Hessen	60437
Frankfurt	Brandenburg	15234

Abhängigkeitserhaltung – Beispiel

PLZVerzeichnis			
Ort	BLand	Straße	PLZ
Frankfurt	Hessen	Goethestraße	60313
Frankfurt	Hessen	Galgenstraße	60437
Frankfurt	Brandenburg	Goethestraße	15234
Frankfurt	*Brandenburg*	*Goethestraße*	*15235*

Straßen	
PLZ	Straße
60313	Goethestraße
60437	Galgenstraße
15234	Goethestraße
15235	*Goethestraße*

Orte		
Ort	Bland	PLZ
Frankfurt	Hessen	60313
Frankfurt	Hessen	60437
Frankfurt	Brandenburg	15234
Frankfurt	*Brandenburg*	*15235*

Siehe funktionale Abhängigkeit: BLand, Ort, Straße→PLZ

Ungültiger Eintrag kann nur durch $\mathrm{PLZVerzeichnis} = \mathrm{Straßen} \bowtie \mathrm{Orte}$ überprüft werden

Abhängigkeitserhaltung – Beispiel

- Zerlegung von $\mathrm{PLZVerzeichnis}(\mathrm{BLand}, \mathrm{Ort}, \mathrm{Straße}, \mathrm{PLZ})$ in

$$\mathrm{Straßen}(\mathrm{PLZ}, \mathrm{Straße}), \qquad \mathrm{Orte}(\mathrm{PLZ}, \mathrm{Ort}, \mathrm{BLand})$$

- Also:
  - Die Zerlegung ist verlustlos, da $\{\mathrm{PLZ}, \mathrm{Straße}\} \cap \{\mathrm{Ort}.\mathrm{Straße}, \mathrm{PLZ}\} = \{\mathrm{PLZ}\}$ ist und die funktionale Abhängigkeit

$$\mathrm{PLZ} \rightarrow \mathrm{PLZ}, \mathrm{Ort}, \mathrm{BLand}$$

    gilt.

  - Die Zerlegung ist nicht abhängigkeitserhaltend, da die funktionale Abhängigkeit

$$\mathrm{Ort}, \mathrm{BLand}, \mathrm{Straße} \rightarrow \mathrm{PLZ}$$

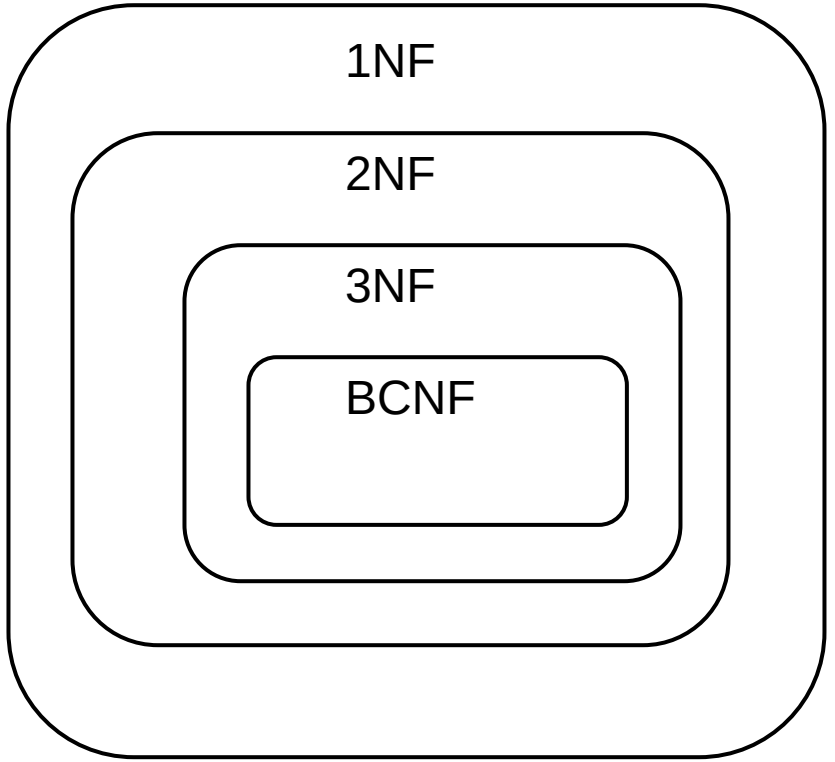
    keiner der neuen Relationen $\mathrm{Straßen}$ oder Orte zugeordnet werden kann.

6. Relationale Entwurfstheorie

1. Funktionale Abhängigkeiten
2. Armstrong-Kalkül
3. Zerlegung von Relationen
4. **Normalformen und Normalisierungen**

Normalformen und Normalisierungen

- Vermeidung von Redundanzen und damit zusammenhängender Anomalien
- Normalformen unterscheiden sich in der Strenge der Anforderungen an die funktionalen Abhängigkeiten
- Normalisierungsalgorithmen berechnen zu einem Schema eine verlustlose und, *wenn möglich*, auch abhängigkeitserhaltende Zerlegung



Erste Normalform

- Alle Attribute haben atomare Wertebereiche (zB String, Integer, …)
  - zusammengesetzte, mengenwertige oder relationenwertige Attribute sind nicht erlaubt
- Beispiel: nicht in 1NF

Eltern-1		
Vater	Mutter	Kind
Johann	Martha	{Else, Lucia}
Johann	Maria	{Theo, Jose}
Heinz	Martha	{Cleo}

in 1NF

Eltern-2		
Vater	Mutter	Kind
Johann	Martha	Else
Johann	Martha	Lucia
Johann	Maria	Theo
Johann	Maria	Jose
Heinz	Martha	Cleo

- Im Folgenden gehen wir stets von Relationen in 1NF aus

NF²-Modelle: non-first-normal-form Modelle

Zweite Normalform

- Betrachte wieder das Beispiel

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...

- Anschaulich ist die 2NF verletzt, wenn eine Relation Informationen aus mehreren Konzepten enthält (hier: $\mathrm{ProfVorl}$ entspricht $\mathrm{Professor} \bowtie \mathrm{Vorlesung}$)
- Attribute dürfen nicht von Teilmengen von Schlüsselkandidaten abhängen
- FDs: $\mathrm{PersNr}, \mathrm{VorlNr} \rightarrow \mathrm{PersNr}, \ldots, \mathrm{SWS}$

  $\mathrm{PersNr} \rightarrow \mathrm{Name}, \mathrm{Rang}, \mathrm{Raum}$

  $\mathrm{VorlNr} \rightarrow \mathrm{Titel}, \mathrm{SWS}$

Zweite Normalform (Definition)

- *Nichtschlüssel-Attribut (NSA)*:

  Seien $\kappa_1, \ldots, \kappa_m$ die Schlüsselkandidaten einer gegebenen Relation $\mathcal{R}$ und X die zugehörige Attributmenge. Dann heißt $X \setminus (\kappa_1 \cup \cdots \cup \kappa_m)$ die Menge aller *Nichtschlüssel-Attribute*.

- *Zweite Normalform*:

  Ein Relationenschema $\mathcal{R}$ mit FDs F ist in *zweiter Normalform*, falls für jedes Nichtschlüssel-Attribut A und jeden Schlüsselkandidaten κ_i gilt:

  - A ist voll funktional abhängig von κ_i, d.h. für kein Attribut $B \in \kappa_i$ gilt $\kappa_i \setminus \{B\} \rightarrow A \in F^+$

- Beispiel:
  - Einziger Schlüsselkandidat von ProfVorl ist $\kappa_1 = \{\mathrm{PersNr}, \mathrm{VorlNr}\}$. Es gilt aber $\mathrm{PersNr} \rightarrow \mathrm{Name} \in F^+$. Also ist Name nicht voll funktional abhängig von κ_1.

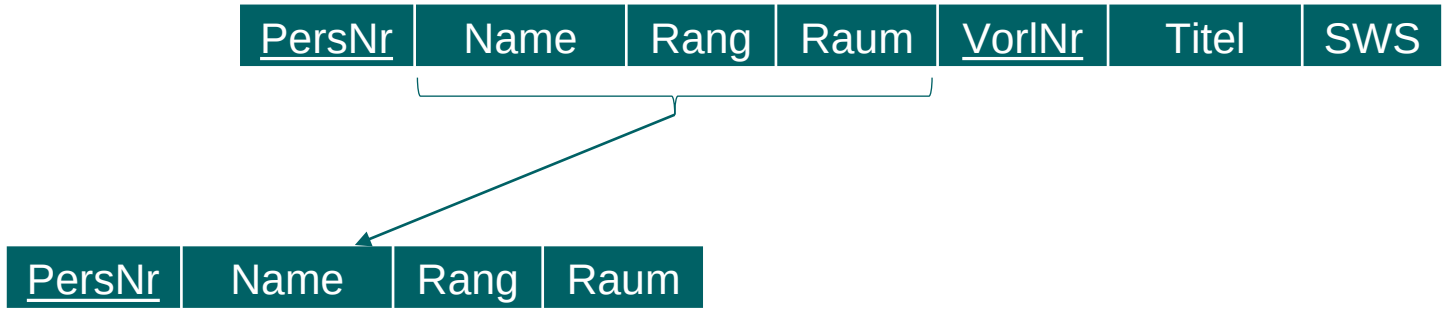
Zweite Normalform – Algorithmus (Skizze)

- Ein Relationenschema $\mathcal{R}$ kann wie folgt in zweite Normalform überführt werden
  a) Fasse alle Nichtschlüsselattribute, die nur von einem Teilschlüssel abhängen, mit diesem Teilschlüssel als Primärschlüssel in einer eigenen Relation zusammen. Alle Attribute, die von demselben Teilschlüssel abhängen, müssen in derselben Relation zusammengefasst werden
  b) Entferne die ausgelagerten Nichtschlüsselattribute aus der Ursprungsrelation und fasse die übriggebliebenen Attribute zu einer neuen Relation zusammen

PersNr	Name	Rang	Raum	VorlNr	Titel	SWS

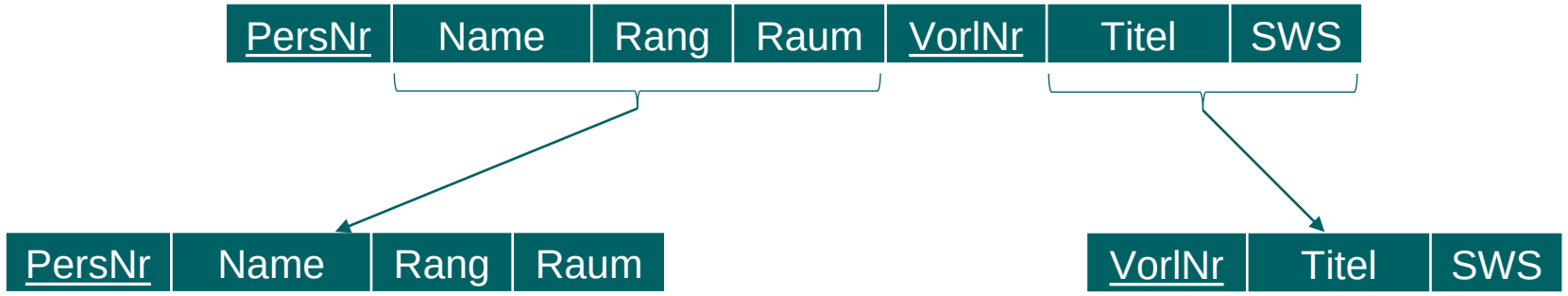
Zweite Normalform – Algorithmus (Skizze)

- Ein Relationenschema $\mathcal{R}$ kann wie folgt in zweite Normalform überführt werden
  a) Fasse alle Nichtschlüsselattribute, die nur von einem Teilschlüssel abhängen, mit diesem Teilschlüssel als Primärschlüssel in einer eigenen Relation zusammen. Alle Attribute, die von demselben Teilschlüssel abhängen, müssen in derselben Relation zusammengefasst werden
  b) Entferne die ausgelagerten Nichtschlüsselattribute aus der Ursprungsrelation und fasse die übriggebliebenen Attribute zu einer neuen Relation zusammen



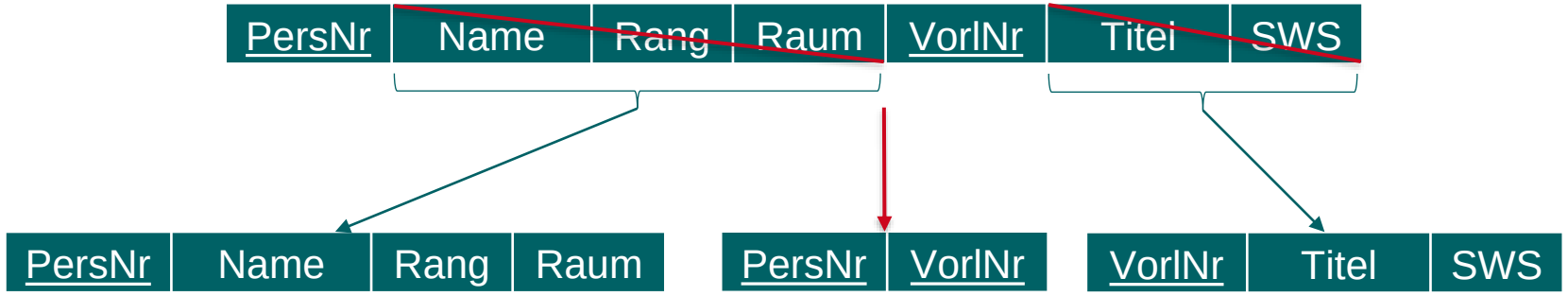
Zweite Normalform – Algorithmus (Skizze)

- Ein Relationenschema $\mathcal{R}$ kann wie folgt in zweite Normalform überführt werden
  a) Fasse alle Nichtschlüsselattribute, die nur von einem Teilschlüssel abhängen, mit diesem Teilschlüssel als Primärschlüssel in einer eigenen Relation zusammen. Alle Attribute, die von demselben Teilschlüssel abhängen, müssen in derselben Relation zusammengefasst werden
  b) Entferne die ausgelagerten Nichtschlüsselattribute aus der Ursprungsrelation und fasse die übriggebliebenen Attribute zu einer neuen Relation zusammen

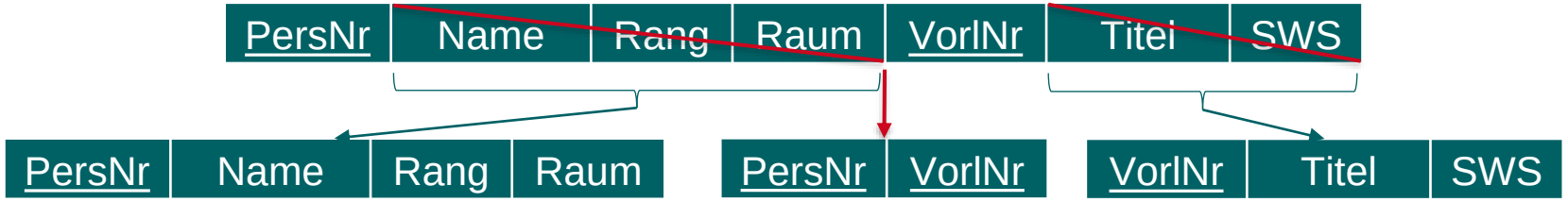


Zweite Normalform – Algorithmus (Skizze)

- Ein Relationenschema $\mathcal{R}$ kann wie folgt in zweite Normalform überführt werden
  a) Fasse alle Nichtschlüsselattribute, die nur von einem Teilschlüssel abhängen, mit diesem Teilschlüssel als Primärschlüssel in einer eigenen Relation zusammen. Alle Attribute, die von demselben Teilschlüssel abhängen, müssen in derselben Relation zusammengefasst werden
  b) Entferne die ausgelagerten Nichtschlüsselattribute aus der Ursprungsrelation und fasse die übriggebliebenen Attribute zu einer neuen Relation zusammen



Zweite Normalform – Beispiel



PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	W3	226	5041	Ethik	4
2125	Sokrates	W3	226	5049	Mäeutik	2
2125	Sokrates	W3	226	4052	Logik	4
...	...	...	...	...	...	...

PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
...	...	...	...

PersNr	VorlNr
2125	5041
2125	5049
2125	4052
...	...

VorlNr	Titel	SWS
5041	Ethik	4
5049	Mäeutik	2
4052	Logik	4
...	...	...

Dritte Normalform – Motivation

- Beispiel: Erweiterung des Universitäts-Beispiels um Adressen

$$\text{ProfessorenAdressen}(\ \text{PersNr}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{PLZ}, \text{Vorwahl}, \text{BLand}\ )$$

ProfessorenAdressen						
PersNr	Name	...	Ort	...	PLZ	...
2125	Sokrates	...	Aachen	...	52052	...
2132	Popper	...	Aachen	...	52052	...
...	...	...	...	...	...	...

- Diese Relation ist in zweiter Normalform
- Die Information, dass die PLZ 52052 zu Aachen gehört ist hier dennoch redundant
- Ursache: funktionale Abhängigkeit $\text{PLZ} \rightarrow \text{Ort}$ zwischen NSAs

Dritte Normalform (Definition)

- *Dritte Normalform*:

  Ein Relationenschema $\mathcal{R}$ mit FDs F ist in *dritter Normalform*, falls für jedes Nichtschlüssel-Attribut B und jede nicht-triviale Abhängigkeit $\alpha \rightarrow B \in \mathrm{F}^+$ gilt:

  - α ist ein Superschlüssel (d.h. nicht notwendig minimal)

- Die zweite Normalform lässt auch Abhängigkeiten zwischen NSAs zu
  - In dritter Normalform dürfen NSAs im Endeffekt nur von Schlüsselkandidaten abhängen

- Beispiel:

$$\mathrm{ProfessorenAdressen}(\ \mathrm{PersNr}, \mathrm{Name}, \mathrm{Rang}, \mathrm{Raum},\ \mathrm{Ort}, \mathrm{Straße}, \mathrm{Hausnr}, \mathrm{PLZ}, \mathrm{Vorwahl}, \mathrm{BLand}\ )$$

  - Sei X die Attributmenge von $\mathrm{ProfessorenAdressen}$. Es gelten die FDs:

$$\mathrm{PersNr} \rightarrow X, \qquad \mathrm{Raum} \rightarrow \mathrm{PersNr}, \qquad \mathrm{Raum} \rightarrow X,$$

$$\mathrm{Ort}, \mathrm{BLand} \rightarrow \mathrm{Vorwahl}, \qquad \mathrm{Ort}, \mathrm{BLand}, \mathrm{Straße}, \mathrm{Hausnr} \rightarrow \mathrm{PLZ}, \qquad \ldots$$

nicht erlaubt!

Dritte Normalform – Synthesealgorithmus

- Synthesealgorithmus:

  Eingabe: Relationenschema $\mathcal{R}$ mit FDs F und Attributmenge X

  Ausgabe: verlustlose, abhängigkeitserhaltende 3NF-Zerlegung $\mathcal{R}_1, \ldots, \mathcal{R}_n$

  1. Bestimme die kanonische Überdeckung F_c zu F

     Zur Wiederholung:

     a) Linksreduktion der FDs

     b) Rechtsreduktion der FDs

     c) Entfernung der FDs der Form $\alpha \rightarrow \emptyset$

     d) Zusammenfassung von FDs mit gleichen linken Seiten

Dritte Normalform – Synthesealgorithmus

- Synthesealgorithmus:

  Eingabe: Relationenschema $\mathcal{R}$ mit FDs F und Attributmenge X

  Ausgabe: verlustlose, abhängigkeitserhaltende 3NF-Zerlegung $\mathcal{R}_1, \ldots, \mathcal{R}_n$

  1. Bestimme die kanonische Überdeckung F_c zu F
  2. Für jede funktionale Abhängigkeit $\alpha \rightarrow \beta \in F_c$:
     - Erstelle Relationenschema $\mathcal{R}_\alpha$ mit Attributmenge $\alpha \cup \beta$
     - Ordne $\mathcal{R}_\alpha$ die FDs $F_\alpha = \{\, \alpha' \rightarrow \beta' \in F_c \mid \alpha', \beta' \subseteq \alpha \cup \beta \,\}$ zu
  3. Falls ein in Schritt 2 erzeugtes Schema $\mathcal{R}_\alpha$ einen Schlüsselkandidaten von $\mathcal{R}$ enthält:
     fertig mit Schritt 3

     sonst: wähle einen Schlüsselkandidaten κ von $\mathcal{R}$ aus und erstelle das zusätzliche Schema $\mathcal{R}_\kappa$ mit funktionaler Abhängigkeit $F_\kappa = \{\kappa \rightarrow \kappa\}$
  4. Eliminiere jedes Schema $\mathcal{R}_\alpha$ dessen Attributmenge in der eines größeren Schemas $\mathcal{R}_{\alpha'}$ enthalten ist

Dritte Normalform – Synthesealgorithmus (Beispiel)

- Beispiel:

  ProfessorenAdressen(PersNr, Name, Rang, Raum,
  Ort, Straße, Hausnr, PLZ, Vorwahl, BLand)

- Funktionale Abhängigkeiten:
  - PersNr → PersNr, Name, Rang, Raum, Ort, Straße, Hausnr, PLZ, Vorwahl, Bland
  - PersNr → Raum
  - Raum → PersNr
  - Ort, Straße, Hausnr, BLand → PLZ
  - Ort, BLand → Vorwahl
  - PLZ → Ort, BLand

Dritte Normalform – Synthesealgorithmus (Beispiel)

- Beispiel:

$$\text{ProfessorenAdressen}(\underline{\text{PersNr}}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{PLZ}, \text{Vorwahl}, \text{BLand})$$

1. die kanonische Überdeckung F_c enthält die FDs

$f_1 = \text{PersNr} \rightarrow \text{Raum}, \text{Name}, \text{Rang}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{Bland}$

$f_2 = \text{Raum} \rightarrow \text{PersNr}, \qquad f_3 = \text{Ort}, \text{Straße}, \text{Hausnr}, \text{BLand} \rightarrow \text{PLZ}$

$f_4 = \text{Ort}, \text{BLand} \rightarrow \text{Vorwahl}, \qquad f_5 = \text{PLZ} \rightarrow \text{Ort}, \text{Bland}$

2. erzeugte Relationenschemata:

aus f_1: $\text{Professor}(\text{PersNr}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{BLand})$ mit $F_1 = \{f_1, f_2\}$

aus f_2: $\mathcal{R}_2(\text{Raum}, \text{PersNr})$ mit $F_2 = \{f_2\}$

aus f_3: $\text{PLZverzeichnis}(\text{Ort}, \text{Straße}, \text{Hausnr}, \text{BLand}, \text{PLZ})$ mit $F_3 = \{f_3, f_5\}$

aus f_4: $\text{Vorwahlverzeichnis}(\text{Ort}, \text{BLand}, \text{Vorwahl})$ mit $F_4 = \{f_4\}$

aus f_5: $\mathcal{R}_5(\text{PLZ}, \text{Ort}, \text{BLand})$ mit $F_5 = \{f_5\}$

Dritte Normalform – Synthesealgorithmus (Beispiel)

- Beispiel:

$$\text{ProfessorenAdressen}(\underline{\text{PersNr}}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{PLZ}, \text{Vorwahl}, \text{BLand})$$

2. erzeugte Relationenschemata:

   aus f_1: $\text{Professor}(\text{PersNr}, \text{Name}, \text{Rang}, \text{Raum}, \text{Ort}, \text{Straße}, \text{Hausnr}, \text{BLand})$ mit $F_1 = \{f_1, f_2\}$

   aus f_2: $\mathcal{R}_2(\text{Raum}, \text{PersNr})$ mit $F_2 = \{f_2\}$

   aus f_3: $\text{PLZverzeichnis}(\text{Ort}, \text{Straße}, \text{Hausnr}, \text{BLand}, \text{PLZ})$ mit $F_3 = \{f_3, f_5\}$

   aus f_4: $\text{Vorwahlverzeichnis}(\text{Ort}, \text{BLand}, \text{Vorwahl})$ mit $F_4 = \{f_4\}$

   aus f_5: $\mathcal{R}_5(\text{PLZ}, \text{Ort}, \text{BLand})$ mit $F_5 = \{f_5\}$

3. PersNr ist ein Schlüsselkandidat des gesamten Schemas $\text{ProfessorenAdressen}$

   ⇒ Es muss keine neue Relation erzeugt werden

4. $\mathcal{R}_2$ und $\mathcal{R}_5$ werden entfernt, da sie in Professor bzw. PLZverzeichnis enthalten sind

Boyce-Codd Normalform

- Ziel: Jede Information wird genau einmal gespeichert
- *BCNF*:

  Ein Relationsschema $\mathcal{R}$ mit FDs F ist genau dann in *Boyce-Codd Normalform*, wenn für alle nicht-trivialen FDs $\alpha \rightarrow \beta \in F^+$ gilt:

  - α ist ein Superschlüssel von $\mathcal{R}$

- Beispiel: In dritter Normalform aber nicht in BCNF

  - zwei Schlüsselkandidaten

$$\kappa_1 = \{\, \text{Ort}, \text{BLand} \,\}, \qquad \kappa_2 = \{\, \text{Ort}, \text{MinisterpräsidentIn} \}$$

  - ist in 3NF: einziges NSA ist EW. EW kommt nur in $\mathrm{f}_1 = \kappa_1 \rightarrow \text{EW}$ vor
  - nicht in BCNF: $f_3 = \text{Ministerpräsidentin} \rightarrow \text{BLand}$, die linke Seite ist kein Schlüssel

BCNF – Dekompositionsalgorithmus

- Achtung: Die resultierende Zerlegung ist i.A. nicht abhängigkeitserhaltend!

  Starte mit $Z = \{ \mathcal{R}, F \}$

  Solange es noch ein Schema $\mathcal{R}_i \in Z$ gibt, das nicht in BCNF ist:

  - finde nicht-triviale, in R_i geltende FD $\alpha \rightarrow \beta$ mit
    - $\alpha \cap \beta = \emptyset$ und $\alpha \nrightarrow X_i$ (X_i ist Attributmenge von $\mathcal{R}_i$)
  - Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i1} = (X_{i1}, F_{i1})$, $\mathcal{R}_{i2} = (X_{i2}, F_{i2})$ *entlang* der FD $\alpha \rightarrow \beta$

    mit Attributmengen:

    $X_{i1} = \alpha \cup \beta$ und $X_{i2} = X_i \setminus \beta$

    und FDs:

    $F_{i1} = \Pi_{Xi1}(F_i^+)$ und $F_{i2} = \Pi_{Xi2}(F_i^+)$
  - Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i1}, \mathcal{R}_{i2}$ hinzu.

Definition:

$\Pi_X(F) := \{ \alpha \rightarrow \beta \in F \mid (\alpha \cup \beta) \subseteq X \}$

Dekompositionsalgorithmus Beispiel 1

- Städte(Ort, BLand, MinisterpräsidentIn, EW)

- Zerlegung *entlang* von $f_3 = \text{MinisterpräsidentIn} \rightarrow \text{Bland}$
  - Städte1(Ort, MinisterpräsidentIn, EW) mit FDs:
    - $f_1, = \text{MinisterpräsidentIn}, \text{Ort} \rightarrow \text{EW}$

    wegen f_1, in F^+ **(wichtig!)**
  - Regierungen(BLand, MinisterpräsidentIn) mit FDs:
    - $f_2 = \text{BLand} \rightarrow \text{MinisterpräsidentIn}$
    - $f_3 = \text{Ministerpräsidentin} \rightarrow \text{BLand}$

in diesem Beispiel ist die Zerlegung abhängigkeitserhaltend, da sich f_1 rekonstruieren lässt.

- **Bitte beachten:** BLand wurde durch MinisterpräsidentIn in f_1, ersetzt!
- Die Relationen Städte1 und Regierungen sind nun in BNCF.

Dekompositionsalgorithmus Beispiel 2

- Beispiel für nicht abhängigkeitserhaltende Zerlegung

$$\text{PLZVerzeichnis}(\text{ Straße, Ort, BLand, PLZ })$$

mit funktionalen Abhängigkeiten

$$f_1 = \text{Straße, Ort, BLand} \rightarrow \text{PLZ} f_2 = \text{PLZ} \rightarrow \text{Ort, Bland}$$

Die Zerlegung von PLZVerzeichnis entlang von f_2 ergibt

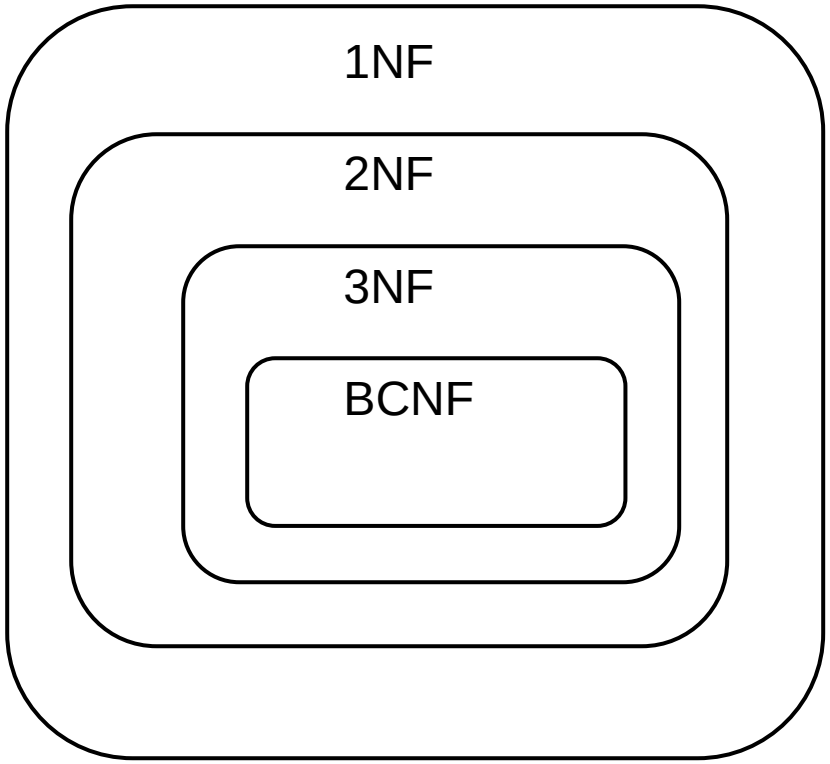
- $\text{Straßen}(\text{ Straße, PLZ })$ mit FDs: $\emptyset$
- $\text{Orte}(\text{ Ort, BLand, PLZ })$ mit FDs: $f_2 = \text{PLZ} \rightarrow \text{Ort, Bland}$

Wie schon diskutiert, geht in diesem Fall die Abhängigkeit f_1 verloren

- Wäre eine Zerlegung nicht abhängigkeitserhaltend, gibt man sich normalerweise mit der dritten Normalform zufrieden

Zusammenfassung

- Ziel der relationalen Entwurfstheorie
  - Was ist ein redundanzfreies relationales Datenbank-Schema?
- Zerlegung von Relationenschemata
  - Verlustlos
  - Abhängigkeitserhaltend
- Normalformen
  - Algorithmen
    - Synthesealgorithmus (3NF)
    - Dekompositionsalgorithmus (BCNF)
  - Abhängigkeitserhaltung ist nur bis zur dritten Normalform garantiert



Transaktionsverwaltung

1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 –
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Ziele dieses Kapitels

Ein DBMS hat die (Angriffs- und Missbrauchs-)Sicherheit und die Korrektheit des Datenbankzustandes unter realen Benutzungsbedingungen zu wahren.

Mit Techniken auf Basis der **Anfragebearbeitung (Views)** unterstützt das DBMS:

- **Integrität** (Integrity Constraints)
  Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
- **Datensicherheit** (Security, Access Rights)
  Schutz vor Zugriffen und Änderungen durch *unberechtigte* Benutzer.

Das Konzept der **Transaktionen** sichert

- **Synchronisation** (Concurrency Control)
  Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer
- **Fehlertoleranz** (Recovery)
  Schutz vor Datenverlust durch technische Fehler (z.B. Programmfehler, Systemabsturz), ohne Unterbrechung des laufenden Betriebs.

Transaktionsverwaltung

1. **Das Transaktionskonzept**
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

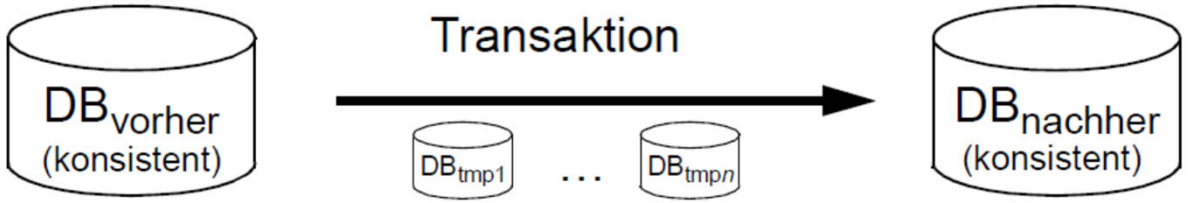
Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Motivation - Transaktionsmanagement

Prinzip	Annahme bisher	In Wahrheit
Isolation	Nur ein Nutzer greift auf die Datenbank lesend und schreibend zu.	Viele Nutzer und Anwendungen lesen und schreiben gleichzeitig.
Atomarität	Anfragen und Updates bestehen aus einer einzigen, atomaren Aktion. DBMS können nicht mitten in dieser Aktion ausfallen.	Auch einfache Anfragen bestehen oft aus mehreren Teilschritten. DBMS können jederzeit ausfallen.

Transaktionen

Transaktionen (TAs) sind die Einheiten *integritätserhaltender Zustandsänderungen* einer Datenbank:



- Eine Transaktion ist „eine Folge von Operationen […], welche eine gegebene Datenbank in ununterbrechbarer Weise von einem konsistenten Zustand in einen [anderen] (nicht notwendig verschiedenen) konsistenten Zustand überführt.“ [Vossen, 2008; S. 638]
- Integritätsaspekte:
  - Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
  - Ablaufintegrität: Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten vermeiden

Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

- Möglicher Bearbeitungsplan:
  1. Erniedrige Stand von “Huber” um 200
  2. Erhöhe Stand von “Meier” um 200

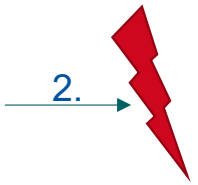
- Möglicher Ablauf:

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.500,00

1.

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.300,00

2. Systemabsturz



Wichtig: Inkonsistenter Datenbankzustand darf nicht entstehen bzw. nicht dauerhaft bestehen bleiben

ACID-Prinzip korrekter Transaktionsabwicklung

Eine grundlegende Charakterisierung von Transaktionsanforderungen ist durch das **ACID**-Prinzip (Härder/Reuter 1983) gegeben:

- **Atomicity** (Atomarität, “alles-oder-nichts”-Prinzip)
  - Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zu tragen
- **Consistency** (Konsistenz, Integritätserhaltung)
  - Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt
- **Isolation** (Isolation, logischer Einbenutzerbetrieb)
  - Innerhalb einer Transaktion nimmt ein Benutzer Änderungen durch andere Benutzer nicht wahr
- **Durability** (Dauerhaftigkeit, Persistenz)
  - Der Effekt einer abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten

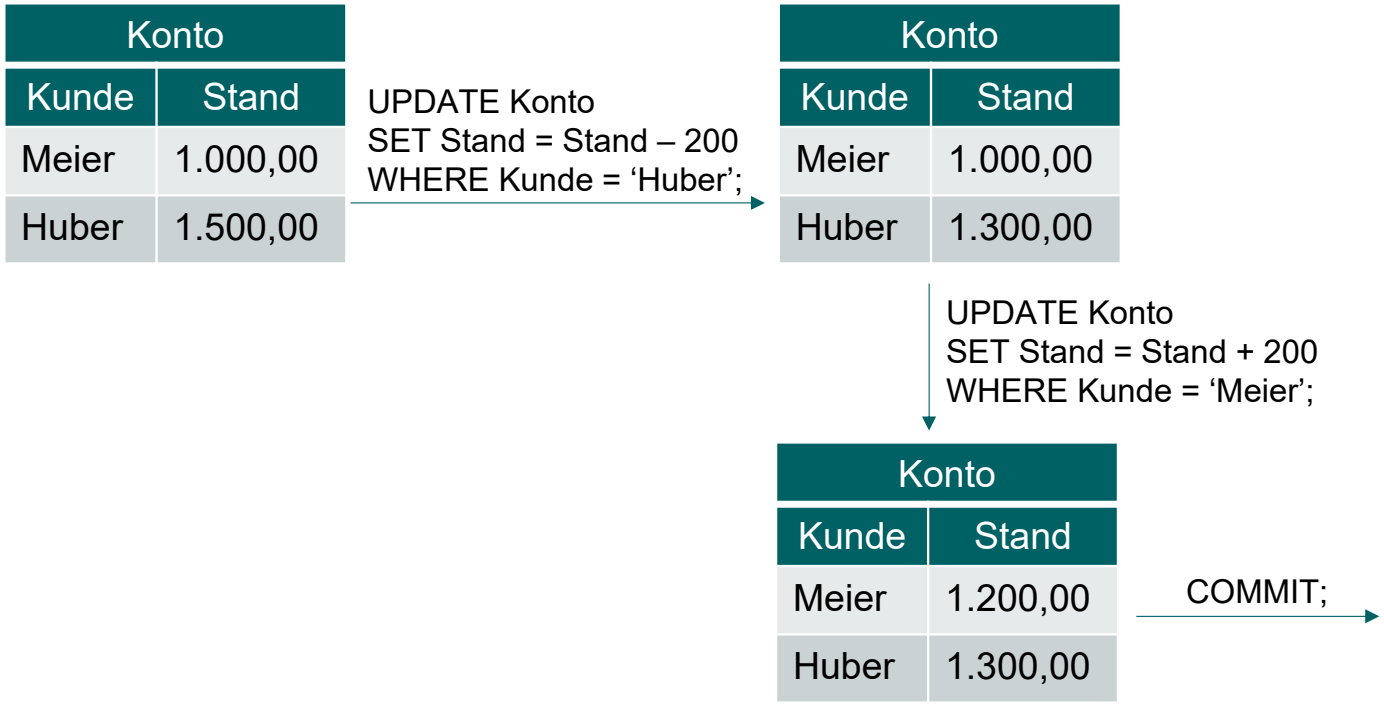
Jede Transaktion muss in endlicher Zeit ausgeführt werden, so dass die ACID Eigenschaften erhalten bleiben

Steuerung von Transaktionen

- **begin of transaction**
  - Beginn der Befehlsfolge einer neuen Transaktion
  - SQL: **begin transaction,** o.ä., oft auch implizit (hier auch: **BOT = Beginn of Transaction)**
- **end of transaction, commit**
  - Bestätigung der Transaktion durch den Benutzer
    - Die Änderungen seit Beginn der Transaktion werden endgültig bestätigt
    - Der jetzt erreichte Zustand soll dauerhaft gespeichert werden
    - Der Zustand wird durch den Benutzer für konsistent erklärt
  - SQL: **commit work** oder nur **commit**
- **abort transaction**
  - Abbruch der Transaktion durch das Programm bzw. den Benutzer (Rücksetzen, ABORT)
    - Die Änderungen der Transaktion werden zurückgesetzt
    - Der ursprüngliche Zustand vor der Transaktion wird wiederhergestellt
  - SQL: **rollback work** oder nur **rollback**

Steuerung von Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

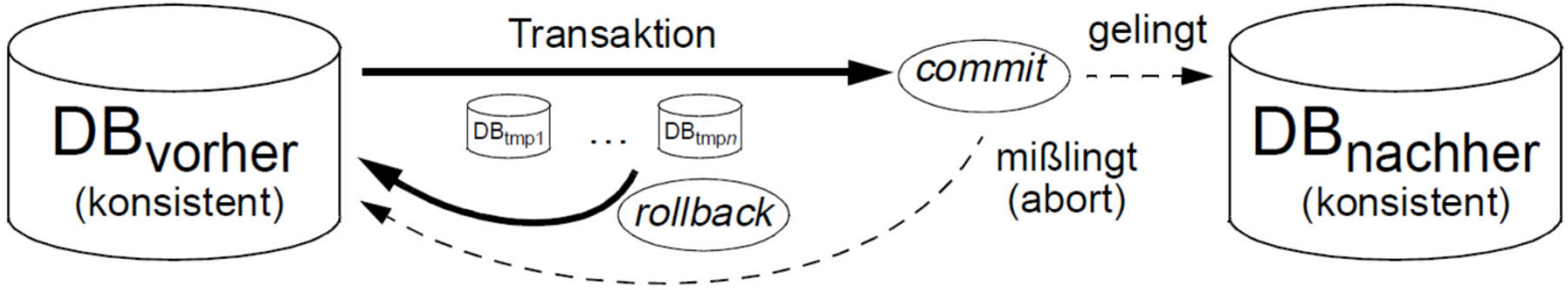


Ausführung einer Transaktion (commit)

Ein **COMMIT** kann …

- *gelingen*: Der neue Zustand wird dauerhaft gespeichert, oder
- *scheitern*: Der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann u.a. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.

Schematischer Ablauf:



Transaktionsverwaltung

1. Das Transaktionskonzept
2. **Synchronisation (Concurrency Control)**
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Transaktionsmodell: Abstraktion auf Lese- und Schreiboperationen

- Eine **Transaktion** (t_i), d.h. ein DB-Programm während der Ausführung, wird vereinfacht betrachtet als ein Programm, das nur aus Lese- und Schreiboperationen auf einer Datenbank besteht
  - Leseoperation **read(x)** liest DB-Objekt x
  - Schreiboperation **write(x)** schreibt DB-Objekt x
- Die Beschränkung auf Lese- und Schreib-Operationen ist aus der Realität abstrahiert (z.B. Weiterverarbeitung von Daten) und dient zur vereinfachten Analyse
- Semantische Information über ein DB-Programm werden nicht mehr betrachtet
- Die Abstraktion ist hinreichend für viele praktische Anwendungen

Probleme beim Ausführen von Transaktionen im Mehrbenutzerbetrieb

Nach dem ACID-Prinzip “Isolation” sollen Transaktionen im logischen Einbenutzerbetrieb ablaufen, d.h. innerhalb einer Transaktion ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen.

Ist die Isolierung der Transaktionen in einem Datenbanksystem nicht sichergestellt, können verschiedene **Anomalien/Synchronisationsprobleme** auftreten:

- Verlorengegangene Änderungen (**Lost Updates**)
- Zugriff auf “schmutzige” Daten (**Dirty Read**, **Dirty Write**)
- Nicht-reproduzierbares Lesen
- Phantomproblem

Probleme beim Ausführung von Transaktionen

Die o.a. Anomalien werden unter anderem am Beispiel der folgenden Flugdatenbank erläutert:

Passagiere		
Flug#	Name	Platz
LH745	Müller	3A
LH745	Meier	6D
LH745	Huber	5C
BA932	Schmidt	9F
BA932	Huber	5C

Fluginfo	
Flug#	AnzPass
LH745	3
BA932	3

Verlorengegangene Änderungen (Lost Update)

Änderungen einer Transaktion können durch Änderungen anderer Transaktionen überschrieben werden und dadurch verloren gehen.

- Beispiel:
  Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus:
  UPDATE Fluginfo SET AnzPass = AnzPass + 1 WHERE Flug# = ‘BA932’;
- Möglicher Ablauf:

T1	T2
Read Fluginfo[Anzpass] → x	
	Read Fluginfo[Anzpass] → y
	$y := y + 1$
	Write y → Fluginfo[Anzpass]
x:= $x + 1$	
Write x → Fluginfo[Anzpass]	

- Ergebnis:
  Beide Transaktionen haben die Anzahl der Passagiere (für denselben Flug) jeweils um eins erhöht. Obwohl zwei Erhöhungen stattgefunden haben, ist in der Datenbank nur die Erhöhung von T1 wirksam. Die Änderung von T2 ist verlorengegangen.

Zugriff auf “schmutzige” Daten (Dirty Read, Dirty Write)

Als “schmutzige” Daten bezeichnet man Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden.

- Beispiel:
  - T1 erhöht das Gehalt um 500 Euro, wird aber später abgebrochen
  - T2 erhöht das Gehalt um 5% und wird erfolgreich abgeschlossen
- Möglicher Ablauf:

T1	T2
UPDATE …. $Gehalt := Gehalt + 500$;	
	UPDATE …. $Gehalt := Gehalt * 1.05$:
	COMMIT;
ROLLBACK;	

- Ergebnis:
  - Der Abbruch der ändernden Transaktion T1 macht die geänderten Werte ungültig, sie werden zurückgesetzt. Die Transaktion T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*).
  - Verstoß gegen *ACID*: Dieser Ablauf verursacht einen dauerhaften fehlerhaften Datenbankzustand (*Consistency*), bzw. T2 muss nach COMMIT zurückgesetzt werden (*Durability*).

Nicht-reproduzierbares Lesen

Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts.

- Beispiel:
  - T1: Gib die Fluginfo für alle Flüge und die Anzahl der Passagiere für BA932 aus
  - T2: Buche den Platz 3F auf dem Flug BA932 für Passagier Meier
- Möglicher Ablauf:

T1	T2
SELECT * FROM Fluginfo	
	INSERT INTO Passagiere(*) VALUES (‚BA932‘,‘Meier‘,‘3F‘);
	UPDATE Fluginfo SET AnzPass= AnzPAss +1 WHERE Flug# = ‚BA932‘;
	COMMIT;
SELECT AnzPass FROM Fluginfo	
Write x → Fluginfo[Anzpass] WHERE Flug# = ‚BA932‘;	

- Ergebnis:
  Die Anweisungen A1.1 und A1.2 liefern ein unterschiedliches Ergebnis für den Flug BA932, obwohl die Transaktion T1 den Datenbankzustand nicht geändert hat.

Phantomproblem

Das Phantomproblem ist ein nicht-reproduzierbares Lesen in Verbindung mit Aggregatfunktionen.

- Beispiel:
  - AnzPass werde jetzt durch COUNT(*) berechnet und nicht mehr in FlugInfo gespeichert
  - T1: Drucke die Passagierliste sowie die Fluginfo für den Flug LH745
  - T2: Buche den Platz 7D auf dem Flug LH745 für Phantomas
- Möglicher Ablauf:

T1	T2
SELECT * FROM Passagiere WHERE Flug# = ‘LH745‘;	
	INSERT INTO Passagiere(*) VALUES (‘LH745‘,‘Phantomas‘,‘7D‘);
	COMMIT;
SELECT AnzPass = COUNT(*) FROM Passagiere WHERE Flug# = ‘LH745‘;	

- Ergebnis:
  Für die Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere schon berücksichtigt ist.

- Grundannahme: Korrektheit
  - Jede Transaktion, isoliert ausgeführt auf einem konsistenten Zustand der Datenbank, hinterlässt die Datenbank wiederum in einem konsistenten Zustand.
  - Lösung aller obigen Probleme: Alle Transaktionen seriell ausführen.
  - Aber: Parallele Ausführung bietet Effizienzvorteile:
    - „Long-Transactions“ über mehrere Stunden hinweg
    - Cache ausnutzen
- Deshalb: Korrekte parallele Pläne (Schedules) finden
  - Korrekt = Serialisierbar

Transaktionen: Definition

- **Annahme**: eine Datenbank $\mathrm{DB} = \{x, y, z, \dots\}$ ist eine Menge von Objekten auf die nur mittels Schreib- und Leseoperationen zugegriffen wird
- Eine **Transaktion** t ist definiert als eine endliche Folge von Schreib- und Leseoperationen:

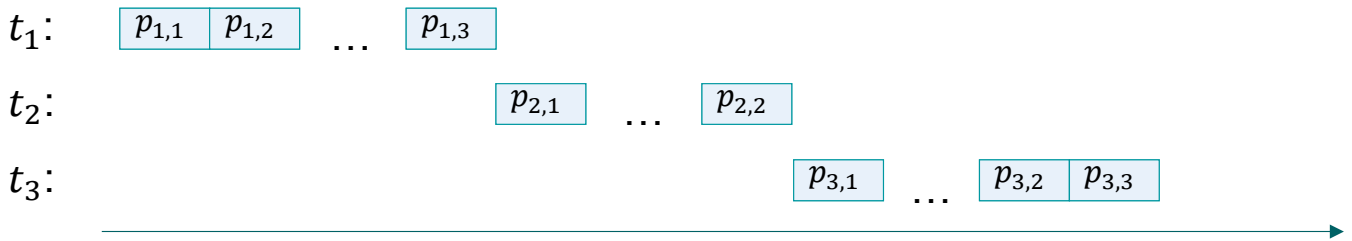
$$t = p_1, \dots, p_n$$

  - mit $n < \infty$
  - $p_i \in \{r(x) \mid x \in \mathrm{DB}\} \cup \{w(x) \mid x \in \mathrm{DB}\}$ für $1 \leq i \leq n$

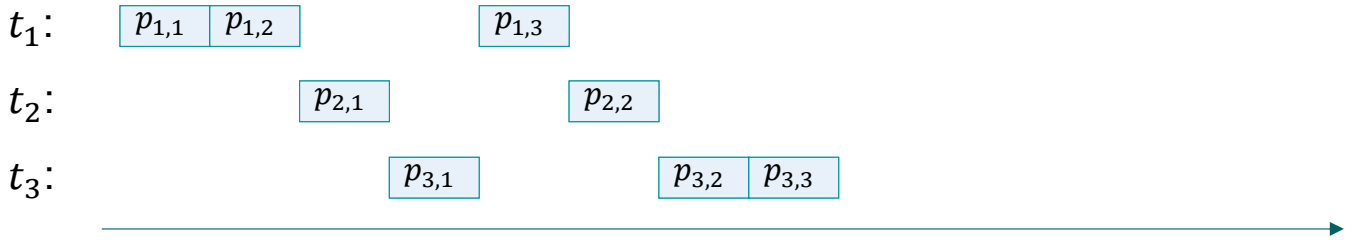
- Indizes kennzeichnen verschiedene (nebenläufige) Transaktionen, z.B.:
  - $t_1 = r_1(x)\, r_1(y)\, r_1(z)\, w_1(z)\, w_1(x)$
  - $t_2 = r_2(x)\, r_2(z)\, w_2(x)\, w_2(y)$

Bearbeitung mehrerer Transaktionen

- Wie können mehrere nebenläufige Transaktionen effizient ausgeführt werden?
- Serieller Ablaufplan (Schedule):



- Verzahnter Schedule:



Bearbeitung mehrerer Transaktionen

- Wie können mehrere nebenläufige Transaktionen effizient ausgeführt werden?

Serieller Ablaufplan (Schedule):

Schritt	t_1	t_2	t_3
1	$p_{1,1}$		
2	$p_{1,2}$		
3	$p_{1,3}$		
4		$p_{2,1}$	
5		$p_{2,2}$	
6		$p_{2,3}$	
7			$p_{3,1}$
8			$p_{3,2}$
9			$p_{3,3}$

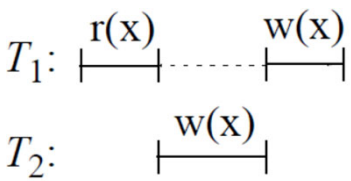
Verzahnter Schedule:

Schritt	t_1	t_2	t_3
1	$p_{1,1}$		
2	$p_{1,2}$		
3		$p_{2,1}$	
4			$p_{3,1}$
5	$p_{1,3}$		
6		$p_{2,2}$	
7			$p_{3,2}$
8		$p_{2,3}$	
9			$p_{3,3}$

Beobachtung:
mögliche Problemursache Nicht-serialisierbare Schedules

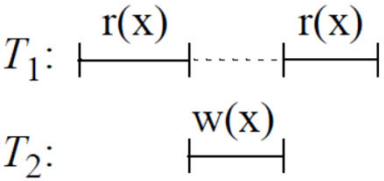
- Man hat früh beobachtet, dass viele Probleme offenbar dann auftreten, wenn Transaktionen sich beinflussen, dh., Schedules „nicht serialisierbar“ sind, z.B.

$s = r_1(x)\, w_2(x)\, w_1(x)$



Lost Update

$s = r_1(x)\, w_2(x)\, r_1(x)$



Non-repeatable Read

- Erster Lösungsansatz zur Formalisierung des Synchronisationsproblems:

Nur „serialisierbare“ Schedules dürfen zugelassen werden.

(Definition im folgenden)

Serielle und Serialisierbare Schedules

Serieller Schedule

Schritt	t_1	t_2
1	BOT	
2	read(A)	
3	write(A)	
4	read(B)	
5	write(B)	
6	commit	
7		BOT
8		read(C)
9		write(C)
10		read(A)
11		write(A)
12		commit

Serialisierbarer Schedule

Schritt	t_1	t_2
1	BOT	
2	read(A)	
3		BOT
4		read(C)
5	write(A)	
6		write(C)
7	read(B)	
8	write(B)	
9	commit	
10		read(A)
11		write(A)
12		commit

Schedules

Serialisierbar?

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Serialisierbar?

Abstrakt

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Gleicher Ablauf, konkret

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Serialisierbar?

Original

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Serialisierbar?

Original

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Serialisierbar?

Serialisierbar?
Nein, denn Effekt entspricht weder dem seriellen Schedule $t_1; t_3$ noch dem seriellen Schedule $t_3; t_1$

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Noch mal die beiden Schedules. Was fällt auf?

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Noch mal die beiden Schedules. Was fällt auf?

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Ergebnisse von $t_1; t_3 \neq t_3; t_1$. Macht das was?

Transaktionsverwaltung

1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. **Protokolle zur Synchronisation (Scheduler)**
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Formaler Korrektheitsbegriff 1: Serialisierbarkeit von Schedules

- Ein **Schedule** für eine Menge $\{t_1, \ldots, t_n\}$ von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der t_i entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.

- Ein **serieller Schedule** ist ein Schedule s von $\{t_1, \ldots, t_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden. Serielle Schedules erfüllen offensichtlich die Isolationsbedingung des ACID-Prinzips.

- Ein Schedule s von $\{t_1, \ldots, t_n\}$ ist **serialisierbar**, wenn er „äquivalent“ ist zu einem (beliebigen) seriellen Schedule (Permutation von $\{t_1, \ldots, t_n\}$).

Um Serialisierbarkeit sicherzustellen, brauchen wir einen **Äquivalenzbegriff**, der
  - möglichst viel Parallelität gestattet (Mehrbenutzerfähigkeit!),
  - auch bei großen Nutzerzahlen effizient zu testen, möglichst automatisch zu garantieren
  - zu jedem Zeitpunkt der Lebensdauer eines DBMS anwendbar ist.
- Wir werden das Konzept „**Konfliktserialisierbarkeit“** im folgenden erarbeiten!

Schedule: Definition

- Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.
- Das **Shuffle-Produkt** $shuffle(T)$ von T bezeichnet die Menge aller Folgen von Aktionen, in welchen die gegebenen Transaktionen $t_1, \ldots, t_n$ als Teilfolgen auftreten und keine weiteren Aktionen vorkommen.

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$

- Dann gilt für die folgende Folge von Aktionen:
  - $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z) \in shuffle(T)$

Schedule: Definition

Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.

- Ein **vollständiger Schedule** s für T ist eine Folge $s \in shuffle(T)$ mit den zusätzlichen Pseudoaktionen c_i) commit) und a_i (abort) für jedes $t_i \in T$ entsprechend den folgenden Regeln:
  - $c_i \in s \Leftrightarrow a_i \notin s$ für alle $1 \leq i \leq n$
  - c_i oder a_i steht in s hinter der letzten Aktion von t_i für alle $1 \leq i \leq n$
- Die **Menge aller vollständigen Schedules** wird als $shuffle_{ac}(T)$ bezeichnet.
- Erweiterung der Definition: Ein **Schedule** ist der Präfix eines Elementes von $shuffle_{ac}(T)$

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\, w_1(x)\, r_1(y)\, w_1(y)$
  - $t_2 = r_2(z)\, w_2(x)\, w_2\ (z)$
  - $t_3 = r_3(x)\, r_3(y)\, w_3(z)$

- Dann gilt für das folgende Schedule:
  - $s_1 = r_1(x)\, r_2(z)\, r_3(x)\, w_2(x)\, w_1(x)\, r_3(y)\, r_1(y)\, w_1(y)\, w_2(z)\, w_3(z) \in shuffle(T)$
  - $s_2 = s_1\, c_1\, c_2\, c_3 \in shuffle_{ac}(T)$
  - $s_3 = r_1(x)\, r_2(z)\, r_3(x)$ ist ein Schedule
  - $s_4 = s_1\, c_1$ ist ein Schedule

Schedule: Definition

- Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.
- Ein vollständiger Schedule $s \in shuffle_{ac}(T)$ ist **seriell**, wenn für jeweils zwei t_i und t_j $(i \neq j)$ alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$
- Dann gilt für die folgenden Schedules:
  - $s_5 = t_1\ c_1\ t_3\ a_3\ t_2\ c_2$ ist ein serieller Schedule

Notationen für Transaktionszustands-Klassen

Sei s ein Schedule. Dann definieren wir die folgenden Mengen für s:

- Die Menge aller Aktionen:
  - $op(s) = \text{Menge aller Aktionen in } s$
- Die Menge aller Transaktionen:
  - $trans(s) = \{t_i \mid s \text{ enthält Aktionen aus } t_i\}$
- Die Menge aller bestätigten Transaktionen:
  - $commit(s) = \{t_i \mid t_i \in trans(s) \wedge c_i \in s\}$
- Die Menge aller abgebrochenen Transaktionen:
  - $abort(s) = \{t_i \mid t_i \in trans(s) \wedge a_i \in s\}$
- Die Menge aller aktiven Transaktionen:
  - $active(s) = trans(s) - \big(commit(s) \cup abort(s)\big)$

Notationen für Transaktionszustands-Klassen

Sei s ein Schedule. Dann definieren wir die folgenden Mengen für s:

- Die Menge aller Aktionen:
  - $op(s) = \text{Menge aller Aktionen in } s$
- Die Menge aller Transaktionen:
  - $trans(s) = \{t_i \mid s \text{ enthält Aktionen aus } t_i\}$
- Die Menge aller bestätigten Transaktionen:
  - $commit(s) = \{t_i \mid t_i \in trans(s) \wedge c_i \in s\}$
- Die Menge aller abgebrochenen Transaktionen:
  - $abort(s) = \{t_i \mid t_i \in trans(s) \wedge a_i \in s\}$
- Die Menge aller aktiven Transaktionen:
  - $active(s) = trans(s) - \big(commit(s) \cup abort(s)\big)$

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$
- Dann gilt für die folgenden Schedules:
  - $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z) \in shuffle(T)$
  - $s_2 = s_1\ c_1\ c_2\ c_3 \in shuffle_{ac}(T)$
  - $s_3 = r_1(x)\ r_2(z)\ r_3(x)$ ist ein Schedule
  - $s_4 = s_1\ c_1$ ist ein Schedule
  - $s_5 = t_1\ c_1\ t_3\ a_3\ t_2\ c_2$ ist ein serieller Schedule

$$
\begin{aligned}
trans(s_2) &= \{t_1, t_2, t_3\} \\
trans(s_4) &= \{t_1, t_2, t_3\} \\
commit(s_4) &= \{t_1\} \\
commit(s_5) &= \{t_1, t_2\} \\
abort(s_4) &= \emptyset \\
abort(s_5) &= \{t_3\} \\
active(s_2) &= \emptyset \\
active(s_4) &= \{t_2, t_3\}
\end{aligned}
$$

Wann sind Operationen zweier Transaktionen in Konflikt?

Gegeben Transaktionen t_i und t_k
- $r_i(x)$ und $r_k(x)$ stehen nicht in Konflikt
- $r_i(x)$ und $w_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $w_i(x)$ und $r_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $w_i(x)$ und $w_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $r_i(x)$ und $w_k(x)$ stehen in Konflikt
- $w_i(x)$ und $r_k(x)$ stehen in Konflikt

Zusammengefasst: Konflikt herrscht falls zwei Aktionen
- das gleiche Datenbankelement betreffen,
- und mindestens eine der beiden Aktionen ein *write* ist.

Konfliktmenge

- Sei s ein Schedule und $t, t' \in trans(s)$ zwei Transaktionen mit $t \neq t'$.
- Zwei Datenoperationen $p \in t$ und $q \in t'$ in s stehen in **Konflikt**, falls sie auf demselben Objekt arbeiten und mindestens eine Datenoperation eine Schreiboperation ist.
- Die **Konfliktmenge** von Schedule s ist definiert als:

$$C(s) = \{(p, q) \mid p, q \text{ in } s \text{ stehen in Konflikt und } p \text{ steht vor } q \text{ in } s\}$$

- Die Elemente der **Konfliktmenge** heißen **Konfliktbeziehungen**.
- Die bereinigte Konfliktmenge $conf(s)$ bezeichnet im Folgenden die Menge der Konfliktmenge eines Schedules s, *bereinigt von abgebrochenen Transaktionen*.
  - Randbemerkung: Wir betrachten hier also nur laufende und erfolgreich abgeschlossene Transaktionen. *Dirty Read-Probleme* können in diesem Modell NICHT formuliert werden, weil sie ja nur Abbruch von Transaktionen entstehen.

Konfliktmenge: Beispiel

- Sei der folgende Schedule gegeben:

$$s = w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(y)\ w_3(x)\ w_3(y)\ c_1\ a_2\ c_3$$

- Dann sind die Konfliktmengen wie folgt gegeben:

$$C(s) = \left\{ \begin{matrix} (w_1(x), r_2(x)), & (w_1(x), w_3(x)), & (r_2(x), w_3(x)), \\ (w_2(y), r_1(y)), & (w_2(y), w_1(y)), & (w_2(y), w_3(y)), \\ (r_1(y), w_3(y)), & (w_1(y), w_3(y)) & \end{matrix} \right\}$$

t_2 wird abgebrochen, daher ist die bereinigte Konfliktmenge:

$$conf(s) = \{(w_1(x), w_3(x)), \quad (r_1(y), w_3(y)), \quad (w_1(y), w_3(y))\}$$

Konfliktäquivalenz

- Zwei Schedules s und s' heißen **konfliktäquivalent**, bezeichnet mit $s \approx_c s'$, falls folgendes gilt:
  - Die Menge der Aktionen sind gleich, d.h.:
    - $op(s) = op(s')$
  - Die bereinigten Konfliktmengen sind gleich, d.h.:
    - $conf(s) = conf(s')$

- Konfliktäquivalenz kann wie folgt überprüft werden:
  - Für zwei gegebene Schedules (mit derselben Menge von Operationen), bestimme die Menge der bereinigten Konfliktbeziehungen und überprüfe sie auf Gleichheit.

Konfliktäquivalenz: Beispiel

- Seien die folgenden Schedules gegeben:

$$s = r_1(x)\ r_1(y)\ w_2(x)\ w_1(y)\ r_2(z)\ w_1(x)\ w_2(y)$$

$$s' = r_1(y)\ r_1(x)\ w_1(y)\ w_2(x)\ w_1(x)\ r_2(z)\ w_2(y)$$

- $conf(s) = \{(r_1(x), w_2(x)),\ (r_1(y), w_2(y)),\ (w_2(x), w_1(x)),\ (w_1(y), w_2(y))\}$
- $conf(s') = \{(r_1(y), w_2(y)),\ (r_1(x), w_2(x)),\ (w_1(y), w_2(y)),\ (w_2(x), w_1(x))\}$

- Die beiden Schedules sind konfliktäquivalent, d.h. es gilt $s \approx_c s'$.

Konfliktserialisierbarkeit

- Ein vollständiger Schedule s heißt **konfliktserialisierbar**, falls ein serieller Schedule s' existiert mit $s \approx_c s'$.
- Die **Klasse aller konfliktserialisierbaren Schedules** bezeichnen wir mit *CSR*.

- Beispiele:
  - $s = r_2(y)\ w_1(x)\ w_1(y)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - $s' = r_1(x)\ r_2(x)\ w_2(y)\ c_2\ w_1(x)\ c_1 \in CSR$

Betrachte:

- $conf(s) = \{(r_2(y), w_1(y)), (w_1(x), w_2(y))\}$
- $conf(s') = \{(r_2(x), w_1(x))\}$

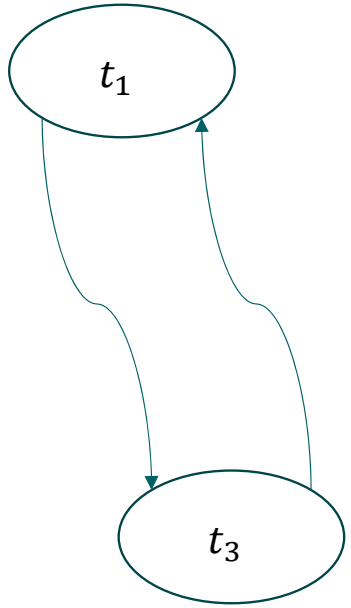
In der Konfliktmenge von s ist ein Zyklus!

- Ob ein Schedule zur Klasse der konfliktserialisierbaren Schedules gehört kann einfach geprüft werden.

Serialisierbar?

Serialisierbar?
Nein, denn Effekt entspricht weder dem seriellen Schedule $t_1; t_3$ noch dem seriellen Schedule $t_3; t_1$

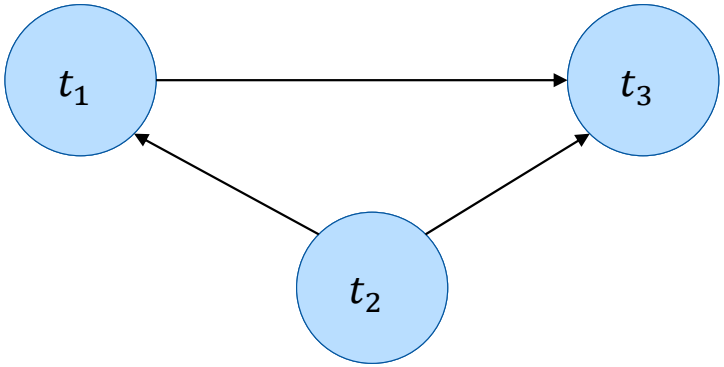
Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	



Konfliktgraph

- Sei s ein Schedule. Der **Konfliktgraph** $G(s) = (V, E)$ von s ist wie folgt definiert:
  - Kontenmenge $V = commit(s)$
  - Kantenmenge $E = \{(t, t') \mid t \neq t' \wedge \exists p \in t, \exists q \in t' : (p, q) \in conf(s)\}$

- Beispiel:
  - Konfliktgraph für Schedule $s = r_1(x)\ r_2(x)\ w_1(x)\ r_3(x)\ w_3(x)\ w_2(y)\ c_3\ c_2\ w_1(y)\ c_1$:



Serialisierbarkeitssatz

- Die Zugehörigkeit eines Schedules s zur Klasse der konfliktserialisierbaren Schedules *CSR* kann durch die folgende Aussage bestimmt werden:

$$\boldsymbol{s \in CSR \Leftrightarrow G(s) \text{ ist azyklisch}}$$

- Beweis wird mittels topologischem Sortieren geführt

- Korollar: Zugehörigkeit zu CSR kann in polynomieller Zeit geprüft werden

Beweis

Konfliktgraph ist zykelfrei ⟺ Schedule ist konfliktserialisierbar

- Konfliktgraph ist zykelfrei ⇐ Schedule ist konfliktserialisierbar
  - Leicht: Konfliktgraph hat Zykel ⟹ Schedule ist nicht konfliktserialisierbar
  - $t_1 \rightarrow t_2 \rightarrow \ \ldots \mathrm{t_n} \ \rightarrow t_1$
- Konfliktgraph ist zykelfrei ⇒ Schedule ist konfliktserialisierbar
  - Induktion über Anzahl der Transaktionen n
  - *n* = 1: Graph und Schedule haben nur eine Transaktion.
  - *n* = *n* + 1:
    - Graph ist zykelfrei
      ⇒ mindestens ein Knoten t_i ohne eingehende Kante
      ⇒ es gibt keine Aktion einer anderen Transaktion, die vor einer Aktion in t_i ausgeführt wird und mit dieser Aktion in Konflikt steht
    - Alle Aktionen aus t_i können an den Anfang bewegt werden (Reihenfolge innerhalb t_i bleibt erhalten).
    - Restgraph ist wieder azyklisch (Entfernung von Kanten aus einem azyklischen Graph kann ihn nicht zyklisch machen).
    - Restgraph hat *n*-1 Transaktionen

Einige Synchronisationsprobleme

- Welche Synchronisationsprobleme werden durch *CSR* verhindert?
  - Lost Update
    - $s = r_1(x)\ r_2(x)\ w_1(x)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - Dirty Read
    - $s = r_1(x)\ w_1(x)\ r_2(x)\ \boldsymbol{a_1}\ w_2(x)\ \boldsymbol{c_2} \in \boldsymbol{CSR}$
  - Phantom
    - $s = r_1(x)\ r_1(y)\ r_2(z)\ w_2(z)\ r_2(x)\ w_2(x)\ c_2\ r_1(z)\ c_1 \notin CSR$

Konfliktserialisierbarkeit und Fehlersicherheit

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

  - aus Sicht der Fehlersicherheit nicht akzeptabel, da t_2 Objekt x von t_1 liest und t_1 danach abbricht (Dirty Read)
  - Datenbank muss dafür sorgen, dass $w_1(x)$ nicht durchgeführt wird und t_2 nicht von t_1 liest
- **Fehlersicherheit** eines Schedules ist die Eigenschaft, dass dieser Schedule sich bei Abbruch einer oder mehreren Transaktionen genauso verhält wie ein ähnlicher Schedule, der ausschließlich die nicht abgebrochenen Transaktionen enthält. Fehlersicherheit ist eine „orthogonale“ Eigenschaft, die erst zusammen mit Konfliktserialisierbarkeit zu korrekter Synchronisation führt

„liest-von“-Notation

- Sei s ein Schedule, dann bezeichnet $p <_s q$ das Auftreten von Aktion p vor q
- Seien $t_i, t_j \in trans(s)$:

1) $\boldsymbol{t_i}$ **liest** $\boldsymbol{x}$ **von** $\boldsymbol{t_j}$ **in** $\boldsymbol{s}$, falls alle drei Bedingungen gelten:
   a) $w_j(x) <_s r_i(x)$
      - Eine Aktion $r_i(x)$ liest x von $w_j(x)$
   b) $a_j \not<_s r_i(x)$
      - t_j ist zum Zeitpunkt des Lesens nicht abgebrochen
   c) $w_j(x) <_s w_k(x) <_s r_i(x) \Rightarrow a_k <_s r_i(x)$
      - $w_j(x)$ ist der letzte echte Schreiber auf x vor $r_i(x)$
2) $\boldsymbol{t_i}$ **liest von** $\boldsymbol{t_j}$ **in** $\boldsymbol{s}$, falls t_i irgendein x von t_j in s liest

„liest-von“-Notation: Beispiel

- Sei s ein Schedule mit Transaktionen $t_1, t_2, t_3 \in trans(s)$:

$$s = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ r_3(x)\ w_2(z)\ a_2\ r_1(z)\ c_1\ c_3$$

- Dann gilt:
  - t_3 liest x von t_2 aber nicht von t_1
  - t_2 liest y von t_1
  - t_1 liest z nicht von t_2

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Eine Transaktion wird freigegeben (committed), wenn alle anderen Transaktionen von denen sie gelesen hat, auch freigegeben wurden.
- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit: Beispiel

Seien s_I, s_{II} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_I) \cup trans(s_{II})$:

$$s_I = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$$

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

- Dann gilt:
  - t_2 liest y von t_1 in s_I und $c_2 \in s_I$, aber $c_1 \not\prec_{s_I} c_2$. Hieraus folgt $s_I \notin RC$.
  - $s_{II} \in RC$, da die Commit-Operation von t_2 hinter der von t_1 steht.

- Weiteres Problem tritt auf in s_{II}, falls t_1 direkt nach $r_2(y)$ abbrechen würde
  - ➢ Dirty Read-Situation würde Abbruch von t_2 nach sich ziehen

Vermeidung kaskadierender Aborts

- Ein Schedule s **vermeidet kaskadierende Aborts**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest } x \text{ von } t_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

- Eine Transaktion darf nur Werte von bereits erfolgreich abgeschlossenen Transaktionen lesen.
- Die **Klasse aller Schedules, welche kaskadierende Aborts vermeiden**, bezeichnen wir mit ACA (avoids cascading aborts).

Vermeidung kaskadierender Aborts: Beispiel

Seien s_{II}, s_{III} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{II}) \cup trans(s_{III})$:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

- Dann gilt:
  - $s_{II} \notin ACA$
  - $s_{III} \in ACA$

- Weiteres Problem tritt auf in s_{III}, falls t_1 direkt nach $w_2(x)$ abstürzt
  - t_2 braucht nicht abgebrochen zu werden
  - Objekt x muss auf den richtigen Zustand zurückgesetzt werden

Striktheit

- Ein Schedule s heißt **strikt**, falls für alle Transaktionen $t_i, \in trans(s)$ und für alle Aktionen $p_i(x) \in op(t_i)$ gilt:

$$w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$$

- Kein Objekt wird gelesen oder überschrieben, bis die Transaktion, welche es zuletzt geschrieben hat, (erfolgreich oder erfolglos) beendet ist.
- Die **Klasse aller strikten Schedules** bezeichnen wir mit ST.

Striktheit: Beispiel

Seien s_{III}, s_{IV} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{III}) \cup trans(s_{IV})$:

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

$$s_{IV} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$$

- Dann gilt:
  - $s_{III} \notin ST$
  - $s_{IV} \in ST$

Korrektheit von Schedules

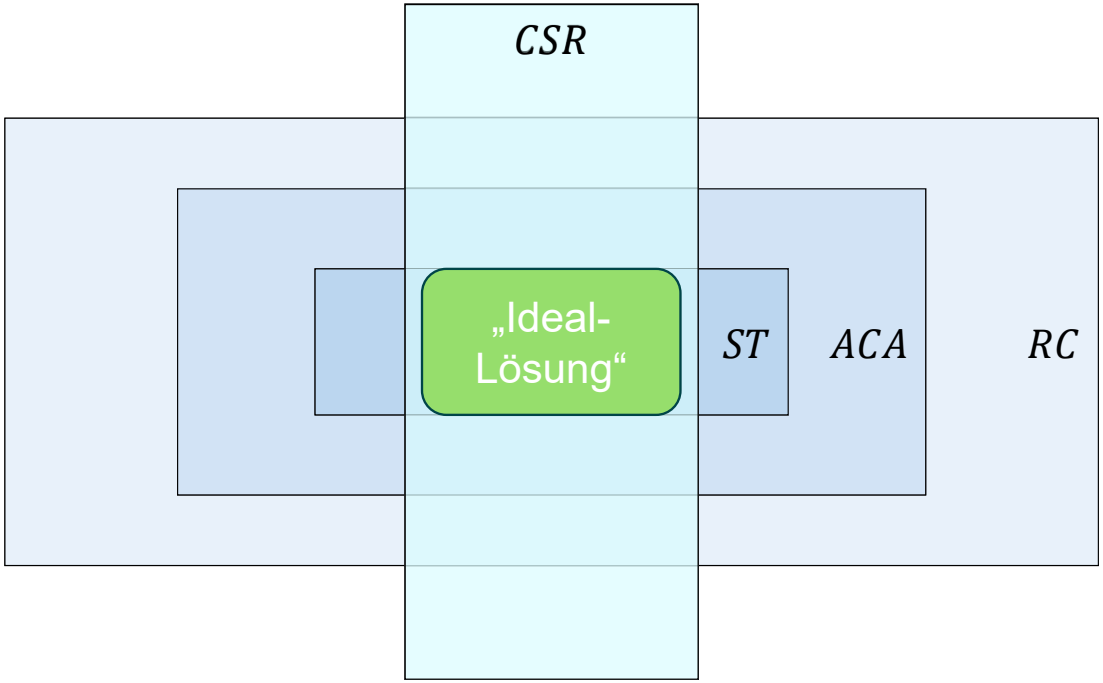
- Fehlersicherheit (ST, ACA, RC) und Konfliktserialisierbarkeit (CSR) sind „orthogonale“ Anforderungen an Schedules

- Ein Schedule s heißt **korrekt**, falls er sowohl konfliktserialisierbar als auch fehlersicher ist, d.h. falls er in der Klasse CSR und in einer der Klassen RC, ACA oder ST liegt.

Fehlersichere Schedules

- Es gilt der folgende Zusammenhang: $ST \subset ACA \subset RC$

CSR : Konfliktserialisierbarkeit
RC : Recoverable
ACA : avoids cascading aborts
ST : Strikt



Einige Synchronisationsprobleme

- Welche Synchronisationsprobleme werden durch *CSR* verhindert?
  - Lost Update
    - $s = r_1(x)\ r_2(x)\ w_1(x)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - Dirty Read
    - $s = r_1(x)\ w_1(x)\ r_2(x)\ \boldsymbol{a_1}\ w_2(x)\ \boldsymbol{c_2} \in \boldsymbol{CSR}$
  - Non Repeatable Read
    - $s = r_1(x) r_2(x) w_2(x) c_2 r_1(x) c_1 \notin CS$

Ist Konfliktserialisierbarkeit ausreichend?

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

- Warum?

Ist Konfliktserialisierbarkeit ausreichend?

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

- Warum?

  - Was passiert in dem Schedule: t_1 liest x, schreibt dann x dann. t_2 liest dann das modifizierte x. Dann bricht t_1 aber ab, und t_2 schreibt und committet einen Wert für x in die Datenbank, der nicht da sein sollte.
  - Also: **Dirty Read**
  - Datenbank muss dafür sorgen, dass $w_1(x)$ nicht durchgeführt wird und t_2 nicht von t_1 liest

Zusätzlich: Rücksetzbarkeit (Recovery)

Anforderung an Zulässige Schedule:

- Transaktionen sollte zu jedem Zeitpunkt vor der Ausführung eines commit zurückgesetzt werden können.
- Das Zurücksetzen einer Transaktion sollte keine anderen Transaktionen beeinflussen

Formalisierung ist notwendig:

„liest-von“-Notation

- Sei s ein Schedule, dann bezeichnet $p <_s q$ das Auftreten von Aktion p vor q
- Seien $t_i, t_j \in trans(s)$:
  1) **t_i liest x von t_j in s**, falls alle drei Bedingungen gelten:
     a) $w_j(x) <_s r_i(x)$
        - Eine Aktion $r_i(x)$ liest x von $w_j(x)$
     b) $a_j \not<_s r_i(x)$
        - t_j ist zum Zeitpunkt des Lesens nicht abgebrochen
     c) $w_j(x) <_s w_k(x) <_s r_i(x) \Rightarrow a_k <_s r_i(x)$
        - $w_j(x)$ ist der letzte echte Schreiber auf x vor $r_i(x)$
  2) **t_i liest von t_j in s**, falls t_i irgendein x von t_j in s liest

Schritt	t_1	t_2	t_3
1	$w_1(x)$		
2	$w_1(y)$		
3		$r_2(u)$	
4		$w_2(x)$	
5		$r_2(y)$	
6			$r_3(x)$
7		$w_2(z)$	
8		a_2	
9	$r_1(z)$		
10	c_1		
11			c_3

Beispiel:

- Sei s ein Schedule mit Transaktionen $t_1, t_2, t_3 \in trans(s)$:
  $s = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ r_3(x)\ w_2(z)\ a_2\ r_1(z)\ c_1\ c_3$

Dann gilt:

t_3 liest x von t_2 aber nicht von t_1
t_2 liest y von t_1
t_1 liest z nicht von t_2

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen in s gilt, dass sie erst dann committed wird, wenn alle Transaktionen, von denen sie gelesen hat, auch committed wurden.

Formal:

Für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen in s gilt, dass sie erst dann committed wird, wenn alle Transaktionen, von denen sie gelesen hat, auch committed wurden.

Formal:

Für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit: Beispiel

Seien s_I, s_{II} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_I) \cup trans(s_{II})$:

$s_I = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7		c_2
8	$w_1(z)$	
9	c_1	

$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Dann gilt:
  - t_2 liest y von t_1 in s_I und $c_2 \in s_I$, aber $c_1 \not\prec_{s_I} c_2$. Hieraus folgt $s_I \notin RC$.
  - $s_{II} \in RC$, da die Commit-Operation von t_2 hinter der von t_1 steht

Weiteres Problem:

Schauen wir uns s_{II} nochmal genauer an:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Was passiert wenn t_1 direkt nach $r_2(y)$ abbricht?

Weiteres Problem:

Schauen wir uns s_{II} nochmal genauer an:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Was passiert wenn t_1 direkt nach $r_2(y)$ abbricht?
  - Dirty Read-Situation: Abbruch von t_1 sieht Abbruch von t_2 nach sich.
  - **kaskadierender Abort**

Vermeidung kaskadierender Aborts

- Ein Schedule s **vermeidet kaskadierende Aborts**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest } x \text{ von } t_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

- Eine Transaktion darf nur Werte von bereits erfolgreich abgeschlossenen Transaktionen lesen.
- Die **Klasse aller Schedules, welche kaskadierende Aborts vermeiden**, bezeichnen wir mit ACA (avoids cascading aborts).

Vermeidung kaskadierender Aborts: Beispiel

Seien s_{II}, s_{III} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{II}) \cup trans(s_{III})$:

$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Dann gilt:
  - $s_{II} \notin ACA$
  - $s_{III} \in ACA$

Schauen wir uns s_{III} nochmal genauer an

Schauen wir uns s_{III} nochmal genauer an.

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Weiteres Problem tritt auf in s_{III}, falls t_1 direkt nach $w_2(x)$ abstürzt
  - t_1 hat zwar Objekt x geschreiben, t_2 braucht aber nicht abgebrochen zu werden
  - Bei Abort von t_1 muss t_2 analysiert werden um den richtigen Wert von x zu bestimmen.
  - Gibt es Schedules bei denen das nicht nötig ist?

Striktheit

- Ein Schedule s heißt **strikt**, falls für alle Transaktionen $t_i, \in trans(s)$ und für alle Aktionen $p_i(x) \in op(t_i)$ gilt:

$$w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$$

- Kein Objekt wird gelesen oder überschrieben, bis die Transaktion, welche es zuletzt geschrieben hat, (erfolgreich oder erfolglos) beendet ist.
- Die **Klasse aller strikten Schedules** bezeichnen wir mit ST.

Striktheit: Beispiel

Seien s_{III}, s_{IV} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{III}) \cup trans(s_{IV})$:

$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

$s_{IV} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4	$w_1(z)$	
5	c_1	
6		$w_2(x)$
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Dann gilt:
  - $s_{III} \notin ST$
  - $s_{IV} \in ST$

Korrektheit von Schedules

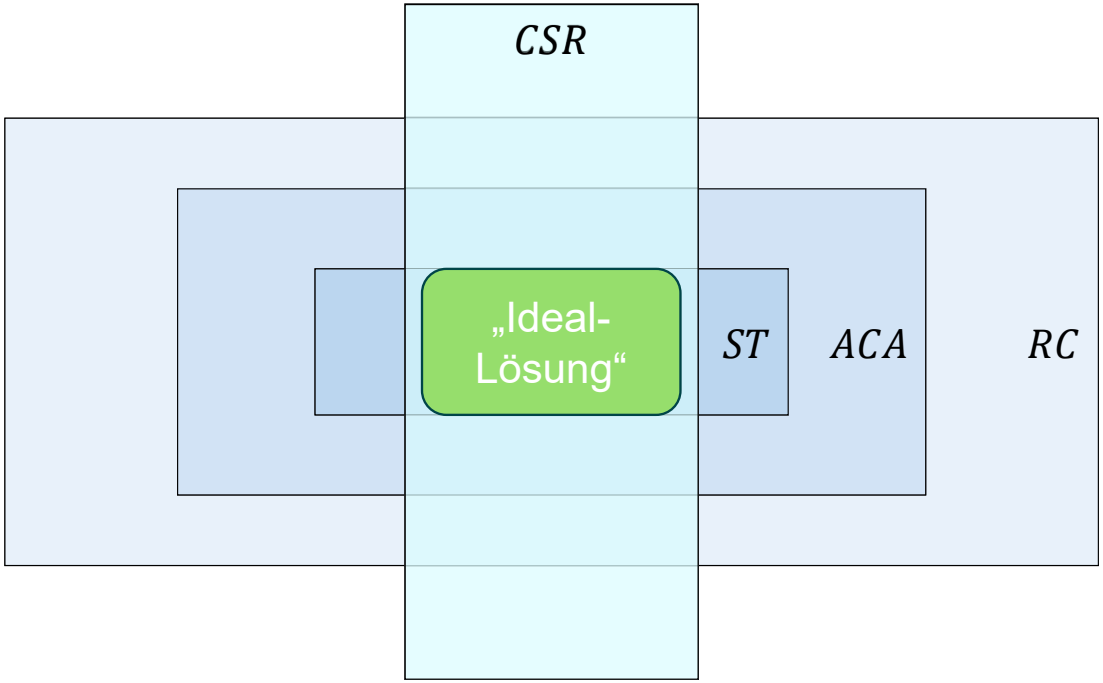
- Fehlersicherheit (ST, ACA, RC) und Konfliktserialisierbarkeit (CSR) sind „orthogonale“ Anforderungen an Schedules

- Ein Schedule s heißt **korrekt**, falls er sowohl konfliktserialisierbar als auch fehlersicher ist, d.h. falls er in der Klasse CSR und in einer der Klassen RC, ACA oder ST liegt.

Zusammenhang zwischen den Klassen RC, ACA, ST, und CSR

- Es gilt der folgende Zusammenhang: $ST \subset ACA \subset RC$

CSR : Konfliktserialisierbarkeit
RC : Recoverable
ACA : Avoids cascading aborts
ST : Strikt



Transaktionsverwaltung

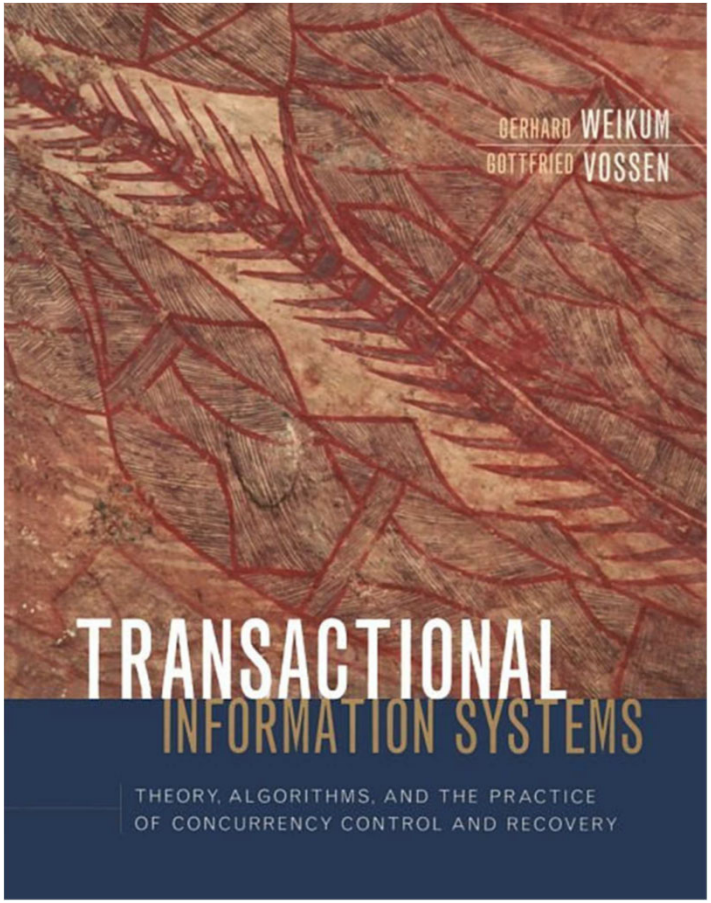
1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. **Protokolle zur Synchronisation (Scheduler)**
4. Fehlertoleranz (Recovery)

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

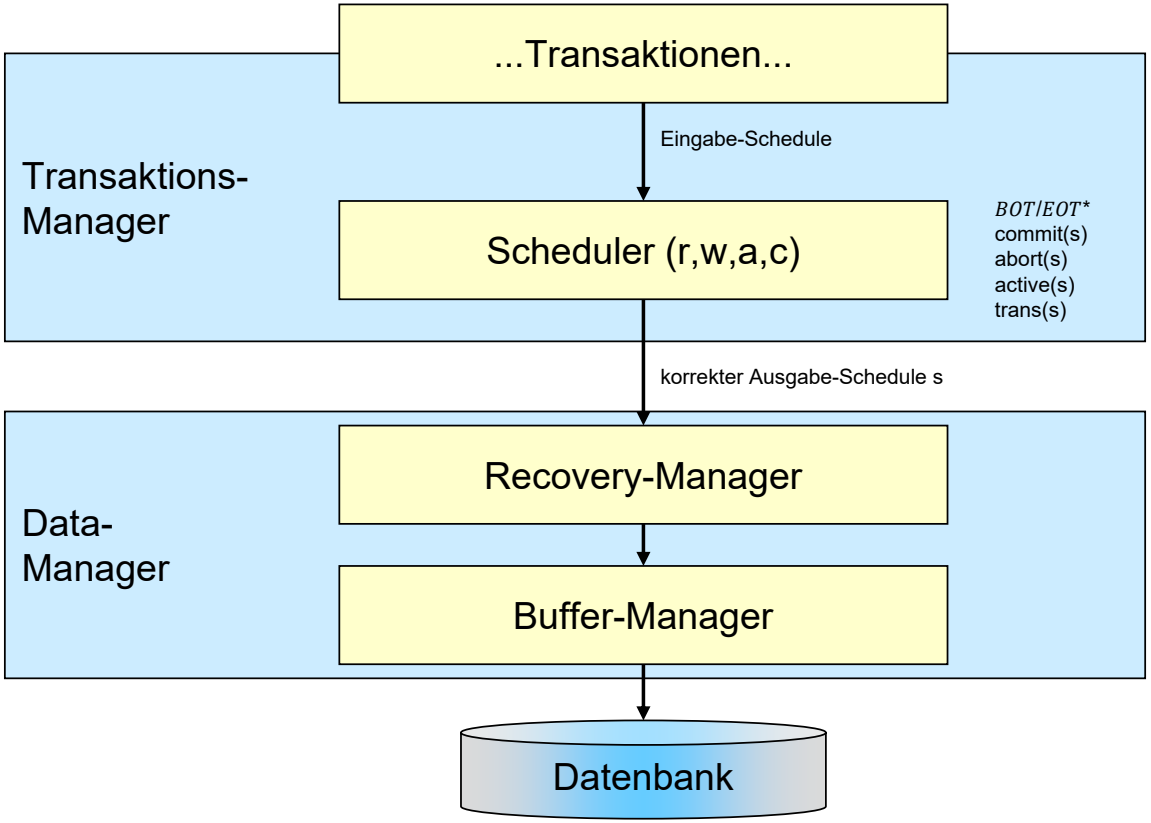
Literatur

Gerhard Weikum and Gottfried Vossen
Transactional Information Systems:
Theory, Algorithms, and
The Practice of Concurrency Control and Recovery

ISBN 1-55860-508-8



Transaktionenverarbeitende Schichten eines DBMS



* BOT – begin of transaction
EOT – end of transaction

Transaktions-Manager

- Der **Transaktions-Manager** nimmt die auszuführenden Transaktionen entgegen und verwaltet
  - die Mengen $trans$, $commit$, $abort$ und $active$
  - für jede Transaktion eine Warteschlange von ausführbereiten Aktionen
- Einzelne Aktionen werden dem **Scheduler** übergeben
- Der Transaktions-Manager umschließt jede Transaktion mit den folgenden Aktionen:
  - BOT (begin of transaction)
    - Kennzeichnet den Beginn einer Transaktion
  - EOT (end of transaction)
    - Kennzeichnet des Ende einer Transaktion
- Wird $EOT(t_i)$ für eine Transaktion t_i erkannt, so wird diese Aktion durch c_i ersetzt.
- Tritt ein Fehler in einer Transaktion t_i auf, so wird dieser durch die Aktion a_i gekennzeichnet.

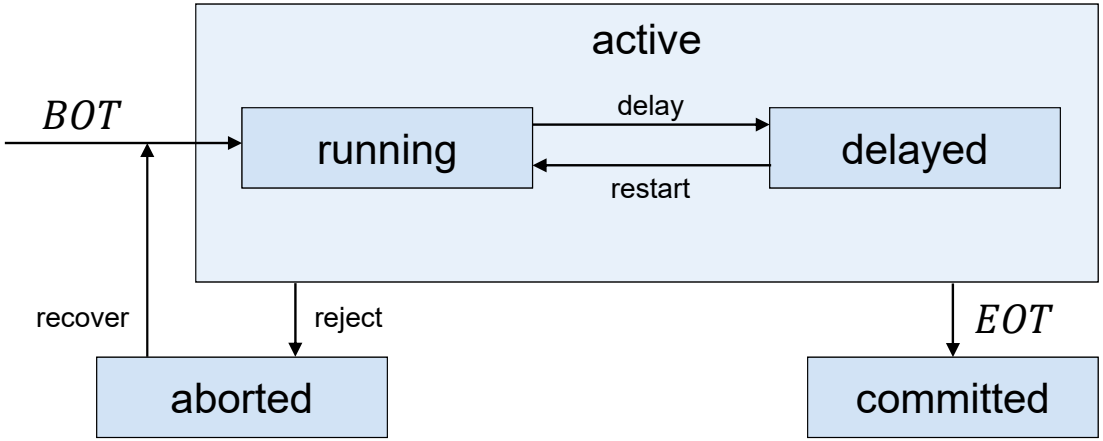
Data-Manager

- Der **Data-Manager** führt die Aktionen des (korrekten) Ausgabe-Schedules der Reihe nach aus:
  - für Aktion r liest er ein Datenobjekt aus der Datenbank in den Puffer
  - für Aktion w schreibt er ein Datenobjekt in die Datenbank oder in den Puffer
  - für Aktion c macht er das Ergebnis der Transaktion „permanent“
  - für Aktion a macht er die Transaktion „ungeschehen“
- Der Data-Manager besteht aus zwei Komponenten:
  - Der **Recovery-Manager** ist dafür verantwortlich, dass in der Datenbank nur die Effekte von freigegebenen und keine Effekte von abgebrochenen Transaktionen erscheinen.
  - Der **Buffer-Manager** stellt die Schnittstelle zur Datenbank dar und verwaltet den Puffer.

Scheduler

- Der **Scheduler** erhält als Eingabe-Schedule Aktionen vom Typ r, w, a, c und muss daraus einen korrekten Ausgabe-Schedule erzeugen
- Dabei kann der Scheduler die Aktion
  a) ausführen (execute)
     - Die Aktion wird sofort ausgegeben, d.h. an den Ausgabe-Schedule angehängt
     - Für alle Aktionen r, w, a, c
  b) zurückweisen (reject)
     - Die Aktion wird nicht ausgeführt, Abbruch der entsprechenden Transaktion t_i mit a_i
     - Nur für Datenoperationen r, w
  c) verzögern (delay)
     - Die Aktion wird weder ausgeführt noch abgelehnt, sondern an den Transaktions-Manager zurückgegeben
     - Nur für Datenoperationen r, w

Zustände einer Transaktion



- Nach dem BOT wird eine Transaktion aktiv
- Die Transaktion kann dann entweder im laufenden oder verzögerten Zustand sein
- Nach dem EOT wird die Transaktion geschrieben, falls sie voher nicht abgebrochen wird

Sicherheit/Ausdrucksstärke eines Schedulers

- Bezeichne $\varepsilon(S)$ das **Erzeugnis** eines Schedulers S, d.h. die Menge aller Schedules, welche S als Ausgabe erzeugen kann.
- Ein Scheduler S ist
  - ***s-sicher***, falls $\varepsilon(S) \subseteq CSR$ gilt
  - ***f-sicher***, falls $\varepsilon(S) \subseteq RC$ (oder ACA oder ST) gilt
  - ***sicher***, falls $\varepsilon(S) \subseteq CSR \cap RC$ (oder ACA oder ST) gilt

- *S-sichere* Scheduler erzeugen konfliktserialisierbare Ausgabe-Schedules.
- *F-sichere* Scheduler erzeugen fehlersichere Ausgabe-Schedules.
- Die *Ausdrucksstärke* eines Schedulers bestimmt sich dadurch, wie gut er die Gesamtmenge sicherer Schedules abdeckt (möglichst viele der konfliktserialisierbaren Schedules bei möglichst effizienter f-Sicherheit).

Sicherheit/Ausdrucksstärke eines Schedulers

- Bezeichne $\varepsilon(S)$ das **Erzeugnis** eines Schedulers S, d.h. die Menge aller Schedules, welche S als Ausgabe erzeugen kann.
- Ein Scheduler S ist
  - ***s-sicher***, falls $\varepsilon(S) \subseteq CSR$ gilt
  - ***f-sicher***, falls $\varepsilon(S) \subseteq RC$ (oder ACA oder ST) gilt
  - ***sicher***, falls $\varepsilon(S) \subseteq CSR \cap RC$ (oder ACA oder ST) gilt

- *S-sichere* Scheduler erzeugen konfliktserialisierbare Ausgabe-Schedules.
- *F-sichere* Scheduler erzeugen fehlersichere Ausgabe-Schedules.
- Die *Ausdrucksstärke* eines Schedulers bestimmt sich dadurch, wie gut er die Gesamtmenge sicherer Schedules abdeckt (möglichst viele der konfliktserialisierbaren Schedules bei möglichst effizienter f-Sicherheit).

Wie kann ein Scheduler konfliktserialisierbare Schedules garantieren?

- Konfliktgraph ist Methode wie ein Scheduler Transaktionen auf Serialisierbarkeit testen kann
- Was kann ein Scheduler machen wenn Transaktionen nicht serialisierbar sind?

Sperrende und Nicht-sperrende Scheduler

- **Sperrende Scheduler**
  - Pessimistische Ablaufsteuerung durch Sperrverfahren, Locking
  - Lese- und Schreibsperren verhindern, dass Änderungen nebenläufige Transaktionen beeinflussen
  - *Nachteil*: Schreibende und nur-lesende Transaktionen müssen ggf. warten, bis andere Transaktionen abgeschlossen sind.
  - *Vorteil*: In der Regel nur wenige Rücksetzungen aufgrund von Synchronisationsproblemen nötig.
  - ➢ Standardverfahren in kommerziellen DBMS
- **Nicht-sperrende Scheduler**
  - Optimistische Ablaufsteuerung durch Zeitstempelverfahren
  - Transaktionen dürfen bis zum COMMIT ungehindert arbeiten.
  - Bei COMMIT wird geprüft, ob ein Konflikt aufgetreten ist (Validierung). Die Transaktion wird ggf. zurückgesetzt.
  - Die Validierung wird anhand von Zeitstempeln durchgeführt (“Gab es seit Beginn der Transaktion ein Commit einer anderen Transaktion, das dieselben Daten betrifft?”).
  - ➢ Nur geeignet, falls Konflikte zwischen Schreibern sehr selten auftreten.

Sperren (Locking)

- **Sperren** (Locks) dienen zur Synchronisation von Zugriffen auf gemeinsam genutzte Datenobjekte
- Diese Sperren werden vom Scheduler für Transaktionen gesetzt und freigegeben
- Gesetzte Sperre signalisiert Unverfügbarkeit des Datenobjektes
  - Vor Zugriff wird Verfügbarkeit geprüft und Sperre gesetzt.
  - Datenobjekt ist für spezifische Transaktion gesperrt
  - Nach Zugriff wird Sperre aufgehoben.
- Für jedes Datenobjekt x gibt es zwei Arten von Sperren:
  - Lese-Sperre: $rl(x)$ (read lock) und $ru(x)$ (read unlock)
  - Schreib-Sperre: $wl(x)$ (write lock) und $wu(x)$ (write unlock)

- Sperren in Konflikt bedeutet gleichzeitig Operationen in Konflikt
- Ein sperrender Scheduler fügt jeder Transaktion *rl/wl/ru/wu*-Aktionen hinzu

Drei Regeln zur Anwendung von Sperren

- Für jede Transaktion t_i, die vollständig in einem Schedule s enthalten ist, gilt:
  - **(L1) Jede Lese-/Schreiboperation wird in der richtigen Reihenfolge ge- und entsperrt:**
    - Falls t_i eine Aktion $r_i(x)$ bzw. $w_i(x)$ enthält, so steht irgendwo davor $rl_i(x)$ bzw. $wl_i(x)$ und irgendwo dahinter $ru_i(x)$ bzw. $wu_i(x)$ in s
  - **(L2) Jedes von der Transaktion verwendete Datenobjekt wird gesperrt:**
    - Für jedes x das von t_i verwendet wird, existiert genau ein $rl_i(x)$ bzw. $wl_i(x)$ in s
  - **(L3) Kein Entsperren ist überflüssig:**
    - Kein ru_i/wu_i ist redundant

- Die von einem sperrenden Scheduler produzierten Schedules enthalten also neben den Daten- sowie Terminierungsoperationen der betreffenden Transaktionen auch Sperr- sowie Entsperroperationen.
- Ist s ein Schedule, so bezeichnet $DT(s)$ die **Projektion** von s auf sämtliche Aktionen des Typs r, w, a, c.

- Die Regeln sind für das einzuführende Sperrprotokoll relevant!

Sperren: Beispiel

- Seien $t_1 = r_1(x)\ w_1(y)$ und $t_2 = w_2(x)\ w_2(y)$ zwei Transaktionen und s_I, s_{II} die folgenden zwei Schedules:

s_I

#	t_1	t_2
1	$rl_1(x)$	
2	$r_1(x)$	
3	$ru_1(x)$	
4		$wl_2(x)$
5		$w_2(x)$
6		$wl_2(y)$
7		$w_2(y)$
8		$wu_2(x)$
9		$wu_2(y)$
10		c_2
11	$wl_1(y)$	
12	$w_1(y)$	
13	$wu_1(y)$	
14	c_1	

Dann gilt:

Beide Schedules erfüllen die Regeln (L1)-(L3)

$DT(s_I) = r_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1 \notin CSR$

$DT(s_{II}) = r_1(x)\ w_1(y)\ c_1\ w_2(x)\ w_2(y)\ c_2 = t_1\ t_2$

s_{II}

#	t_1	t_2
1	$rl_1(x)$	
2	$r_1(x)$	
3	$wl_1(y)$	
4	$w_1(y)$	
5	$ru_1(x)$	
6	$wu_1(y)$	
7	c_1	
8		$wl_2(x)$
9		$w_2(x)$
10		$wl_2(y)$
11		$w_2(y)$
12		$wu_2(x)$
13		$wu_2(y)$
14		c_2

Sperrprotokoll

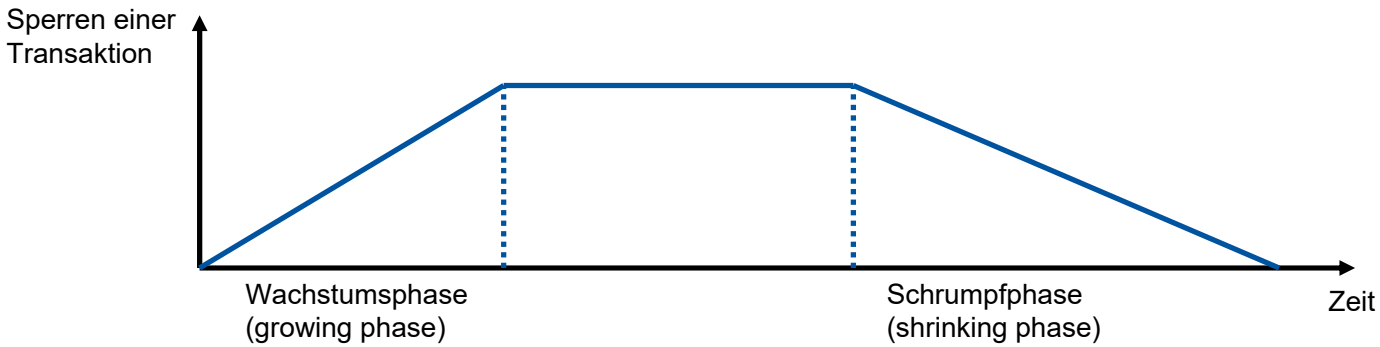
- Ein Scheduler arbeitet nach einem **Sperrprotokoll**, falls für jeden Ausgabe-Schedule s und jede Transaktion $t_i \in trans(s)$ gilt:
  - t_i erfüllt die Regeln (L1)-(L3), sowie
  - **(L4)** ist x durch t_i und t_j gesperrt, für $t_i, t_j \in trans(s)$ und $i \neq j$, so sind diese Sperren nicht in Konflikt, d.h. sie sind kompatibel gemäß der nachfolgend dargestellten Tabelle (d.h., mehrere Lesesperren sind zulässig):

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	-	-

  - ist $i = j$, so ist das Setzen von Schreib-/Lesesperren bei schon bestehenden Lese-/Schreibsperren derselben Transaktion t_i/t_j zulässig.

2-Phasen-Sperrprotokoll (*2PL*)

- Ein Sperrprotokoll ist **zweiphasig**, falls für jeden erzeugten Schedule s und jede Transaktion $t_i \in trans(s)$ gilt:
  - Nach der ersten ru_i/wu_i-Aktion folgt keine weitere rl_i/wl_i-Aktion
- In der ersten Phase einer Transaktion werden Sperren ausschließlich gesetzt, in der zweiten Phase werden Sperren ausschließlich freigegeben:



- Ein entsprechender Scheduler heißt $2PL$-Scheduler (2-Phase-Locking)

Zusammenfassung der Regeln für einen $2PL$-Scheduler:

- **(L1) Jede Lese-/Schreiboperation wird in der richtigen Reihenfolge ge- und entsperrt.**
- **(L2) Jedes von der Transaktion verwendete Datenobjekt wird gesperrt.**
- **(L3) Kein Entsperren ist überflüssig.**
- **(L4) Ist x in zwei Transaktionen gesperrt, sind so sind diese Sperren kompatibel.**
- **(2PL) In der ersten Phase einer Transaktion werden Sperren ausschließlich gesetzt, in der zweiten Phase werden Sperren ausschließlich freigegeben**

Korrektheitsbeweis: Prinzip

- Zu zeigen: $\varepsilon(2PL) \subset CSR$ (die vom Scheduler erzeugten Schedules sind konfliktserialisierbar)
- Beweis in zwei Schritten:

1. Formalisierung der Eigenschaften von Schedules $s \in \varepsilon(2PL)$
2. Zeigen, dass diese Eigenschaften Konfliktserialisierung zur Folge haben

Lemma 1: Eigenschaften von Schedules $s \in \varepsilon(2PL)$

Sei ein Schedule $s \in \varepsilon(2PL)$.
Dann gilt für jede Transaktion $t_i \in DT(s)$:

1. **Jede Lese-/Schreiboperation wird in korrekter Reihenfolge ge- und entsperrt:** Falls eine Aktion $p_i(x)$ in $CP\big(DT(s)\big)$ vorkommt, dann kommen auch $pl_i(x)$ und $pu_i(x)$ darin vor mit der folgenden Reihenfolge: $pl_i(x) < p_i(x) < pu_i(x)$.

2. **Sind zwei Aktionen in Konflikt, so sind die Sperren nicht gleichzeitig gesetzt:** Falls $t_j \in commit\big(DT(s)\big)$ mit $i \neq j$ und die Aktion $p_i(x)$ und $q_j(x)$ in $CP\big(DT(s)\big)$ in Konflikt, so gilt entweder $pu_i(x) <_s ql_j(x)$ oder $qu_j(x) <_s pl_i(x)$

3. **Alle Sperren werden zuerst gesetzt (1. Phase) und dann aufgehoben (2. Phase)**: Sind die Aktionen $p_i(x)$ und $q_i(y)$ in $CP\big(DT(s)\big)$ so gilt: $pl_i(x) <_s pu_i(y)$

4. **L2 und L3 gelten**

Beweis: Eigenschaften folgen aus den Definitionen: 1. folgt aus L1, 2. aus L4 3. folgt aus (2PL). 4. ist direkt die Definition

Korrektheitsbeweis: Schritt 1, Teil 2

Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

Sei ein Schedule $s \in \varepsilon(2PL)$ und G der Konfliktgraph von $CP(DT(s))$. Dann gilt:

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. G ist azyklisch.

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. **Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.**
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. G ist azyklisch.

Beweis:

1. Da (t_i, t_j) eine Kante in G, ist, gibt es eine Konfliktbeziehung $(p_i(x), q_i(x))$. Nach Lemma 1.1 gibt es Lock- und Unlock Operationen in $CP(s)$ mit $pl_i(x) <_s p_i(x) <_s pu_i(x)$.
2. Nach Lemma 1.2 (nicht Gleichzeitigkeit) folgt aus der Konfliktbeziehung $(p_i(x), q_j(x))$ entweder $pu_i(x) <_s ql_j(x)$ oder $qu_j(x) <_s pl_i(x)$.
   Aus $qu_j(x) <_s pl_i(x)$ folgt aber $q_j(x) <_s p_i(x)$, was im Widerspruch zu Lemma 1.1 und Definition Konfliktmenge steht.
   Also folgt $pu_i(x) <_s ql_j(x)$ und damit auch $p_i(x) <_s q_j(x)$ (Lemma 1.1)

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. **Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.**
3. G ist azyklisch.

Beweis durch Induktion über $k < n$:

1. $k = 2$: Siehe Beweis Lemma 2.1
2. $k > 2$: Wir nehmen an die Behauptung ist wahr für k mit $k < n$.
   Dann gibt es Aktionen und Datenobjekte $p_1(x), o_k(z) \in CP(DT(s))$ mit $pu_1(x) <_{CP(s)} ol_k(z)$ (B1).
   (t_k, t_{k+1}) ist eine Kante in G. Nach Lemma 2.1 gibt Aktionen und Datenobjekte $o'_k(x), q_{k+1}(y) \in CP(DT(s))$ mit $o'_k u(x) <_{CP(s)} ql_{k+1}(y)$ (B2)
   Nach Lemma 1.3 (2PL) gilt $ol_k(z) <_{CP(s)} o'u_k(y)$ (B3).
   Aus den drei Ungleichungen (B1), (B3) und (B2) folgt mittels Transitivität von "$<_{CP(s)}$": $\mathrm{pu}_1(\mathrm{x}) <_\mathrm{s} \mathrm{ql}_{\mathrm{k}+1}(\mathrm{y})$.
   qed.

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. **G ist azyklisch.**

Beweis durch Widerspruch:

Angenommen G enthält einen Zyklus: $(t_1, t_2, \ldots, t_n, t_1)$ mit $n > 1$.
Dann gibt es zwei Aktionen $p_1(x)$, $q_1(x)$ mit $pu_1(x) <_s ql_1(x)$ (mit Lemma 2.2)
Das steht im Widerspruch zu Lemma 1.3 (2PL Regel)

Korrektheitsbeweis: Schritt 2,

Theorem 1: Sicherheit: $\boldsymbol{\varepsilon(2PL) \subset CSR}$

$\subseteq$ folgt direkt aus Lemma 2.3 ($\boldsymbol{G}$ ist azyklisch.)

Echte Teilmenge folgt aus folgenden Schedule:

$$s = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$$

$s \in CSR$ (da $s \approx_s t_3t_1t_2$) aber $s \notin \epsilon(2PL)$ da aus

- $wu_1(x) <_s rl_2(x)$
- $ru_3(y) <_s wl_1(y)$
- $rl_2(x) <_s r_2(x)$
- $r_3(y) <_s ru_3(y)$
- $r_2(x) <_s r_3(y)$

folgt $wu_1(x) <_s wl_1(y)$, was der zwei-Phasen Eigenschaft widerspricht.

Konservatives/Statisches 2-Phasen-Sperrprotokoll (*C2PL*)

- **Wichtig:** Das C2PL Sperrprotokoll erweitert das 2PL Sperrprotokoll.
  - C2PL folgt allen Regeln des 2PL, und erweitert dieses um die Regeln auf dieser Folie.
- Alle Sperren einer Transaktion werden gesetzt, bevor die erste r/w-Aktion ausgeführt wird:

- Eigenschaften
  - Vorteil: Deadlocks vermieden, weil Transaktion wartet, bis alle Locks verfügbar sind - aber Risiko des „Verhungerns“, wenn viele Sperren gebraucht werden.
  - Nachteil: Alle r/w-Aktionen müssen im Vorhinein (BOT) bekannt sein - potentiell zu viele Sperren!

Strenge/Dynamische 2-Phasen-Sperrprotokoll (*S2PL*)

- **Wichtig:** Das S2PL Sperrprotokoll erweitert das 2PL Sperrprotokoll.
  - Das S2PL folgt allen Regeln des 2PL, und erweitert dieses um die Regeln auf dieser Folie.
- Alle Sperren einer Transaktion werden *sofort* nach der letzten Aktion aufgehoben:

- Es ist ausreichend, *Schreibsperren* bis nach der letzten $\mathrm{r/w}$-Operation zu halten.
- Ein $S2PL$-Scheduler ist *sicher*, d.h. es gilt sogar: $\boldsymbol{\varepsilon(S2PL) \subseteq CSR \cap ST}$.
- S2PL garantiert also Lösungen im „Idealbereich“ mit einem einfachen, effizienten und jederzeit anwendbaren Verfahren, das auch gut mit Fehlertoleranz (Recovery) kombinierbar ist, diese aber auch braucht (S2PL ist nicht Deadlock-frei).
- Werden **alle** Sperren bis nach der letzten r/w-Operation gehalten, spricht man von einem starken S2PL-Scheduler (SS2PL).

Ausblick: Verbesserungsmöglichkeiten für x2PL-Sperrprotokolle

- Falls die von x*2PL* gesperrten Objekte „groß“ sind, sind nur wenige Sperren zu verwalten, aber Konflikte zwischen Sperren sind häufiger.
- Falls die von x*2PL* gesperrten Objekte „klein“ sind, ist mehr Nebenläufigkeit möglich, aber die Verwaltungskosten für Sperren sind auch höher.
- Verallgemeinerung von x*2PL* zu hierarchischen Sperrobjekten:
  - Multiple Granularity Locking (MGL)
  - Tree Locking (TL)

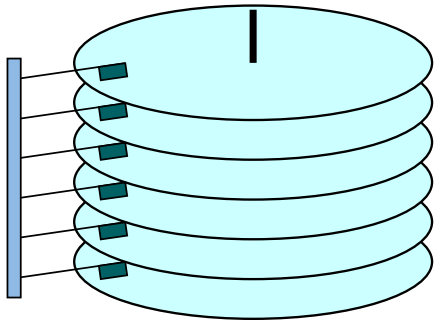
Transaktionsverwaltung

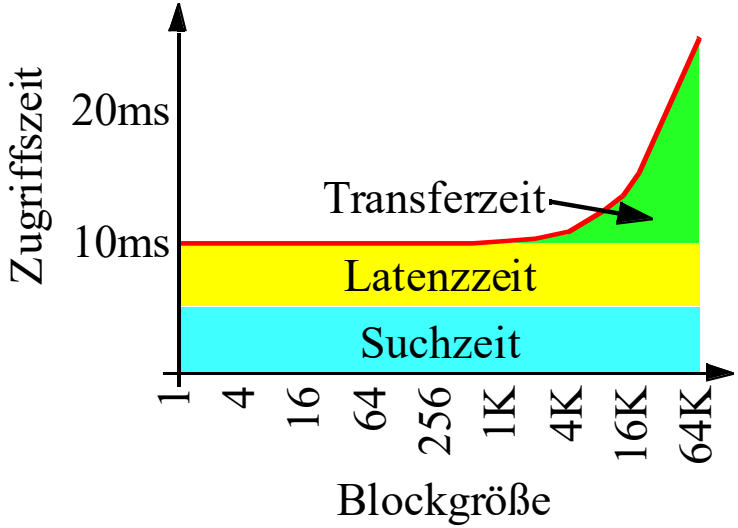
1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. **Fehlertoleranz (Recovery)**

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Technischer Kontext Speicherhierarchie

- Hauptspeicherzugriff (< 50 ns)
- Zugriffszeit bei Festplatten
  - Armpositionierung: Suchzeit (□ 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (□ 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)
- Blockorientierter Zugriff
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB
  - Kompatibel mit Paging-Mechanismus des Betriebssystems
  - Sequentielle Blockerarbeitung minimiert Armbewegung





Klassifikation von Fehlern

Ein DBMS soll die dauerhafte und konsistente Verfügbarkeit des Datenbestandes (Atomicity und Durability) sicherzustellen. Ein wichtiger Aspekt ist die Toleranz gegenüber Fehlern, die im laufenden Betrieb und auf den Datenträgern auftreten können.

- **Medienfehler**
  Verlust von permanenten Daten durch Zerstörung des Speichermediums, z.B. Plattencrash, Brand, Wasserschaden, etc.
- **Transaktionsfehler**
  Lokaler Fehler einer noch nicht abgeschlossenen Transaktion, z.B. aufgrund
  - Fehler und Abbruch des Anwendungsprogrammes (z.B. Division durch Null, …)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - expliziter Abbruch der Transaktion (**rollback**) durch den Benutzer
  - Konflikte mit nebenläufigen Transaktionen
- **Systemfehler**
  Verlust von Hauptspeicherinformation z.B. wegen Stromausfall, Ausfall der CPU, Absturz des Betriebssystems, etc. Die permanenten Speicher (Platten) sind nicht betroffen.

Technische Grundlagen

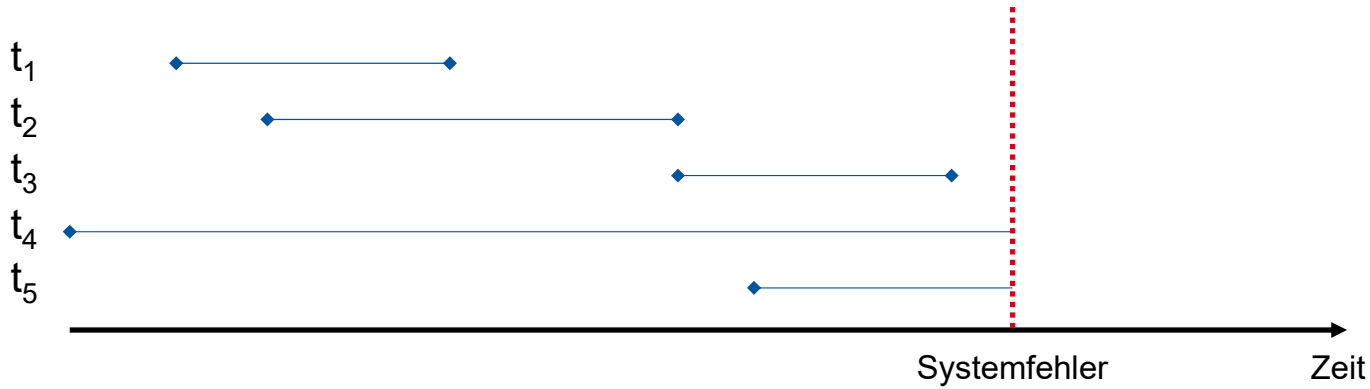
Es gibt unterschiedliche technische Maßnahmen zur Wiederherstellung:

- Für Medienfehler: Duplizierung (d.h. gezielte Redundanz) des Datenbankzustandes,
  - Bänder (oder andere Backup-Speicher): Kaltstart auf altem Datenbankzustand
  - Spiegelplatten erlauben Warmstart (Wiederanlauf bei laufendem Betrieb)
  - verteilte Rechenzentren an anderem Ort erhöht Unabhängigkeit (z.B. Erdbeben in Kalifornien, Katasterdaten Palästina)

- Bei Transaktions- und Systemfehlern
  - Mitprotokollierung der laufenden Schedules incl. BOT, COMMIT, ABORT) von Transaktionen in Logfiles
  - Bei allen Schreiboperationen müssen sowohl der alte als auch der neue Wert geloggt werden
  - Lese-/ Schreibobjekte sind meist physische Speicherblöcke (Paging)

- Logfiles
  - Logfiles entstehen sequentiell
  - eigenes Sekundärspeichermedium ermöglichen schnelles Lesen und Schreiben

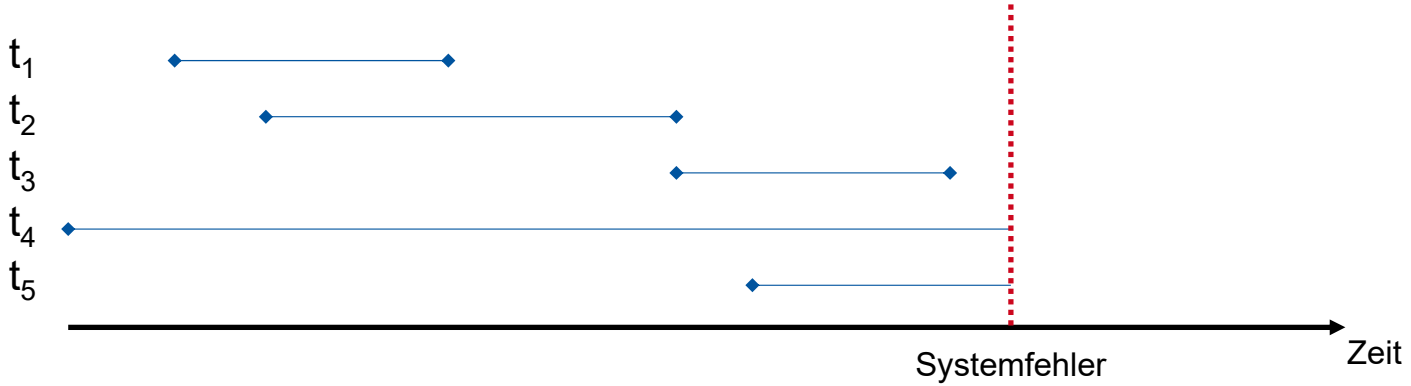
Szenario nach einem Fehler



Transaktionen werden in zwei Klassen eingestuft:

- Transaktionen, die vor dem Fehler bereits COMMITed waren. (t_1, t_2, t_3)
  Durability erfordert ein **REDO**, falls ihre Ergebnisse nicht im stabilen Speicher sind.
- Transaktionen, die zum Zeitpunkt des Fehlers noch aktiv waren (t_4, t_5).
  Atomarität erfordert ein **UNDO**, falls einige Ergebnisse bereits im stabilen Speicher sind.

Grundprinzipien eines Warmstart Recovery-Algorithmus (ARIES, C. Mohan, IBM)



- Ein korrekter Recovery-Algorithmus muss berücksichtigen, dass
  - auch für Logfiles das Zusammenspiel zwischen Hauptspeicher und stabilem Speicher gilt
  - jederzeit, sogar *während* der Recovery erneut Fehler auftreten können.
- **Write-Ahead Logging** bei Schreiboperationen:
  - Der alte Wert muss vor einer Schreiboperation geloggt werden (sicheres UNDO).
  - Der neue Wert muss ins stabile Log, bevor er in den stabilen Datenspeicher geschrieben wird, und spätestens vor dem COMMIT (sichere Durability).
  - Nach Absturz geht das **REDO** der abgeschlossenen Transaktionen, deren Ergebnisse noch nicht im stabilen Speicher stehen, dem **UNDO** der laufenden Transaktionen voraus.

Zusammenfassung Transaktionsmanagement

- Transaktionen sind Operationsfolgen auf Datenbanken
- Erfüllt ACID-Prinzips (trotz Mehrbenutzerbetrieb und Fehler)
- Korrekte Synchronisation und Fehlersicherheit werden formal mit Read-Write-Modells über zwei Korrektheitskriterien und Scheduler definiert:
  - Konfliktserialisierbarkeit aller aktiven und abgeschlossenen Transaktionen (Test: Zyklentest in Konfliktgraphen, Algorithmus z.B.: 2PL)
    - verhindert Lost Updates und Phantom-Problem
    - Fehlersicherheitskriterien wie Rücksetzbarkeit, Vermeidung kaskadierender Aborts und Striktheit (verhindern Dirty Read mit unterschiedlichen Trade-Offs zwischen Recoveryeffizienz und Einschränkungen der Verzahnung (z.B. S2PL)).

- Fehlertoleranz im laufenden Betrieb:
  - durch Redundanz in Form von Backups/Spiegelung (Medienfehler) und Operationslogs.
  - Mit **Write-Ahead**-Logging kombiniert mit **REDO/UNDO-Recovery** erlaubt Wiederanlauf auch wenn während des Recovery weitere Abstürze stattfinden.

Ausblick „Implementierung von Datenbanken“

Lange Transaktionen (auch **Workflows** genannt) erstrecken sich über Tage, Wochen oder Monate. Sie finden sich in vielen komplexen Designanwendungen. Dabei sollen auch „inkonsistente“ Zwischenzustände dem Recovery unterliegen:

- Definition eines *Sicherungspunktes*, auf den sich eine (noch aktive) Transaktion zurücksetzen lässt. Die Änderungen dürfen allerdings noch nicht endgültig festgeschrieben werden, da die Transaktion bis hin zur Bestätigung des COMMIT noch scheitern kann.
  SQL: **savepoint** <identifier>
- *Zurücksetzen* der aktiven Transaktion auf einen definierten Sicherungspunkt.
  SQL: **rollback to savepoint** <identifier> oder nur **rollback to** <identifier>

Weitere aktuelle Fragestellungen im heutigen Hochleistungsdatenmanagement:

- Vom ACID-Prinzip zum *CAP-Theorem für verteilte Datenbanken.*
- *Hauptspeicherdatenbanken* (SAP-HANA) verändern die Speicherhierarchie.
- *NoSQL*-Datenbanken unterstützen nicht-relationale Datenformate.
- *Blockchain*-Datenbanken betonen Nachvollziehbarkeit, *Data Spaces* Datensouveränität.

Nicht-Standard Datenmodelle und Datenbanken

1. Einführung
2. RDF und Semantic Web
3. Graphdatenbanken

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Nicht-Standard Datenmodelle und Datenbanken

1. **Einführung**
2. RDF und Semantic Web
3. Graphdatenbanken

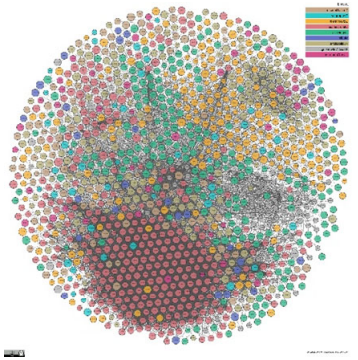
Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Strukturierte und Semistrukturierte Datenmodelle

- Unstrukturierte Daten
  - Daten folgen keinem Format, Schema oder Grammatik
  - Text, Bitströme auf Speichern, rohe Video- & Bilddaten, …

NoSQL
(Not Only SQL)

- Semi-Strukturierte Daten
  - Daten folgen einem flexiblen Format, optionale Felder, graphbasierte Daten
  - Webseiten, XML documente, JSON, RDF, …



- Strukturierte Daten
  - Daten folgen einem striktem Schema
  - relationale Datenbanken, Tabellen, Sensordaten, …

Relation "Mitarbeiter"

Personal-nummer	Name	Gehalt
1427	Meier	3217,33
8219	Schmidt	1425,87
2624	Müller	2438,21
⋮	⋮	⋮

Relation "Mitarbeiter in Abteilung"

Personalnummer	Abteilung
1427	3-1
8219	2-2
2624	3-1
⋮	⋮

Not only SQL

Relationale Datenbanken sind für große Datenmengen **oder** viele gleichzeitige Zugriffe optimiert. Das Internet hat den Bedarf an Big Data Lösungen in dem die Datenmenge groß **und** die Anzahl der Zugriffe hoch ist

- **Nicht-Relational:** Keine Tabellenstruktur, kein Schema, ermöglicht agile Verarbeitung großer Datenmengen.
- **Verschiedene Datenmodelle**
- **Skalierbarkeit:** Hoch skalierbar, geeignet für Verteilung auf viele Server.
- **Flexibilität:** Schema-los, bietet Flexibilität bei Datenspeicherung und -manipulation.
- **Leistung:** Schnelle Schreib- und Lesevorgänge, ideal für Echtzeitanwendungen.
- **Big Data & Echtzeit-Web-Apps:** Ideal für Big Data und schnelle Webdienste.
- **Konsistenzmodell:** Nutzt "eventuelle Konsistenz" für hohe Verfügbarkeit.

Skalierung des Datenzugriffs

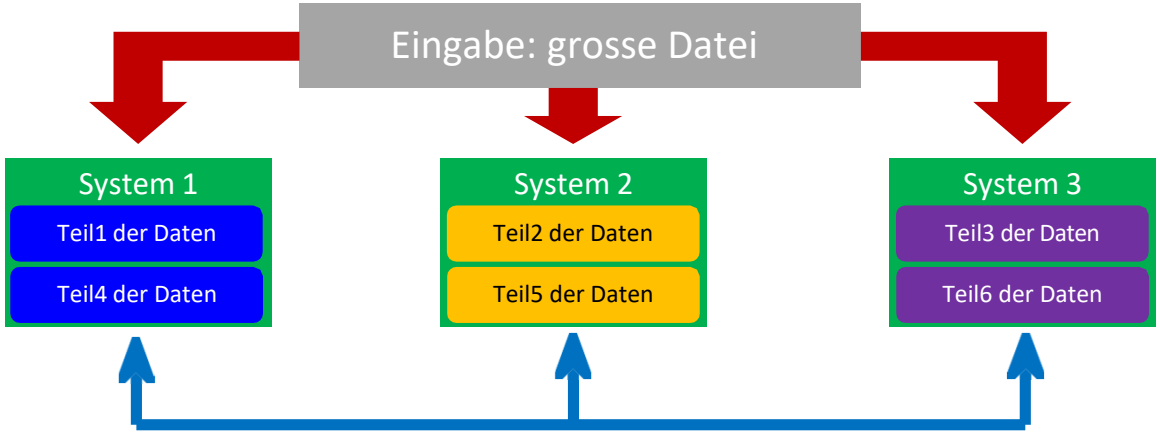
Traditionelle DBMS skalieren:

1. Vertikal:
   - Durch Aufrüsten der Hardware (z.B., schneller CPU, mehr RAM oder Sekundärspeicher)
   - Limitiert durch CPUs, RAM, Sekundärspeicher, Netzwerkbandbreite, die mit einer einzelnen Maschine möglich sind

2. Horizontal:
   - Durch hinzufügen von mehr Maschinen
   - Benötigt verteilte Datenbanken und wahrscheinlich Replikationen
     - Verteilt Tabellen auf verschiedene Maschinen
     - Verteilt Tupel von Tabellen auf verschiedene Maschinen
     - Verteilt Spalten von Tabellen auf verschiedene Maschinen
   - Limitiert durch Lese-Schreib Verhältnis (Synchronisation/Konflikte) sowie Kommunikationsoverhead

Sharding: Verteilen von Zeilen

Grössere Performance durch Verteilung von Zeilen der Tabellen auf mehrere DBMS („sharding“). Sharding ist nötig für die Parallelisierung.



Z.B., Teil 1,5 und 3 können parallel verarbeitet werden

Grenzen der Parallelisierung: das Amdahlsches Gesetz

- Das Amdahlsche Gesetz ist benannt nach Gene Amdahl, einem Pionier der Computerarchitektur.
- Es wird verwendet, um den maximalen Verbesserungsfaktor zu berechnen, der durch Parallelisierung einer Aufgabe erzielt werden kann.
- Annahmen:
  - Die sequentielle Ausführung eines Programmes braucht t Zeiteinheiten, und die parallele Ausführung auf *p* Prozessoren braucht T_p Zeiteinheiten.
  - s steht für den nicht-parallelisierbaren Teil des Programmes, daher $1 - s$ für den parallelisierbaren Teil.
- Dann ist der maximale Performanzgewinn gegeben durch Amdahls Formel: $\frac{t_1}{t_p} = \frac{t_1}{t_1 \times s + t_1 \times \frac{1-s}{p}} = \frac{1}{s + \frac{1-s}{p}}$

- Das Amdahlsche Gesetz stellt fest, dass die Verbesserung der Systemleistung durch Parallelisierung durch den Anteil der Aufgabe begrenzt ist, der nicht parallelisiert werden kann.

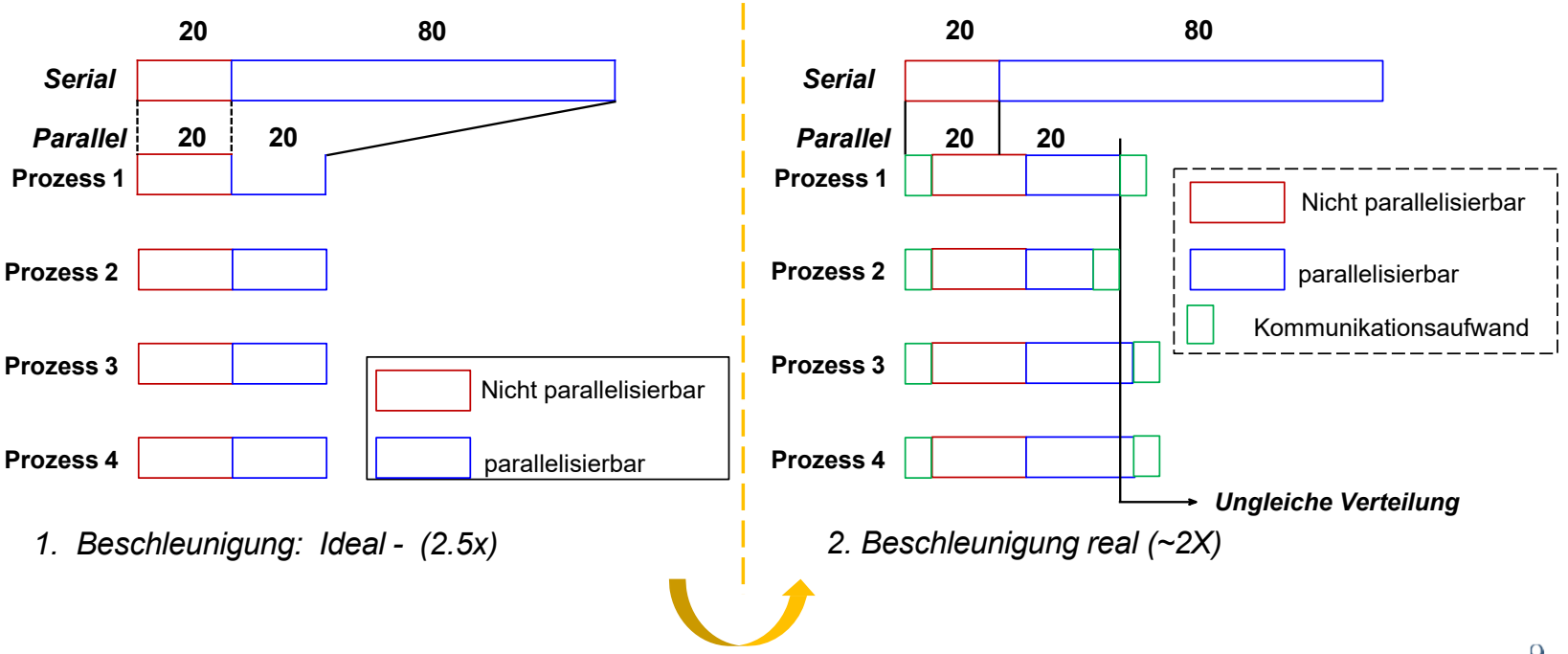
Beispiel

- Annahmen:
  - 60% einer Anfragebearbeitung kann parallelisiert werden
  - 6 Maschinen werden bei der Tupelselektion verwendet.
- Nach dem Amdahlschen Gesetz ist die maximale Beschleunigung: $\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0{,}4+\frac{0{,}6}{6}} = 2.0$
- Das Amdahlsche Gesetz stellt fest, dass die Verbesserung der Systemleistung durch Parallelisierung durch den Anteil der Aufgabe begrenzt ist, der nicht parallelisiert werden kann.

Trotz 6 Prozessoren bekommt man nur die doppelte Performanz!

Kommunikation und ungleiche Verteilungen

- Tatsächlich ist das Amdahlsche Gesetz zu sehr vereinfacht
- Kommunikationsaufwand und ungleiche Verteilung bei der Verarbeitung beeinflussen ebenfalls parallele Programme



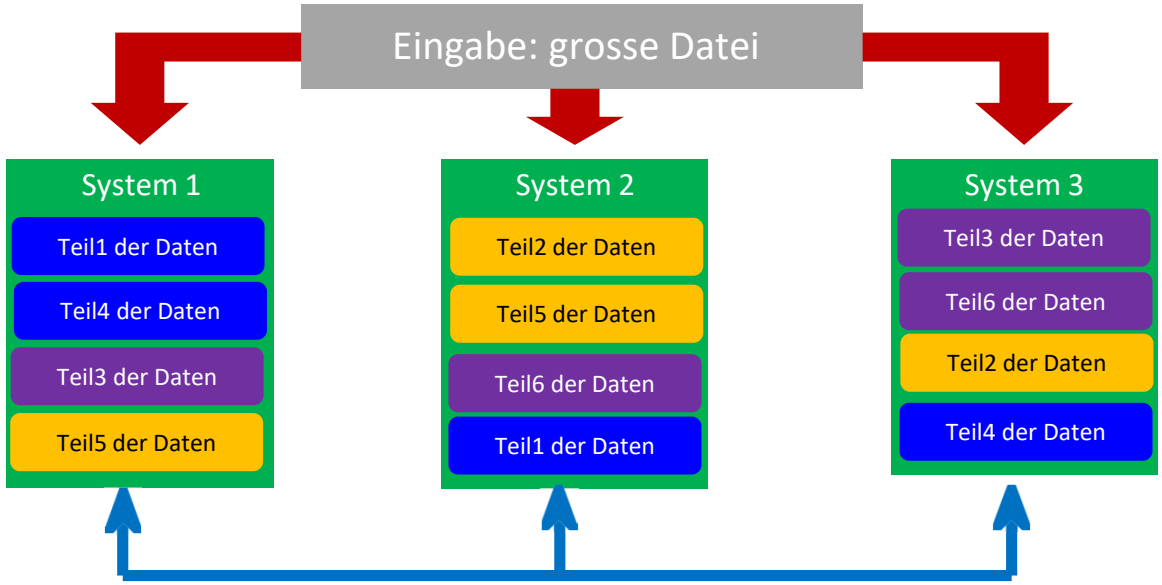
1. Beschleunigung: Ideal - (2.5x)

2. Beschleunigung real (~2X)

Sharding und Replizierung

Warum Datenreplizierung?

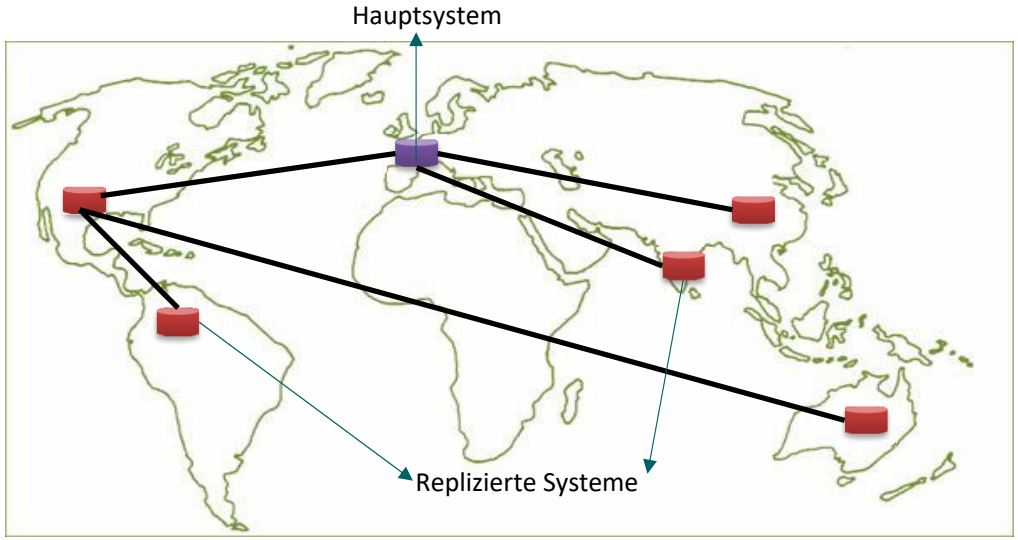
- Replizierung von Daten über Systeme hilft bei:
  - Vermeidung von Performanzflaschenhälsen
  - Erhöhung der Fehlertoleranz durch Vermeidung von Single Point of Failure



Sharding und Replizierung

Warum Datenreplizierung?

- Replizierung von Daten über Systeme hilft bei:
  - Vermeidung von Performanzflaschenhälsen
  - Erhöhung der Fehlertoleranz durch Vermeidung von Single Point of Failure
  - Verbessert auch die *Skalierbarkeit* und *Verfügbarkeit*

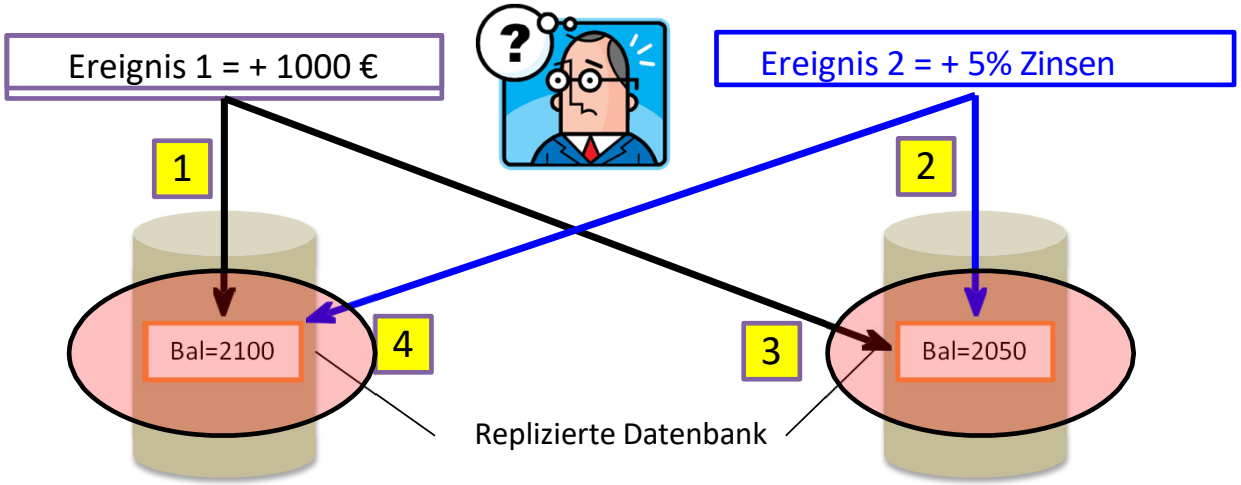


Aber…

Konsistenz wird schwierig.

Beispiel:

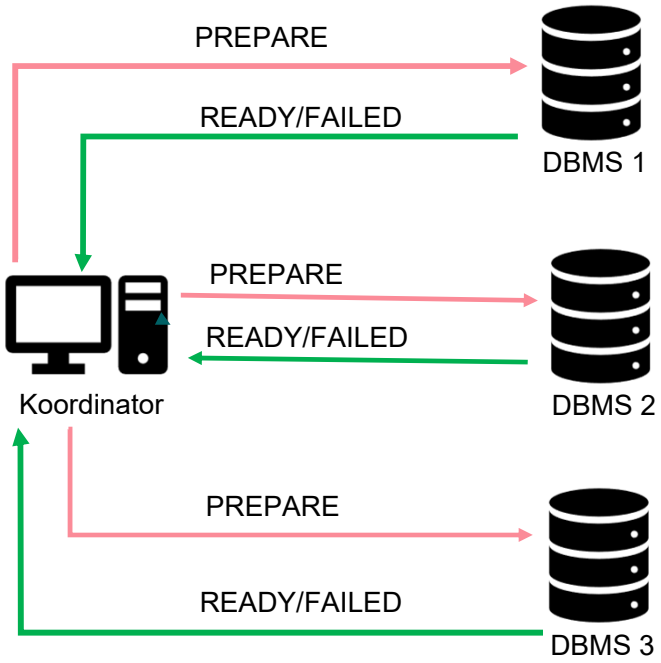
- Die Datenbank einer Bank sei über zwei Systeme repliziert
- Beibehaltung der Konsistenz der replizierten Daten wirft Fragen auf
- Scheduling nimmt serielle Ausführung an…



Zwei Phasen Commit (2PC) Protokoll

Das Zwei-Phasen Comm Protokoll kann zur Sicherstellung der Atomarität und Konsistenz verwendet werden, erhöht aber den seriellen Anteil der Transaktionen, und braucht gewöhnlich eine zentrale „Lock-Autorität“ oder Koordinator.

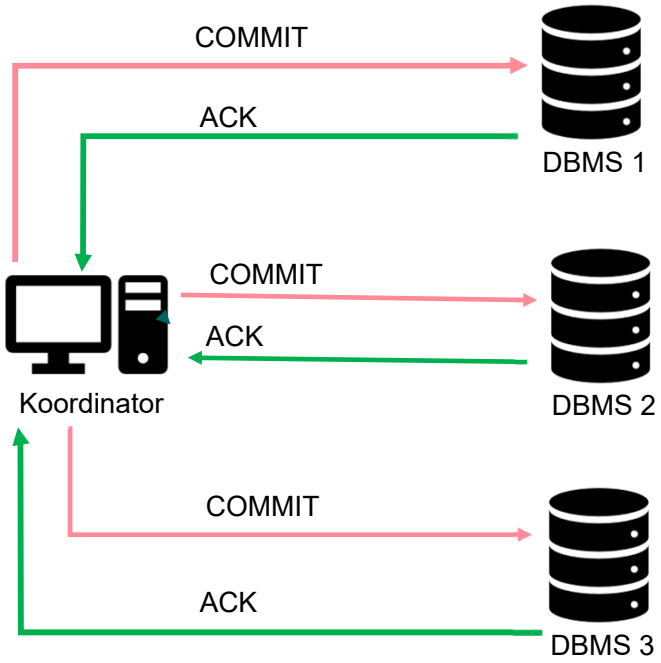
Phase 1:
PREPARE



Zwei Phasen Commit (2PC) Protokoll

Das Zwei-Phasen Commit Protokoll kann zur Sicherstellung der Atomarität und Konsistenz verwendet werden, erhöht aber den seriellen Anteil der Transaktionen, und braucht gewöhnlich eine zentrale „Lock-Autorität“ oder Koordinator.

Phase 2:
COMMIT



Strikte Konsistenz limitiert Skalierbarkeit!

Das CAP Theorem

Das CAP-Theorem, auch bekannt als Brewers Theorem, ist ein grundlegendes Prinzip, das die Einschränkungen von verteilten Datenbanksystemen beschreibt. CAP steht für Konsistenz (Consistency), Verfügbarkeit (Availability) und Partitionstoleranz (Partition Tolerance). Die drei Eigenschaften sind:

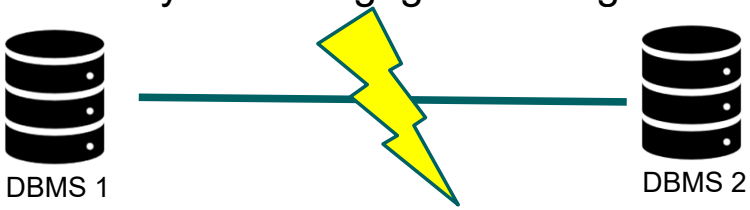
1. Konsistenz (**C**onsistency): Jeder Knoten sieht zu jedem gegebenen Zeitpunkt immer die gleichen Daten (d.h., strikte Konsistenz)
2. Verfügbarkeit (**A**vailability): Weiterbetrieb, auch wenn Knoten in einem Cluster abstürzen oder einige Hardware- oder Softwareteile aufgrund von Upgrades ausfallen
3. Partitionstoleranz (**P**artition Tolerance) : Weiterbetrieb bei Vorhandensein von Netzwerkpartitionen (Unterbrechungen in der Konnektivität)

CAP-Theorem: Jede verteilte Datenbank mit geteilten Daten kann **höchstens zwei der drei** wünschenswerten Eigenschaften, C, A oder P, garantieren.

Gilbert, S., & Lynch, N. (2002, June). Brewers Conjunction and the Feasability of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News , p. 33(2).

CAP Beispiele

Angenommen wir haben zwei System auf gegenüberliegen Seiten einer Netzwerkpartition:



- Verfügbarkeit und Partitionstoleranz gibt die Konsistenz auf
- Konsistenz und Partitionstoleranz heißt die Systeme sind bei Netzwerkpartition nicht verfügbar, da Daten nicht synchronisiert werden können.
- Konsistenz und Verfügbarkeit ist nur möglich wenn es keine Netzwerkpartition gibt, da die Partitionstoleranz aufgegeben wird.

Grosse Datenbanken

- 24/7 Verfügbarkeit ist bei Anwendungen, wie sie Unternehmen wie Google und Amazon haben, wichtig
  - Jede Nicht-Verfügbarkeit bedeutet Umsatzverluste
- Bei Skalierung einer Datenbank auf tausende von Systemen wächst die Wahrscheinlichkeit von Ausfällen oder Netzwerkproblemen stark an
- Um Verfügbarkeit und Partitionstoleranz zu garantieren, muss daher nach dem CAP-Theorem die (strikte) Konsistenz geopfert werden.

Konsistenzkompromisse

- Balance bzgl. zwischen Konsistenz vs. Verfügbarkeit und Skalierbarkeit
- Was „ausreichende“ Konsistenz ist, hängt von der Anwendung ab

Einfacher zu implementieren und potentiell hohe Performanz

Im Allgemeinen schwer zu implementieren und verringert die Performanz

Performanz = Durchsatzes (wie viele Transaktionen pro Sekunde das System verwalten kann) und der Latenz (wie lange Sie warten müssen)

BASE

- CAP-Theorem zeigt das strikte Konsistenz, Verfügbarkeit und Partitionstoleranz gleichzeitig nicht garantiert warden dann.
- Daher: Datenbanken mit Relaxierung der ACID-Eigenschaftenen
- Insbesondere Anwendung der **BASE** Eigenschaften:
  - **B**asically **A**vailable: das System garantiert Verfügbarkeit
  - **S**oft State: der Zustand des Systems verändert sich über die Zeit und kann für kurze Zeitintervalle inkonsistenz sein
  - **E**ventually Consistent: das System wird irgendwann konsistent

Dan Pritchett: BASE: An Acid Alternative. ACM Queue 6(3): 48-55 (2008)
Werner Vogels: Eventually consistent. Commun. ACM 52(1): 40-44 (2009)

Eventually Consistent

- Eine Datenbank hat die Eigenschaft *eventuell konsistent* zu sein, wenn:
  - Alle Kopien graduell auf einen einzigen konsistenten Zustand konvergieren, falls es in einem spezifizierten Zeitintervall keine Änderungen gibt.

NoSQL (=Not Only SQL) Databases

Eigenschaften von NoSQL Datenbanken:

- Keine striktes Schema
- Keine strikte Einhaltung der ACID Prinzipien
- Verfügbarkeit wichtiger als (strikte) Konsistenz
- Konsistenz wird irgendwann erreicht, wenn es keine Updates gibt

Auch: nur einige dieser Eigenschaften

Verschiedene Klassen von NoSQL Datenbanken

Taxonomie von NoSQL Datenbanken:

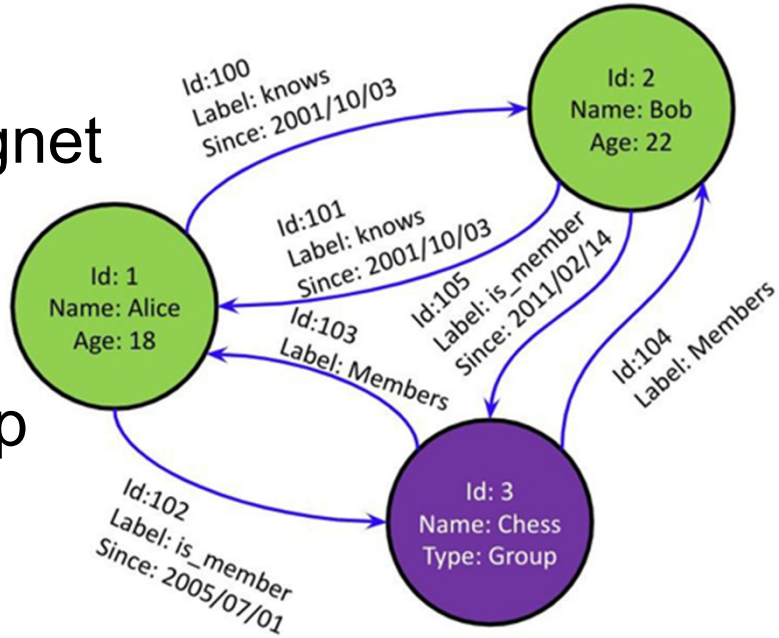
- Key/Value-Datenbanken
- Dokumentdatenbanken
- Graph-Datenbanken
- spaltenorientierte Datenbank

Dokumentendankenbanken

- Dokumente und Texte werden in einem etabliertem Format gespeicher (z.B., XML, JSON, PDF, Office)
- Dokumente können indexiert werden
  - Bessere Performanz als Dateisysteme
- Beispiele: MongoDB, CouchDB

Graphdatenbanken

- Daten werden durch Knoten und Kanten dargestellt
  - Knoten sind wie ER „Entities“
  - Kanten sind ER „Relationen“
- Insbesondere für heterogene Daten geeignet
- Beispiele: Neo4J, RDF, Apache Tinkerpop



Key-Value Datenbanken

- Schlüssel werden auf (möglicherweise komplexe) Werte abgebildet
- Schlüssel können gehashed und verteilt werden
- Solche Datenbanken unterstützen normalerweise CRUD (create, read, update, delete) Operationen
  - Keine Joins oder Aggegateoperationen
- Beispiele: Amazon DynamoDB oder Apache

spaltenorientierte Datenbank

- Hybrid zwischen Key-Value Datenbanken und relationalen Datenbanken
  - Werte werden in Gruppen von Null oder mehr Spalte für Spalte gespeichert.
  - Zeilenorientiert:
    1,Schmidt,Josef,40000; 2,Müller,Maria,50000; 3,Meier,Julia,44000;
  - Spaltenorientiert:
    1,2,3;Schmidt,Müller,Meier;Josef,Maria,Julia;40000,50000,44000;
- Beispiel: SAP Hana

Zusammenfassung

- Datenbanken können nach strukturierte und nicht-strukturierte klassifiziert werden
- Datenbanken können horizontal oder vertikal skaliert werden
- Strikte Konsistenz beschränkt Skalierbarkeit
- CAP Theorem motiviert BASE Eigenschaften
- Verschiedene Typen von NoSQL Datenbanken
  - Im folgenden: Fokus auf Graphdatenbanken

Nicht-Standard Datenmodelle und Datenbanken

1. Einführung
2. **RDF und Semantic Web**
3. Graphdatenbanken

Informatik 5 - Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Das RDF Datenmodell

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Informationsdarstellung in Text

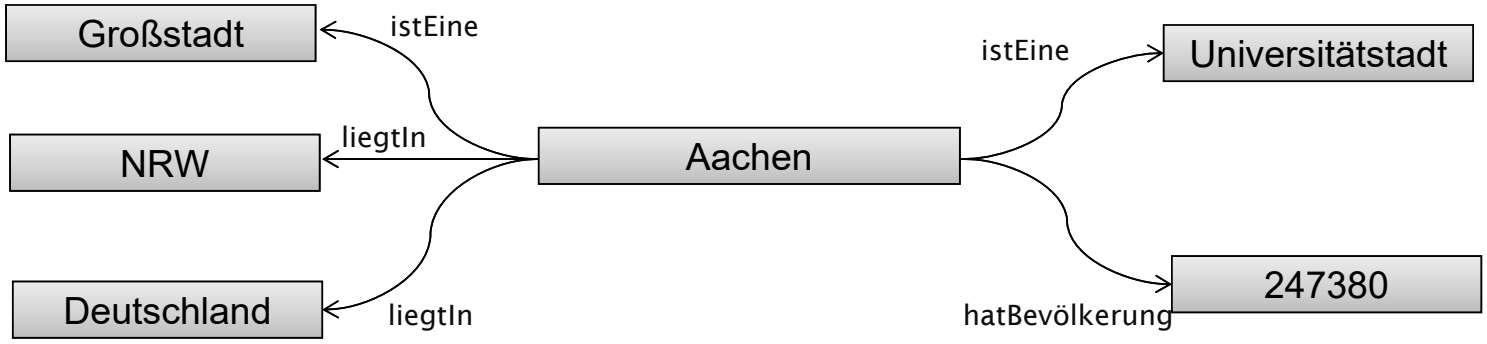
Aachen ist eine Großstadt in Nordrhein-Westfalen, Deutschland. Aachen ist außerdem eine Universitätsstadt. Aachen hat eine Bevölkerung von 247380 Menschen, […]…

- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.
- …

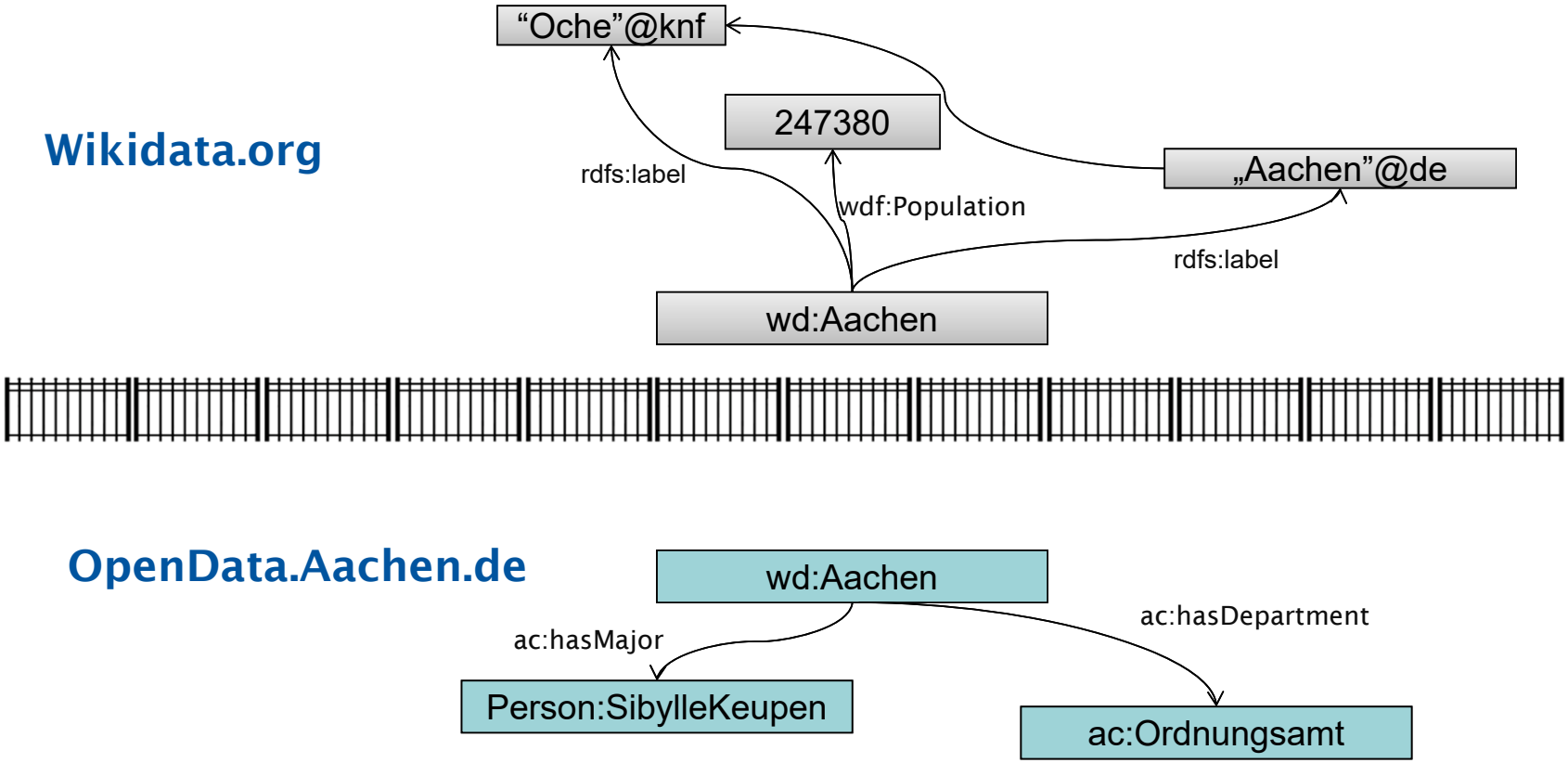
Informationsdarstellung in Tripeln

- Vereinfachte Darstellung der Informationen folgt einer durchgängigen **Subjekt** - **Prädikat** - **Objekt** – Struktur, auch Tripel genannt
- Tripel: Beziehung einer Ressource zu anderen Ressourcen und Werten

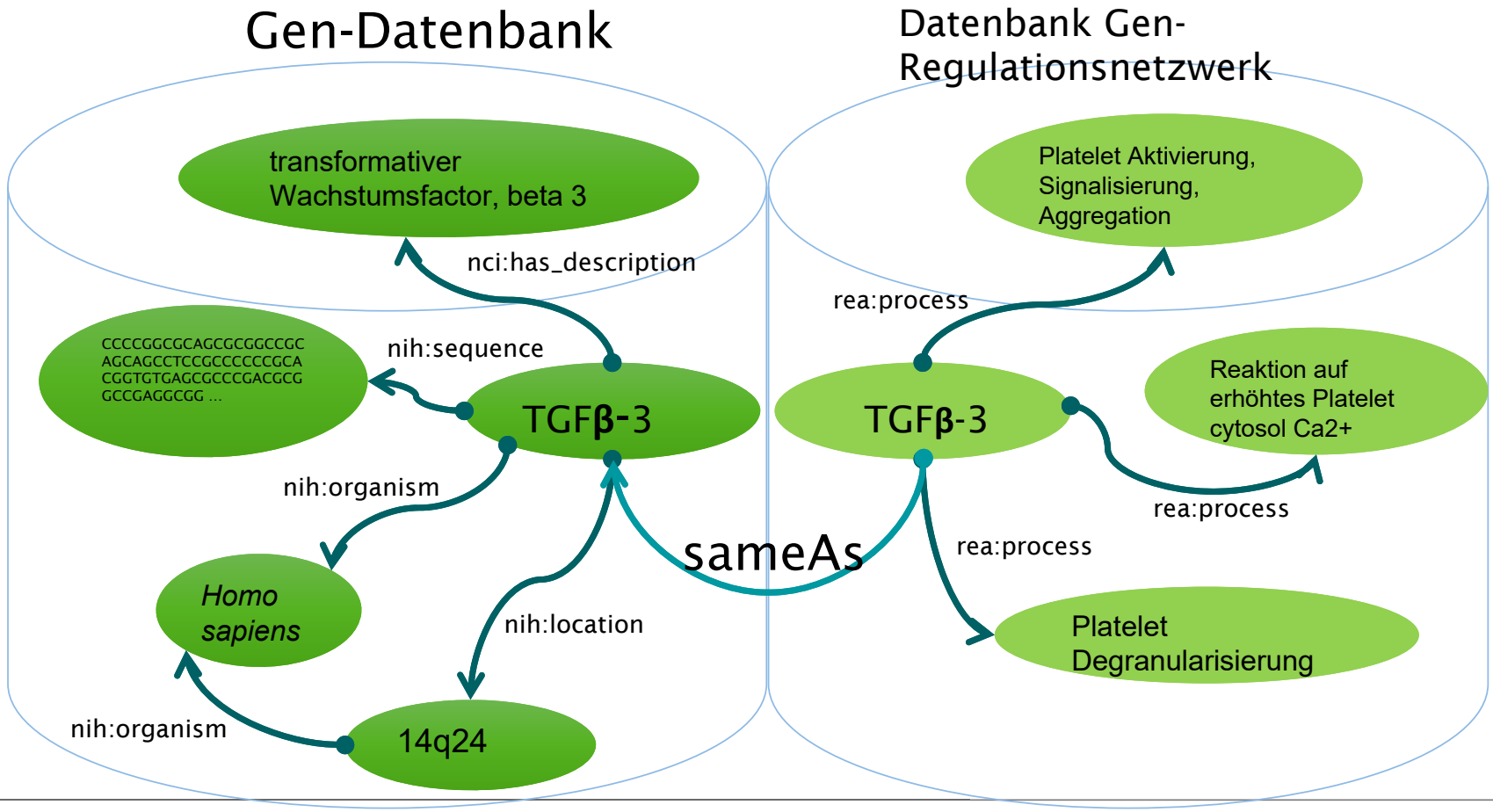
- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.



Wieso Graphen?



Ein Beispiel aus der Biologie



Standards für das Web

1. **RDF – Resource Description Framework**
   - Ein Graph-basiertes Datenformat: Knoten und Kanten
   - Objekte eindeutig identifiziert durch URIs
   - Information wird verknüpft

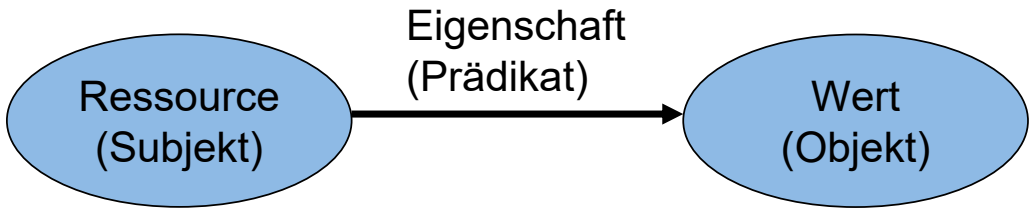
2. **Vokabulare und Ontologien**
   - Ermöglicht das gemeinsame Verständnis für eine Domäne
   - Erlauben das Organisieren von Wissen in einem maschinen-lesbaren Format
   - Gibt Daten einen auswertbare Bedeutung

RDF Überblick

- RDF = Resource Description Framework
  - W3C Empfehlung seit 1998
    - http://www.w3.org/RDF
  - Version 1.1 seit 2014
    - http://www.w3.org
- RDF ist ein Datenmodell
  - Zuerst nur für Metadaten auf Web-Seiten verwendet, dann generalisiert
  - Drückt strukturierte Informationen aus
  - Universales Format zum automatisierten Datenaustausch
- Graph-basierte Datenstruktur
  - Mit Knoten und Kanten

Der RDF Kern

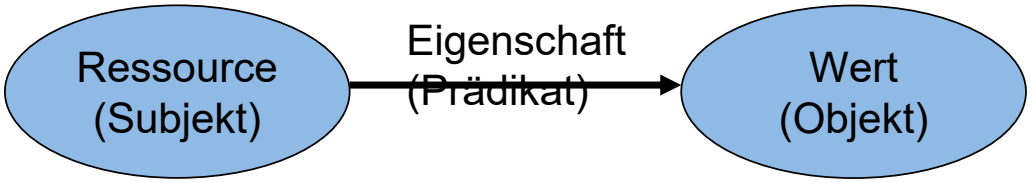
- RDF baut auf den Beziehungen zwischen Ressourcen auf
- Tripel der Form (s, p, o) sind der fundamentale Baustein von RDF
  - **Subjekt**
  - **Prädikat**
  - **Objekt**
- RDF verwendet Identifikatoren vom Web (URIs) um Ressourcen zu identifizieren



Das Subjekt hat eine ***Eigenschaft*** mit einem **Wert**

RDF Tripel

- RDF Tripel:
  - Subjekt: Eine Ressource kann eine URI oder ein Blank Node sein.
  - Prädikat: Eine URI welche eine Eigenschaft der Ressource identifiziert.
  - Objekt: Der Wert einer Eigenschaft der Ressource. Dies kann eine URI, ein Literal oder ein Blank Node sein.



Das Subjekt hat eine ***Eigenschaft*** mit einem **Wert**

Die Teile eines RDF Graphen

- URIs
  - Verwendet um Ressourcen eindeutig zu referenzieren
- Literale
  - Beschreiben Wertigkeiten
- „Blank Nodes“
  - Erlauben die existentielle Quantifizierung der Eigenschaften einer Entität ohne ihr einen Namen zu geben
- RDF Graphen müssen **nicht zusammenhängend** sein

Was sind URIs ?

- URI = Uniform Resource Identifier
  https://tools.ietf.org/html/rfc3986
- Weltweiter, eindeutiger Identifikator für Ressourcen
- Jedes Objekt kann eine Ressource sein, wenn es eine eindeutige Identität hat
  - Beispiele: Bücher, Orte, Personen, Beziehungen zwischen diesen Dingen, oder abstrakte Konzepte
- Eindeutige Kennzeichner werden schon für bestimmte Domänen verwendet, z.B. ISBN für Bücher oder Steuer-ID für Personen in Deutschland
- URIs sind eine Erweiterung der URL
  - Nicht jede URI gehört zu einer Webseite, aber meistens werden URLs als URIs für Webseiten verwendet.

Syntax von URIs

- Protokoll ":" Hierarchie ["?" Anfrage] ["#" Fragment]

- Beispiele:
  - `http://en.wikipedia.org/w/index.php?search=rdf`
  - `http://en.wikipedia.org/wiki/Resource_Description_Framework#Examples`
  - `urn:example:animal:ferret:nose`

Literale

- Verwendet um Datenwerte auszudrücken
- Durch Strings repräsentiert
- Interpretation des Literals anhand des **Datentyps**
- Literale können niemals der Ursprung einer Kante in einem RDF Graph sein (Literale können nicht Subjekt eines Tripels sein)
- Literale können nicht einer Kante zugeordnet sein

- **Language Tags:** Optionale Auskunft über die Sprache eines Strings, kann **nicht** zusammen mit einem Datentyp verwendet werden. Beispiel: `"Aachen"@DE`

Datentypen für Literale

- Literale ohne Datentypen werden wie Strings behandelt
  - Beispiel mit „kleiner als”: "02" < "100" < "11" < " 2"
- Datentypen erlauben eine semantische Interpretation
- Datentypen werden durch frei wählbare URIs identifiziert
- Am gebräuchlichsten sind XML Schema Datentypen (XSD)
- Syntax: `"Datenwert"^^<Datentyp-URI>`
- Es gibt nur zwei vordefinierte Datentypen in RDF:
  - *rdf:HTML* und *rdf:XMLLiteral*

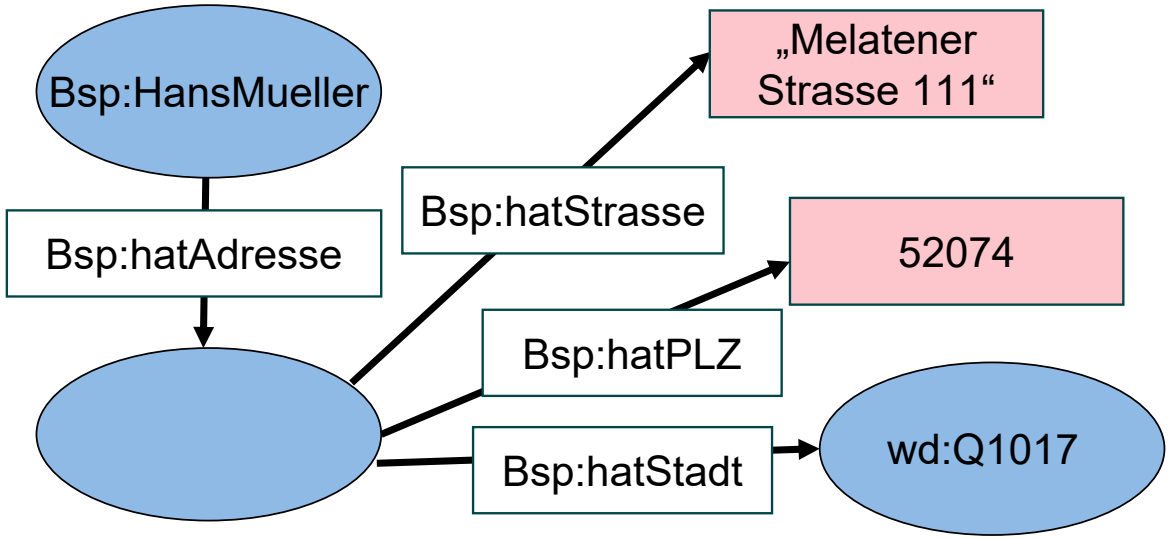
- Beispiel:
  - `"123"^^http://www.w3.org/2001/XMLSchema#int`

Blank Nodes

- Ein Blank Node hat keinen globalen Identifikator
  - Idee: „Ein Blank Node ist ein Platzhalter“
  - Jeder Blank Node hat eine eindeutige Identität innerhalb eines Graphen
  - Es kann geprüft werden ob zwei Blank Nodes gleich sind
  - Blank Nodes werden als “Existentielle Quantifikatoren” verwendet

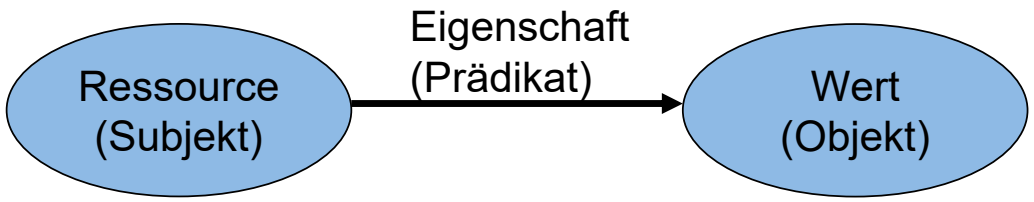
Beispiel für die Verwendung von Blank Nodes

- “Hans Mueller hat die Addresse „Melatener Strasse 111, 52074 Aachen”



Übersicht: Regeln für RDF Tripel

	URI	Literal	Blank Node
Subjekt	X		X
Prädikat	X		
Objekt	X	X	X



Serialisierung von RDF Graphen

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Formate um RDF Graphen zu Serialisieren

Frage: Wie kann man einen Graphen in einer Datei speichern?

- **Turtle: Text-Format mit dem Ziel der besseren Lesbarkeit durch Menschen (wird in dieser Vorlesung verwendet)**
- **JSON-LD:** W3C Proposed Recommendation. Format um RDF als JSON zu serialisieren, bzw existierendes JSON als RDF zu interpretieren. Empfohlen von Google.
- **N-Triples**: Text-Format welches sich sehr einfach parsen lässt
- **Notation 3 (N3):** alternatives Text-Format mit Nicht-Standard Features die RDF erweitern
- **RDF/XML:** das erste offizielle Format um RDF als XML zu serialisieren
- **RDFa:** Mechanismus um RDF in (X)HTML einzubetten

[1] https://www.youtube.com/watch?v=cSF48tbsjJw&feature=youtu.be&t=1353

Turtle Syntax 1

- Turtle steht für “Terse RDF Triple Language”
- Format um RDF Triples als Strings darzustellen
- URIs immer in <>-Klammern:
  `<http://www.wikidata.org/entity/Q1017>`
- Literale in doppelten Anführungszeichen:
  - `"Aachen"@DE`
  - `"51.33332"^^xsd:float`
  - Integer werden als Literale mit dem Integer Datentyp interpretiert: `32` als `"32"^^xsd:int`
- Alle Tripel werden als Sätze von Subjekt, Prädikat und Objekt dargestellt, gefolgt von einem Punkt:
  - `<http://www.wikidata.org/entity/Q1017> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .`
- Leerzeichen und Zeilenumbrüche werden außerhalb von Identifikatoren ignoriert

Turtle Syntax 2

- Abkürzungen können für Namespaces definiert werden:
  - `@prefix abbr ':' <URI> .`
  - Beispiel: `@prefix wd: <http://www.wikidata.org/entity/> .`

- Ohne Namespaces:

```
<http://www.wikidata.org/entity/Q1017> <http://www.w3.org/2000/01/rdf-
schema#label> "Aachen"@de .
```

- Mit Namespaces:

```
@prefix wd:  < http://www.wikidata.org/entity/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
   wd:Q1017 rdfs:label "Aachen"@de .
```

Turtle Syntax 3

- Tripel mit dem selben Subjekt können zusammengefasst werden:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: http://www.w3.org/2000/01/rdf-schema#
@prefix wd: < http://www.wikidata.org/entity/>
@prefix wdt <http://www.wikidata.org/prop/direct/>

wd:Q1017    rdfs:label    "Aachen"@de ;
            wdt:P6        wd:Q1893149 .
```

- Tripel mit dem gleichen Subjekt und Prädikat können auch zusammengefasst werden:

```
wd:Q1017 rdfs:label „Aachen"@de, „Aix-la-Chapelle"@fr;
         wdt:P6     wd:Q99688800 .
```

Turtle: Vorteile und Nachteile

- Vorteile:
  - Kurzgefasst, daher effiziente Speicherung möglich
  - Einfach für Menschen zu lesen
  - Sehr nah am RDF Datenmodell
- Nachteile:
  - Geringe Unterstützung in Software-Werkzeugen
  - Geringe Verbreitung außerhalb des Semantic Webs

Siehe: https://www.w3.org/TR/turtle/

Beispiele zur Verwendung von RDF Daten

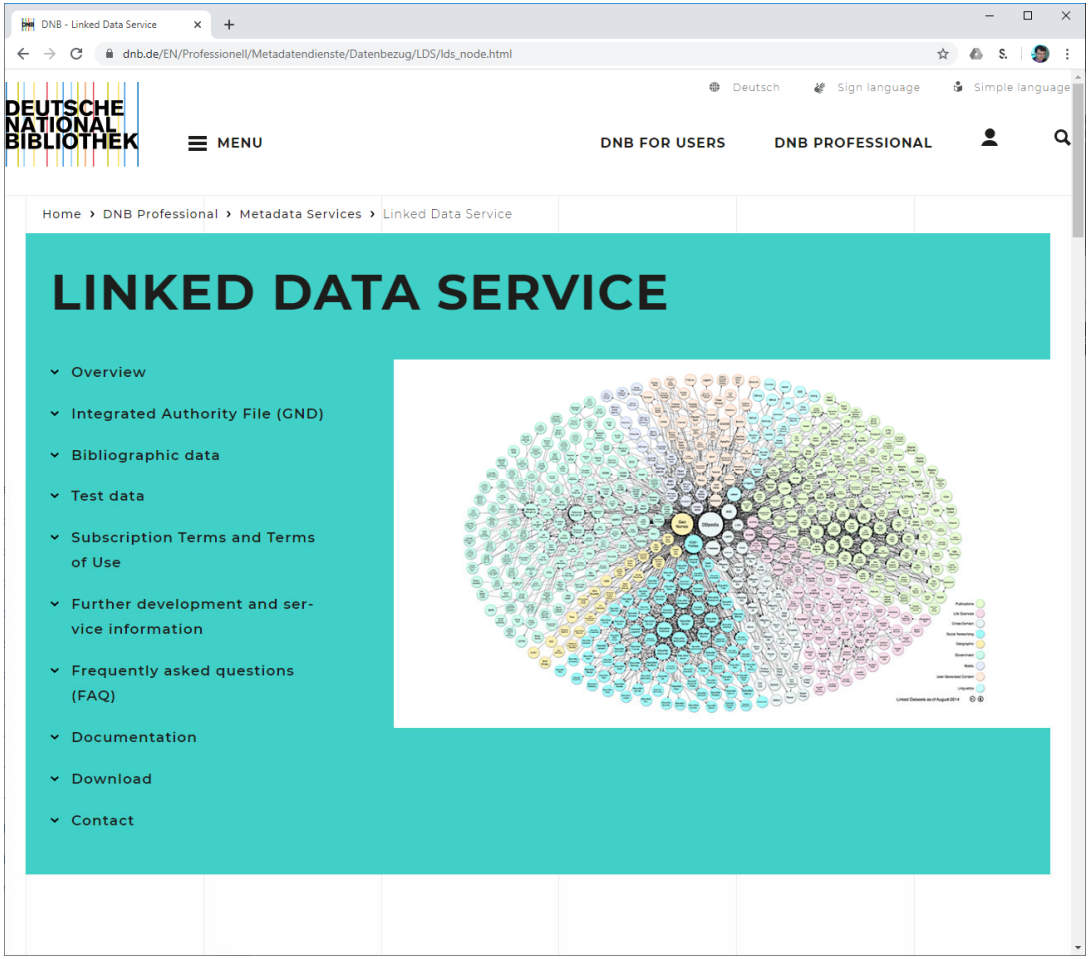
Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

RDF in freier Wildbahn: Schema.org

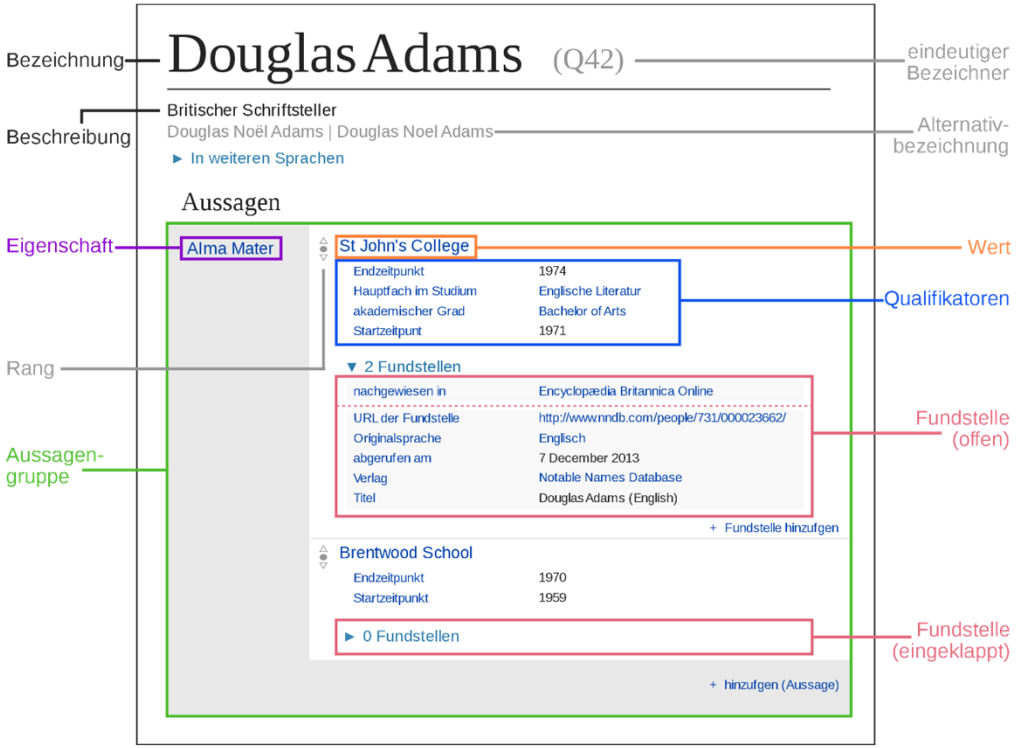
- 2010 von Google, Yahoo!, Microsoft & Yandex gestartet
- Ziele:
  - Vokabulare welches von allen Suchmaschinen unterstützt wird
  - Vereinfacht den Job des Webmasters und der Suchmaschinen-Optimierung
- Vokabular um Web-Seiten zu annotieren
- Kann Entitäten von den folgenden Typen annotieren:
  - „Creative“
  - „Works“
  - „Events“
  - „Organisations“
  - „Persons“
  - „Places“
  - „Products“
  - ...

Screenshot von Linked Data Service DNB



Wikidata

- frei bearbeitbaren Wissensdatenbank mit dem Wikipedia zu unterstützen.
- von Wikimedia Deutschland gestartet als gemeinsame Quelle für Daten
- Globale Datenbank für Identifier
- 82,4 Millionen Datenobjekte vorhanden (Stand März 2020), wird u.a. von Siri und IBM Watson verwendet [1][2]



[1] https://www.wired.com/story/inside-the-alexa-friendly-world-of-wikidata/
[2] https://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext

SPARQL: Anfrage-Sprache für RDF Graphen

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

SPARQL: Anfrage-Sprache für RDF Graphen

- SPARQL steht für “SPARQL Protocol and RDF Query Language” (gesprochen: “Sparkl”)
- https://www.w3.org/TR/sparql11-query/

Struktur einer SPARQL Anfrage

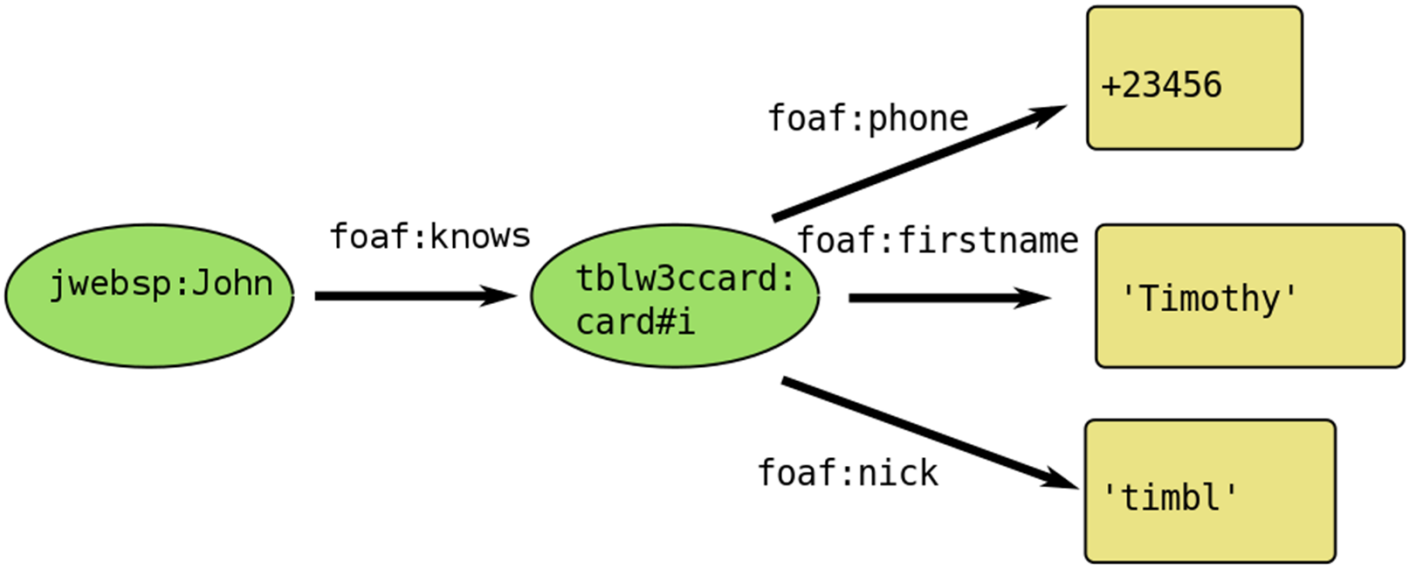
```
# prefix declarations
PREFIX ex: <http://example.com/resources/>
....
# query type# projection  # dataset definition
SELECT    ?x ?y           FROM ...

# graph pattern
WHERE {
    ?x a ?y
}

# query modifiers
ORDER BY ?y
```

Bestandteile einer SPARQL Anfrage:

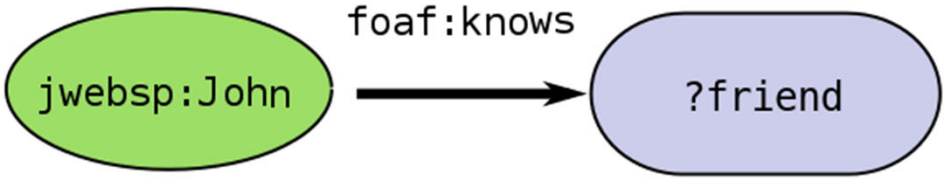
- Definition von Namespaces
- Anfrage Klausel: Projektion
  - Vier Möglichkeiten: SELECT, ASK, CONSTRUCT, DESCRIBE
- WHERE Klausel: Selektion durch ein Graph-Muster
- Anfrage Modifikatoren

Ein kleiner Beispiel RDF Graph

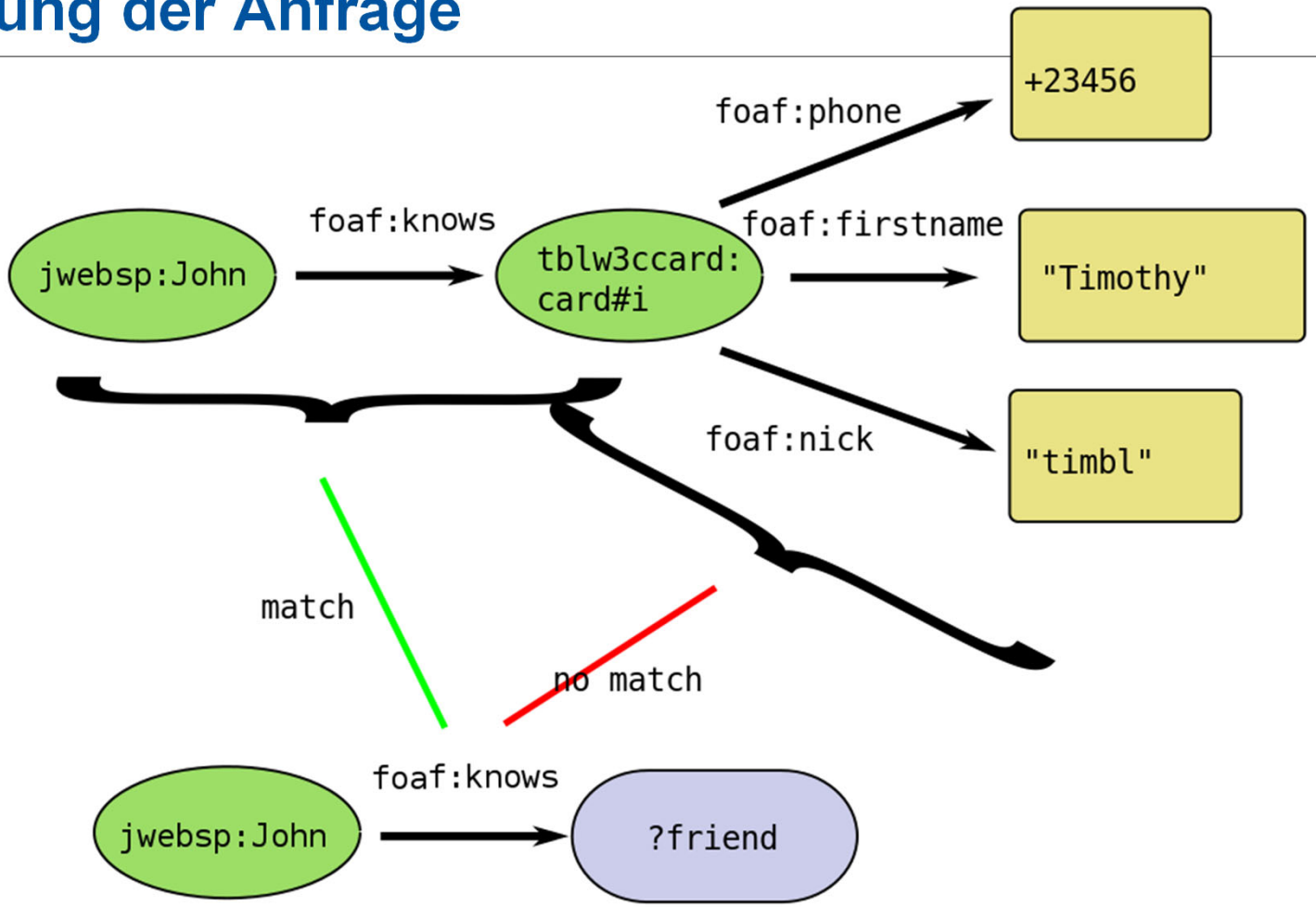
```
@prefix wbsp: <http://mywebspace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```



Eine sehr einfache Anfrage



Beantwortung der Anfrage



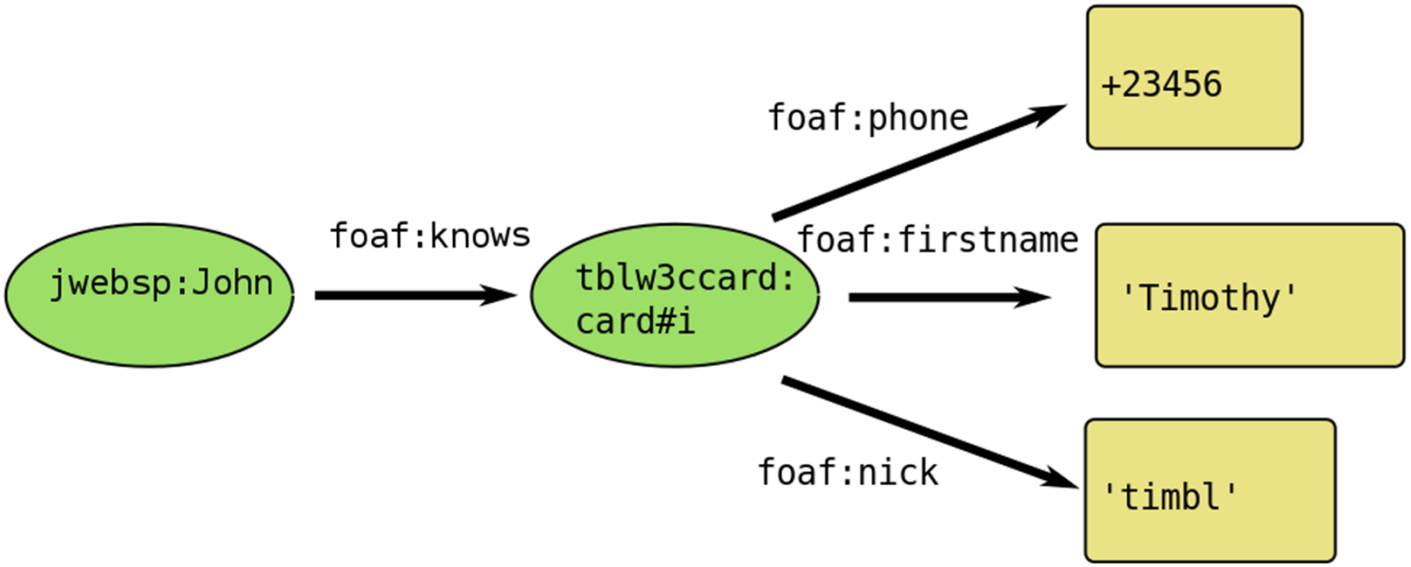
Eine etwas komplexere Anfrage

SPARQL für die Anfrage

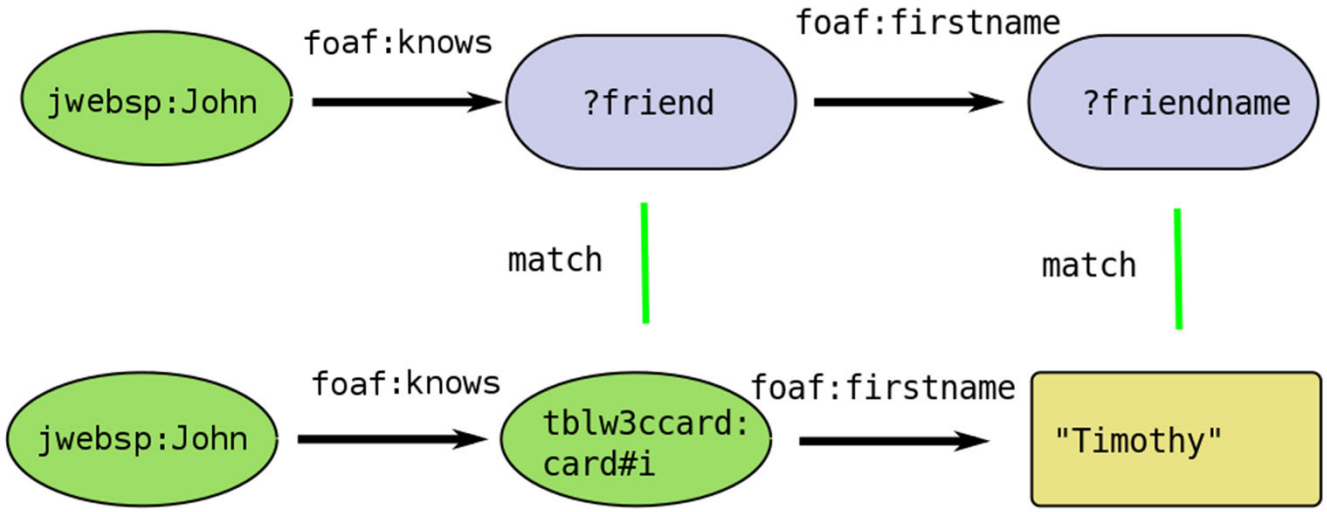
```
SELECT ?friend ?friendname WHERE {

 jwebsp:John foaf:knows ?friend .

 ?friend foaf:firstname ?friendname

}
```

Ein kleiner Beispiel RDF Graph

```
@prefix wbsp: <http://mywebspace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```



Beantwortung der zweiten Anfrage



?friend	?friendname
tblw3ccard:card#i	"Timothy"

SPARQL Anfrage Möglichkeiten

- SELECT
  - Ergebnis ist eine Tabelle der Ergebnisse
- ASK
  - Ergebnis ist eine boolesche Variable.
  - Wahr, wenn das Graph-Muster wenigstens einmal passt
- CONSTRUCT
  - Ergebnis ist das Erzeugen von neuen Tripel nach einem gegebenen Graph-Muster
- DESCRIBE
  - Ergebnis ist die Beschreibung von Ressourcen

FROM Klausel

- Spezifiziert welche Graphen für das Ergebnis berücksichtigt werden sollen
- Optional
  - Falls ausgelassen wird der sogenannte *default* Graph verwendet
  - Falls angegeben, werden nur die angegebenen Graphen berücksichtigt
  - Falls ein oder mehrere “named graph(s)” angegeben sind, können diese in der Anfrage referenziert werden

WHERE Klausel

- Enthält die Graph-Muster
- Konjunktiv
- Variablen werden an die Werte aus den Ergebnissen für die Graph-Muster gebunden

- Format der Graph-Muster:
  - Wieder die Subjekt / Prädikat / Objekt Form
  - Variablen können an jeder Position vorkommen.

Anfrage Modifikatoren

- Verändert das Ergebnis einer Anfrage
- LIMIT und OFFSET teilen das Ergebnis auf
- ORDER BY ASC / DESC
- Beispiele:
  - `SELECT * WHERE {.....} LIMIT 10`
    - Nur die ersten 10 Ergebnisse werden zurückgeliefert
  - `SELECT * WHERE {.....} ORDER BY ASC(...) LIMIT 10`
    - Nur die ersten 10 Ergebnisse werden zurückgeliefert, aufsteigend alphabetisch sortiert.

Beispiele für Graph-Muster

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John . :Tim foaf:name "Tim" .

SELECT ?name WHERE {:John foaf:name ?name}
-->     "John"
SELECT ?friend WHERE {:John foaf:knows ?friend}
-->      :Tim
SELECT ?friend ?name WHERE {:John foaf:knows
?friend . :John foaf:name ?name}
-->      :Tim "John"
SELECT ?friendsname WHERE {:John foaf:knows
?friend . ?friend foaf:name ?friendsname}
-->     "Tim"
```

Graph-Muster: Kartesisches Produkt

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?person ?friendsname WHERE {
    ?person foaf:knows ?friend .
    ?somebody foaf:name ?friendsname
}

:John "John"
:John "Tim"
:Tim "John"
:Tim "Tim"
```

Ressourcen vergleichen

- Vergleichen von URIs:
  - `foaf:name == <http://xmlns.com/foaf/spec/name>`
- Kein Umwandeln von z.B. reservierten Zeichen
  - `myns:John%20Doe != myns:John Doe` (gibt einen Fehler)
- Großschreibung beachten!
  - `foaf:name != <http://xmlns.com/foaf/spec/Name>`

Literale vergleichen

- Literale werden Zeichen für Zeichen verglichen
- Wenn Literale einen Datentyp haben, liegt es im Ermessen der SPARQL Engine den Datentyp zu interpretieren und zu vergleichen
  - Wichtig z.B. bei `xsd:int` , `xsd:date`
- Literale mit Language Tag werde auch Zeichen für Zeichen verglichen

Filter

- Verändern das Ergebnis von Graph-Mustern
- Erlauben es Werte zu testen
- Wichtigste Funktion: Restriktionen von Literal Wertebereichen
  - String Vergleich
  - Reguläre Ausdrücke (regular expressions)
  - Numerische Vergleiche
- Tests der Sprache / des Datentyps von Strings
- Das Ergebnis eines String Tests ist entweder wahr, falsch oder “type error”

Filter: Übersicht

- Logische Filter: ! , && , ||
- Mathematisch: + , - , * , /
- Vergleichs-Operatoren: = , != , > , < , ...
- Tests bzgl. RDF Datenmodell: isURI, isBlank, isLiteral, str, lang, datatype
- Tests bzgl. SPARQL Resultatmenge: bound
- Andere Operatoren: sameTerm, langMatches, regex

Filter: Strings

- str(): Wert eines Literals ohne den Datentypen und Tag
- contains(): Suche innerhalb eines Literals
- regex(): Einsatz einer vollwertigen regular expression

Filter: String Beispiel

```
:John :age 32 ; foaf:name "John"@en .
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .

SELECT ?friend {?friend foaf:name "John" . }
→   empty

SELECT ?friend {?friend foaf:name "Tim" . }
→   :Tim

SELECT ?friend {?friend foaf:name ?name .
FILTER ( str(?name) = "John")}
→   :John

SELECT ?friend {?friend foaf:name ?name .
 FILTER contains(?name, "im")}
→   :Tim
```

Filter: Sprache und Datentyp

- lang(?x) : Zugriff auf den Sprach-Bezeichner eines Literals
- langMatches(lang(?x), “en“): prüft ob ein gegebener Sprach-Bezeichner mit einem anderen Sprach-Bezeichner übereinstimmt
- datatype(?x) : Zugriff auf den Datentyp eines Literals

Filtern mit numerischen Operatoren

```
:John :age 32 ; foaf:name "John"@en .
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .

SELECT ?friend WHERE {?friend :age ?age
FILTER (?age>25)}
-->     :John
```

Filtern mit logischen Operatoren

```
:John :age 32 ;foaf:name "John"@en .
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend WHERE {
       ?friend foaf:name ?name .
       ?friend :age ?age .
       FILTER (str(?name) = "Tim" && ?age>25)}
```

```
-->    empty
```

```
SELECT ?friend WHERE {
       ?friend foaf:name ?name .
       ?friend :age ?age .
       FILTER (str(?name) = "Tim" || ?age>25)}
```

```
-->    :Tim
-->    :John
```

Optionale Werte in SPARQL

- Ähnlich wie ein left join in SQL
- Erlaubt es unvollständige Daten abzufragen
- “OPTIONAL” bearbeitet ein vollständiges Graph-Muster
- Syntax:
  - { Graph-Muster1 } OPTIONAL { Optionales-Muster}

Beispiel einer Anfrage mit “OPTIONAL” (1)

```
:John foaf:knows :Tim ;
      foaf:name "John"
      foaf:phone "+123456"
:Tim  foaf:knows :John ;
      foaf:name "Tim" .

SELECT ?name ?phone
 WHERE    {?person foaf:name ?name .
      ?person foaf:phone ?phone}
```

→"John" "+123456"

Suboptimales Ergebnis?

Beispiel einer Anfrage mit “OPTIONAL” (2)

```
:John foaf:knows :Tim ;
      foaf:name "John"
      foaf:phone "+123456"
:Tim  foaf:knows :John ;
      foaf:name "Tim" .

SELECT ?name ?phone WHERE {
    ?person foaf:name ?name .
        OPTIONAL {?person foaf:phone ?phone}}


--> "John" "+123456"
--> "Tim"
```

UNION in Graph-Mustern

- Syntax: `{Graph Muster 1} UNION {Graph Muster 2}`
- Erlaubt es mehrere Muster zu kombinieren

SPARQL Beispiel

```
:John rdf:type foaf:Person ; foaf:name "John" .
:Tim  rdf:type foaf:Person ; foaf:name "Tim" .
:Jane rdf:type foaf:Person ; rdfs:label "Jane" .

SELECT ?name WHERE
  {?person rdf:type foaf:Person . ?person foaf:name ?name}
--> "John"
--> "Tim"

SELECT ?name WHERE
  {?person rdf:type foaf:Person .
           {?person foaf:name ?name} UNION{?person rdfs:label ?name}}
--> "John"
--> "Tim"
--> "Jane"
```

Projektion

```
SELECT ?s ?o WHERE {?s ?p ?o}
  --> only the variables specified, in this case ?s and ?o


SELECT * WHERE {.....}
  --> all variables mentioned in the graph patterns


SELECT DISTINCT ........
  --> eliminates duplicates in the result
```

Count

- Eine sehr einfache Funktion um zu zählen wie oft eine Variable durch ein Ergebnis gebunden wird

- Beispiel:

```
:John foaf:knows :Tim ; foaf:name "John" .
:Tim foaf:knows :John ; foaf:name "Tim" .

SELECT count(?person) {?person foaf:name ?name}
--> 2
```

SPARQL Anfragen nach Klassen

- Vokabulare definieren Klassen
  - foaf:Person
  - foaf:Dokument

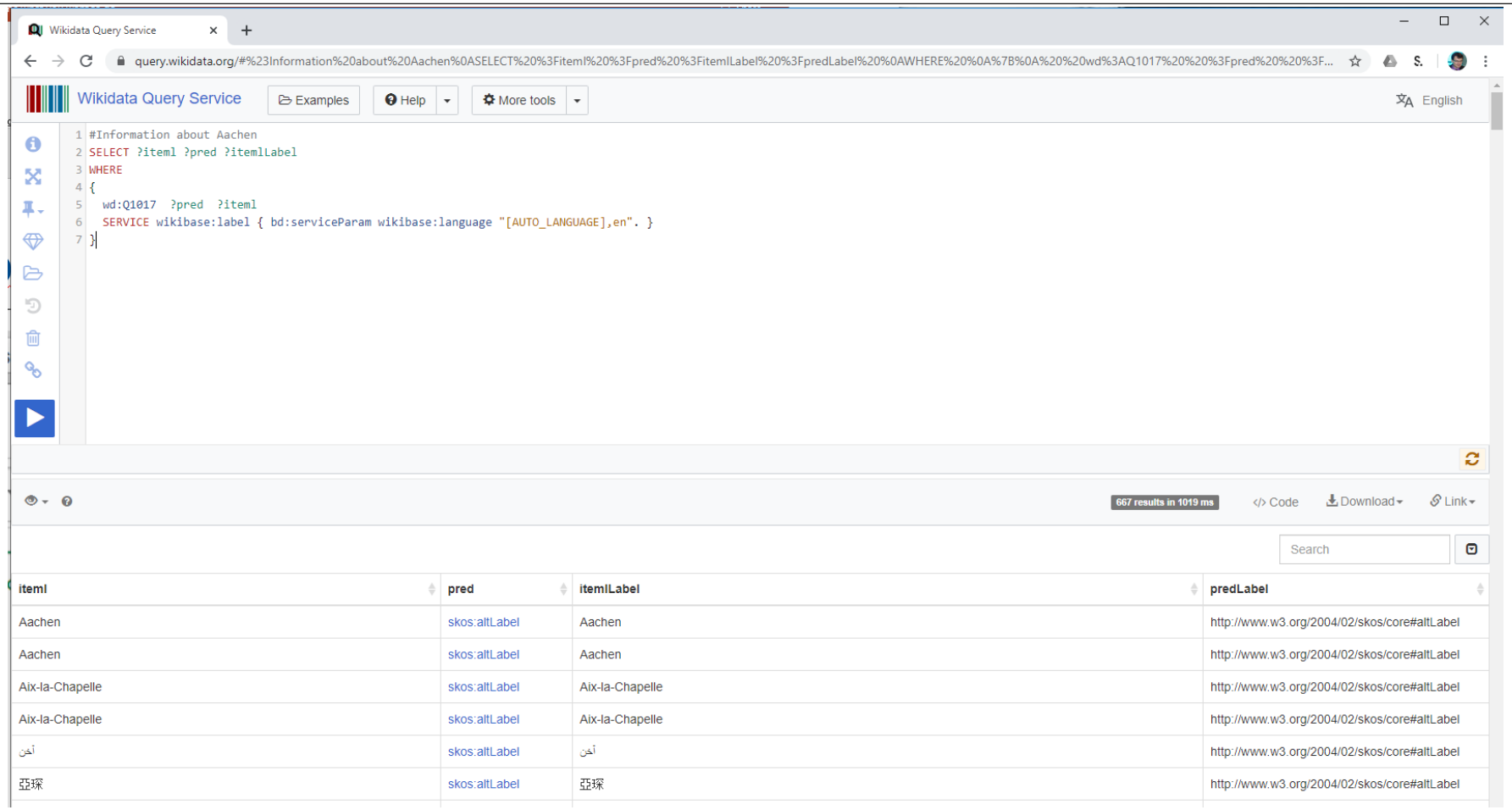
- rdf:type assoziiert eine Instanz mit einer Klasse
  - Wird auch mit „a“ abgekürzt:
  - `:John a foaf:Person == :John rdf:type foaf:Person`

SPARQL in der Praxis

Informatik 5 –
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Screenshot von Wikidata SPARQL Service



SPARQL Anfragen stellen

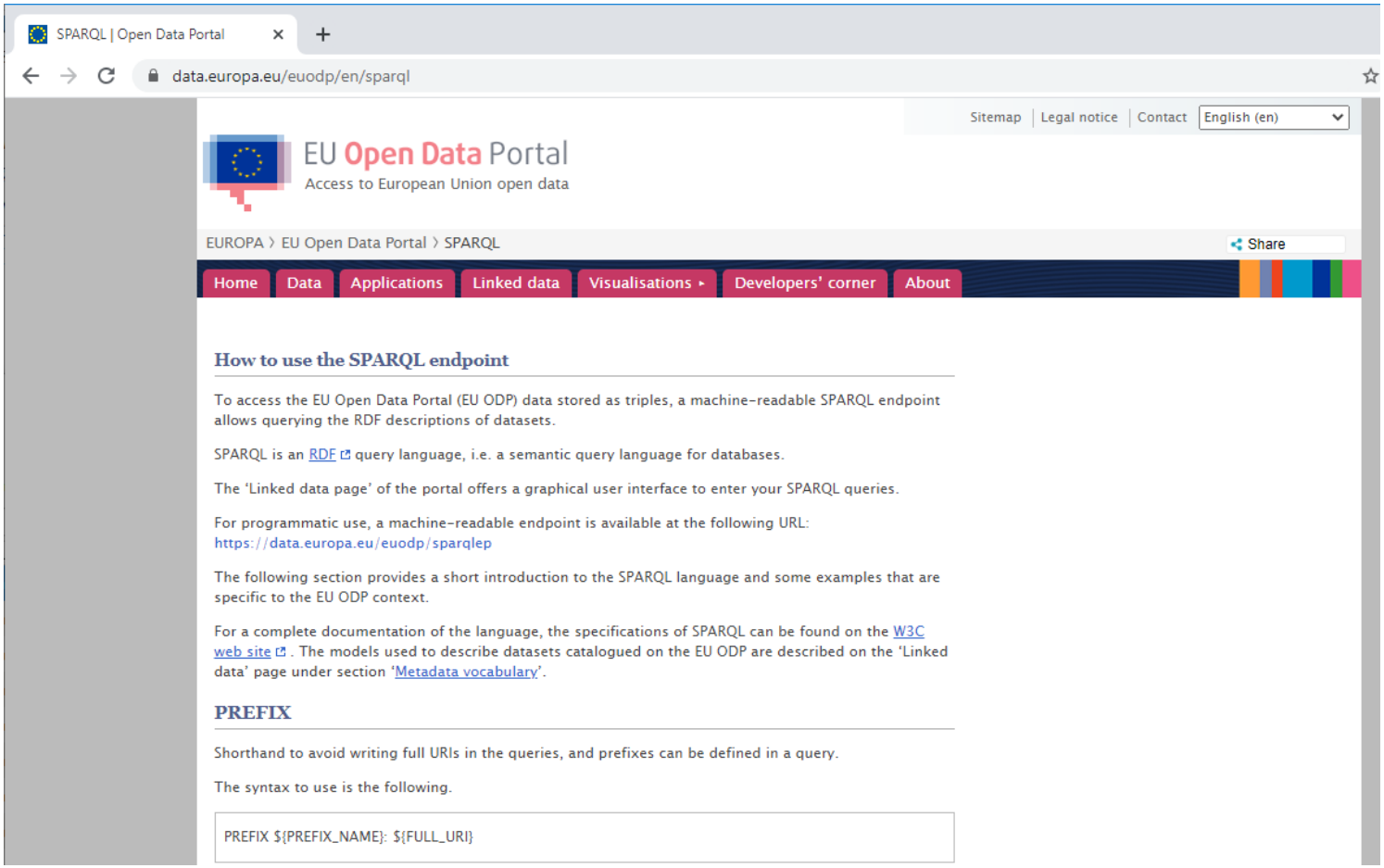
- Benutzte SPARQL um mehr über Entitäten herauszufinden mit „DESCRIBE“

```
#Information about Aachen
Describe  wd:Q1017
```

- Benutzte SPARQL um mehr über Entitäten herauszufinden

```
SELECT ?p ?o ?ol
WHERE {
  wd:Q1017  ?p  ?o .
  OPTIONAL { ?o rdfs:label ?ol }
}
```

EU Open Data Portal



Standards für Vokabulare und Ontologien

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Standards für das Web

1. RDF – Resource Description Framework

- Ein Graph-basiertes Datenformat: Knoten und Kanten
- Objekte eindeutig identifiziert: URIs
- Information wird verknüpft

2. Vokabulare und Ontologien

- Ermöglicht das gemeinsame Verständnis für eine Domäne
- Organisieren von Wissen in einem maschinen-lesbaren Format
- Gibt Daten einen auswertbare Bedeutung

RDF Schema

- Was ist RDF Schema?
- Wir können mit RDF Tripeln Fakten ausdrücken:
  - `ex:AlbertEinstein ex:discovered ex:TheoryOfRelativity`
- **Aber wie kann man solche Fakten definieren?**
- Wie können wir definieren das das Prädikat ex:discovered eine Person als Subjekt und eine Theorie als Objekt hat?
- Wie können wir die Tatsache ausdrücken, das Albert Einstein ein Forscher war und das jeder Forscher ein Mensch ist?

- Solches Wissen wird *schematisches Wissen* genannt
  - Englisch: *schema knowledge* oder *terminological knowledge*
- **RDF Schema** erlaubt es uns dieses Wissen als Model auszudrücken

RDF Schema (abgekürzt: RDFS)

- Von der W3C standardisiert, mit dem Status einer „recommendation“
- Ist selbst ein RDF Vokabular, deswegen ist jedes RDF Schema auch ein RDF Graph
- Das RDF Schema Vokabular ist generisch und nicht an einen bestimmten Anwendungsbereich gebunden
- Erlaubt es die Semantik von RDF Vokabularen zu definieren, welche von Anwendern erstellt wurden (und nicht von der W3C)
- Der Namespace von RDF Schema ist:
  - `http://www.w3.org/2000/01/rdf-schema#`
- Normalerweise wird das Präfix als `rdfs` abgekürzt

Klassen in RDF Schema

Eine *Klasse* ist eine Menge von Dingen oder Entitäten. In RDF sind diese Dinge mit URIs identifiziert

Die Mitgliedschaft einer Entität in einer Klasse ist definiert durch das **rdf:type** Prädikat.

Die Tatsache das ex:MyBlueVWGolf ein Mitglied / eine Instanz der Klasse ex:Car ist, kann wie folgt ausgedrückt werden:

```
ex:MyBlueVWGolf   rdf:type                ex:Car .
```

Eine Entität kann eine Instanz mehrerer Klassen sein.

```
ex:MyBlueVWGolf   rdf:type                ex:Car .
ex:MyBlueVWGolf   rdf:type                ex:GermanProduct .
```

.

Hierarchien von Klassen

- Klassen können mit dem **rdfs:subClassOf** Prädikat in Hierarchien gegliedert werden
- Jede Instanz von *ex:Car* ist auch ein *ex:MotorVehicle*

```
ex:Car        rdfs:subClassOf      ex:MotorVehicle .
```

Implizites Wissen 1

- Aus der Schema Definition folgt auch implizites Wissen

```
ex:MyBlueVWGolf        rdf:type              ex:Car .
ex:Car                 rdfs:subClassOf       ex:MotorVehicle .
```

- Daraus folgt als logische Konsequenz das folgende Statement:

```
ex:MyBlueVWGolf        rdf:type              ex:MotorVehicle .
```

Implizites Wissen 2

- Die folgenden Tripel

```
ex:Car                rdfs:subClassOf        ex:MotorVehicle .
ex:MotorVehicle       rdfs:subClassOf        ex:Vehicle .
```

- Implizieren das folgende Statement als eine logische Konsequenz:

```
ex:Car          rdfs:subClassOf        ex:Vehicle
```

- Wir sehen also das *rdfs:subClassOf* **transitiv** ist

Wir definieren uns eine Klasse

- Jede URI welche eine Klasse kennzeichnet, ist eine Instanz von **rdfs:Class**.
- Um eine eigene Klasse zu definieren ist folgendes Tripel notwendig:

```
ex:Car          rdf:type        rdfs:Class .
```

- Darüber hinaus gilt, das *rdfs:Class* selbst eine Instanz von *rdfs:Class* ist:

```
rdfs:Class      rdf:type                rdfs:Class .
```

Äquivalenz von Klassen

- Um auszudrücken das zwei Klassen äquivalent sind, können folgende Tripel benutzt werden:

```
ex:Car          rdfs:subClassOf        ex:Automobile .
ex:Automobile   rdfs:subClassOf        ex:Car .
```

- Woraus das folgende Tripel folgt:

```
ex:Car rdfs:subClassOf ex:Car .
```

- Daraus folgt auch das *rdfs:subClassOf* reflexiv ist

Vordefinierte RDFS Klassen

Neben rdfs:Class sind auch andere Klassen vordefiniert:

- **rdfs:Resource** ist die Klasse aller Dinge. Es ist die Superklasse aller anderen Klassen.
- **rdf:Property** ist die Klasse aller Prädikate.
- **rdf:Datatype** ist die Klasse aller Datentypen, und jede Instanz dieser Klasse ist eine Subklasse von *rdfs:Literal*.
- **rdfs:Literal** ist die Klasse aller Literale. Jedes Literal mit einem Datentyp ist auch eine Instanz von *rdfs:Datatype*.
- **rdf:langString** ist die Klasse aller Literale mit einem Sprachbezeichner. Die Klasse ist selbst eine Instanz von *rdfs:Datatype* und eine Subklasse von *rdfs:Literal*.
- **rdf:XMLLiteral** ist die Klasse der XML Literale. Es ist eine Subklasse von *rdfs:Literal* und eine Instanz von *rdfs:Datatype*
- **rdf:Statement** ist die Klasse der RDF Tripel. Jedes RDF Tripel ist eine Instanz dieser Klasse, mit den Eigenschaften *rdf:subject*, *rdf:predicate* und *rdf:object*.

Wir definieren uns ein Prädikat

- So wie Klassen definiert werden, werden auch Prädikate definiert:

```
ex:drives rdf:type rdf:Property .
```

- Mit diesem neuen Prädikat können wir ausdrücken das Max einen bestimmten VW Golf fährt:

```
ex:Max ex:drives ex:MyBlueVWGolf .
```

Hierarchische Prädikate

- Mit rdfs:subPropertyOf kann eine Hierarchie von Prädikaten definiert werden:

```
ex:drives    rdfs:subPropertyOf    ex:controls .
```

- Zusammen mit dem Statement:

```
ex:Max      ex:drives      ex:MyBlueVWGolf .
```

- Folgt daraus:

```
ex:Max      ex:controls     ex:MyBlueVWGolf .
```

Range und *Domain* für Prädikate

- Jedes Prädikat hat Domain und Range, welche spezifizieren zu welcher Klasse das Subjekt und das Objekt des Tripels gehören müssen.

```
ex:Max   ex:drives   ex:MyBlueVWGolf .
^^^^^^                ^^^^^^^^^^^^^^^
Domain                Range
```

- Definiert mit **rdfs:domain** und **rdfs:range**

```
ex:drives rdfs:domain   ex:Person .
ex:drives rdfs:range ex:Vehicle .
```

- Das gleiche für Datentypen:

```
ex:hasAge rdfs:range xsd:nonNegativeInteger .
```

Die Semantik von *Domain* und *Range*

- Hinweise zur Semantik von *Domain* und *Range:*

- “Weil *ex:MyBlueVWGolf* mit *ex:drives* verwendet wurde, wissen wir das es ein *ex:Vehicle* ist, zusätzlich zu allen anderen Klassenzugehörigkeiten die es haben mag.”

Prädikate mit mehreren *Range* Zugehörigkeiten

- Aus den folgenden Aussagen

```
ex:drives   rdfs:range   ex:Car .
ex:drives   rdfs:range   ex:Ship .
```

- Schließen wir das der *Range* von *ex:drives* sowohl ein *ex:Car* als auch ein *ex:Ship* ist, also **beides**!

- Eine bessere Möglichkeit um auszudrücken das das Objekt eines Tripels sowohl ein Auto als auch ein Schiff sein **kann**, ist wie folgt:

```
ex:Car      rdfs:subClassOf   ex:Vehicle .
ex:Ship     rdfs:subClassOf   ex:Vehicle .
ex:drives   rdfs:range        ex:Vehicle .
```

Implizites Wissen aus *Domain* und *Range*

- Sobald wir domain und range definiert haben, müssen wir auf unbeabsichtigte Folgen achten.
- Aus diesem Schema

```
ex:isMarriedTo    rdfs:domain    ex:Person .
ex:isMarriedTo    rdfs:range     ex:Person .
ex:RWTH  rdf:type       ex:Institution .
```

- und dem zusätzlichen Statement:

```
ex:Max    ex:isMarriedTo    ex:RWTH .
```

- Folgt als logische Konsequenz:

```
ex:RWTH    rdf:type    ex:Person .
```

Reifikation 1

- Wie kann man in RDF die folgende Information ausdrücken?
  - “Der Kommissar vermutet das der Butler den Gärtner getötet hat.”

```
ex:Detective   ex:supposes   "The butler killed the gardener" .
ex:Detective   ex:supposes   ex:theButlerKilledTheGardener .
```

- Beide Varianten sind nicht zufriedenstellend.
- Was wir wirklich wollen, ist direkt über diese Tripel Aussagen zu machen:

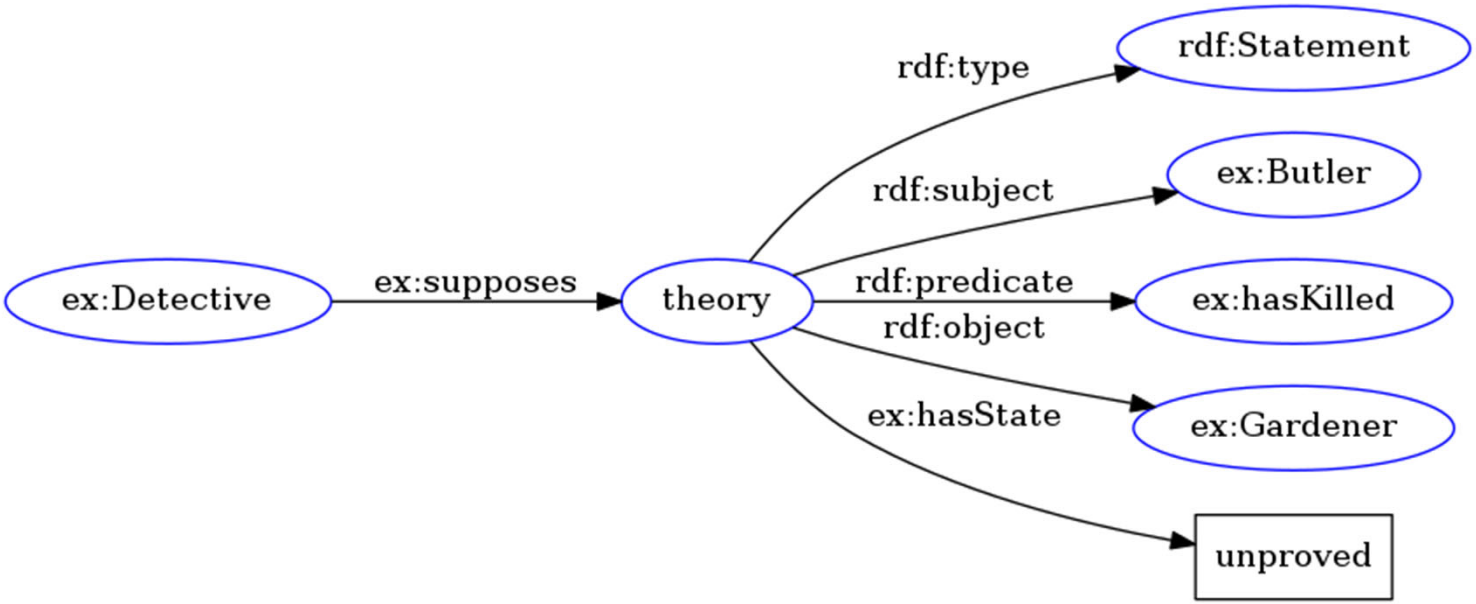
```
ex:Butler   ex:killed   ex:Gardener .
```

Reifikation 2

- Mit der Klasse rdf:Statement erlaubt RDF Schema in einem Tripel eine Aussage über ein anderes Tripel zu machen.
- Dazu werden die folgenden Prädikate verwendet:
  - **rdf:subject** definiert eine *rdfs:Resource*, die das Subjekt eines Tripels ist
  - **rdf:predicate** definiert eine *rdf:Property*, welche das Prädikat eines Tripels ist
  - **rdf:object** definiert eine *rdf:Resource* welche ein Objekt eines Tripels ist
- Das folgende Beispiel zeigt wie ein RDF Tripel als Ressource beschrieben wird, und wie Aussagen über das Tripel gemacht werden (z.B. das die Theorie noch nicht bewiesen wurde).

```
ex:Detective   ex:supposes          :theory .
:theory        rdf:type             rdf:Statement .
:theory        rdf:subject          ex:Butler .
:theory        rdf:predicate        ex:hasKilled .
:theory        rdf:object           ex:Gardener .
:theory        ex:hasState          "unproved" .
```

Reifikation Beispiel als Graph



Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
ex: http://www.example.org/
http://www.example1.org/

Reifikation 3

- Bitte Beachten Sie das die folgende Aussage **nicht** aus dem vorherigen Beispiel hervorgeht:

```
ex:Butler   ex:hasKilled   ex:Gardener .
```

- Das erlaubt es in RDF Aussagen zu machen über andere Aussagen, welche falsch oder unbewiesen sind.

Zusätzliche Informationen

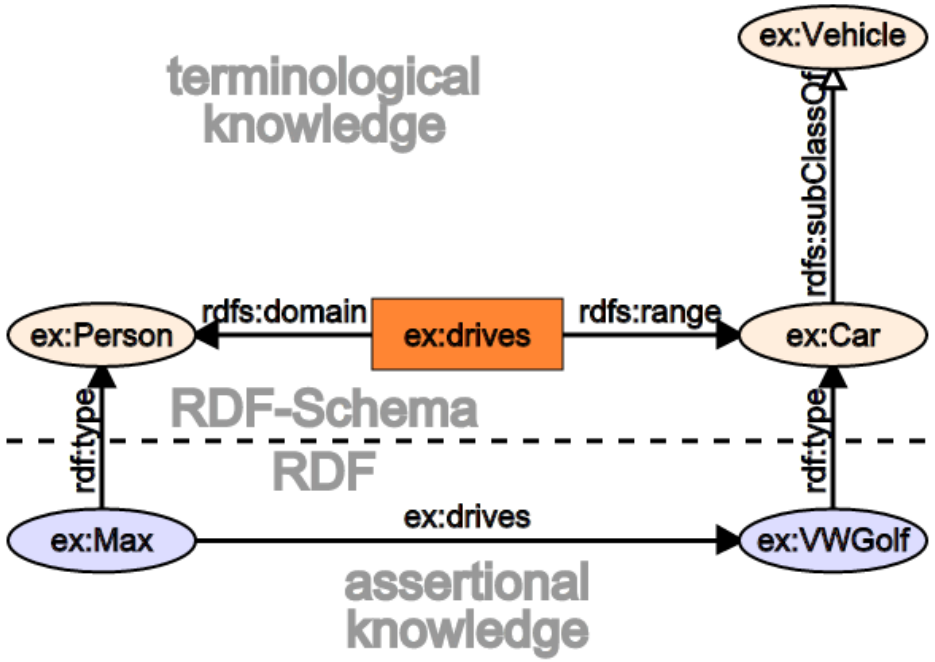
- RDF Schema erlaubt es weitere Informationen über eine Ressource mit den folgenden Prädikaten auszudrücken:
- **rdfs:label** gibt einer Ressource eine Namen (“human readable name”)
- **rdfs:comment** gibt einer Ressource eine Erklärung oder Kommentar.
- rdfs:seeAlso gibt eine URI an bei der weitere Informationen zu einer Ressource gefunden werden können.
- rdfs: isDefinedBy gibt eine URI an, welche eine Ressource definiert. (rdfs:isDefinedBy ist eine Subproperty von rdfs:seeAlso).

```
ex:VWGolf   rdfs:label            "VW Golf" .
ex:VWGolf   rdfs:comment          "The VW Golf is a popular german car..." .
ex:VWGolf   rdfs:seeAlso          http://www.wikipedia/VW_Golf .
ex:VWGolf   rdfs:isDefinedBy      http://www.Volkswagen.de/ex2:Vw_golf .
```

- Der Vorteil diese Prädikate zu verwenden, ist, das die zusätzliche Information auch als RDF ausgedrückt wird.

Vergleich: RDF und RDF Schema

- RDF Schema ist RDF zusammen mit einem Vokabular um Wissen auszudrücken (schematisches Wissen).
- Das Diagramm zeigt die verschiedenen konzeptuellen Ebenen.



Einschränkungen von RDFS

- RDF Schema kann als leichtgewichtige Sprache zur Definition von Vokabularen und Ontologien verwendet werden.
- Jedoch hat RDF Schema auch einige Einschränkungen bzgl. der Definition von Vokabularen und Ontologien.
- RDF Schema erlaubt es nicht die folgenden Sachverhalte auszudrücken:
  - **Negationen von Ausdrücken:** z.B. die Domain eines Prädikats darf eine bestimmte Klasse nicht enthalten.
  - **Keine Einschränkung der Kardinalität:** z.B. zwischen 0 und 1 *example:isMarriedTo* Prädikate pro Person.
  - **Keine Mengen von Klassen:** Es ist nicht möglich auszudrücken, das eine Domain auf mehrere Klassen zutrifft. Dann benötigen wir eine neue Superklasse.
  - **Keine Metadaten für ein Schema:** z.B. keine Versionsnummer.

Zusammenfassung RDF Schema

- RDF Schema drückt **Wissen** durch die **Definition von Klassen und Prädikaten** aus (schematisches Wissen)
- Klassen und Prädikate(“properties”) können in **Hierarchien** angeordnet werden.
- Für Prädikate können ***Domain*** (Einschränkung des Subjekts) und ***Range*** (Einschränkung des Prädikats) definiert werden
- Ein Schema erlaubt es auf **implizit definiertes Wissen** zu schließen (“inference of implicit knowledge”).

- RDF Schema kann für **“leichtgewichtige” Vokabulare** und Ontologien verwendet werden, ist aber nicht so mächtig wie z.B. OWL. (OWL wird **nicht** in dieser Vorlesung behandelt.)

Zusammenfassung: Semantic Web

- RDF: das Datenmodell des Semantic Web
- Serialisierung von RDF Graphen
- Beispiele zur Verwendung von RDF Daten
- SPARQL: Anfrage-Sprache für RDF Graphen
- SPARQL in der Praxis
- Standards für Vokabulare und Ontologien im Semantic Web

Nicht-Standard Datenmodelle und Datenbanken

1. Einführung
2. RDF und Semantic Web
3. **Graphdatenbanken**

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Not only SQL

Relationale Datenbanken sind für große Datenmengen **oder** viele gleichzeitige Zugriffe optimiert. Das Internet hat den Bedarf an Big Data Lösungen in dem die Datenmenge groß **und** die Anzahl der Zugriffe hoch ist

- **Nicht-Relational:** Keine Tabellenstruktur, ermöglicht agile Verarbeitung großer Datenmengen.
- **Verschiedene Datenmodelle:** Umfasst Key-Value-Stores, Dokumenten-Datenbanken, Wide-Column-Stores und Graphdatenbanken.
- **Skalierbarkeit:** Hoch skalierbar, geeignet für Verteilung auf viele Server.
- **Flexibilität:** Schema-los, bietet Flexibilität bei Datenspeicherung und -manipulation.
- **Leistung:** Schnelle Schreib- und Lesevorgänge, ideal für Echtzeitanwendungen.
- **Big Data & Echtzeit-Web-Apps:** Ideal für Big Data und schnelle Webdienste.
- **Konsistenzmodell:** Nutzt "eventuelle Konsistenz" für hohe Verfügbarkeit.

NoSQL: Datenbank-Typen

Key-Value-Store:
Vereinfachte Datenstruktur, die Daten als Schlüssel-Wert-Paare speichert.
Ideal für Anwendungen, die schnelle Lese- und Schreibzugriffe benötigen.
Beispiele: Redis, DynamoDB.
Dokumenten-Datenbanken:
Speichert Daten als "Dokumente", meist im JSON-Format.
Flexible Schemata, einfach hinzuzufügen und zu ändern.
Ideal für Anwendungen, die komplexe, hierarchische Datenstrukturen verwalten.
Beispiele: MongoDB, CouchDB.
Wide-Column-Stores:
Speichern Daten in Spalten statt in Zeilen, gut für die Analyse großer Datenmengen.
Hoch skalierbar und effizient bei der Speicherung großer Datenmengen.
Beispiele: Cassandra, HBase.
Graphdatenbanken:
Verwenden Graphenstrukturen mit Knoten, Kanten und Eigenschaften, um relationale Daten darzustellen.
Ideal für Anwendungen, die komplexe Beziehungen und Verbindungen analysieren.
Beispiele: Neo4j, Blazegraph, Amazon Neptune -> Gremlin/ApacheTinkerpop

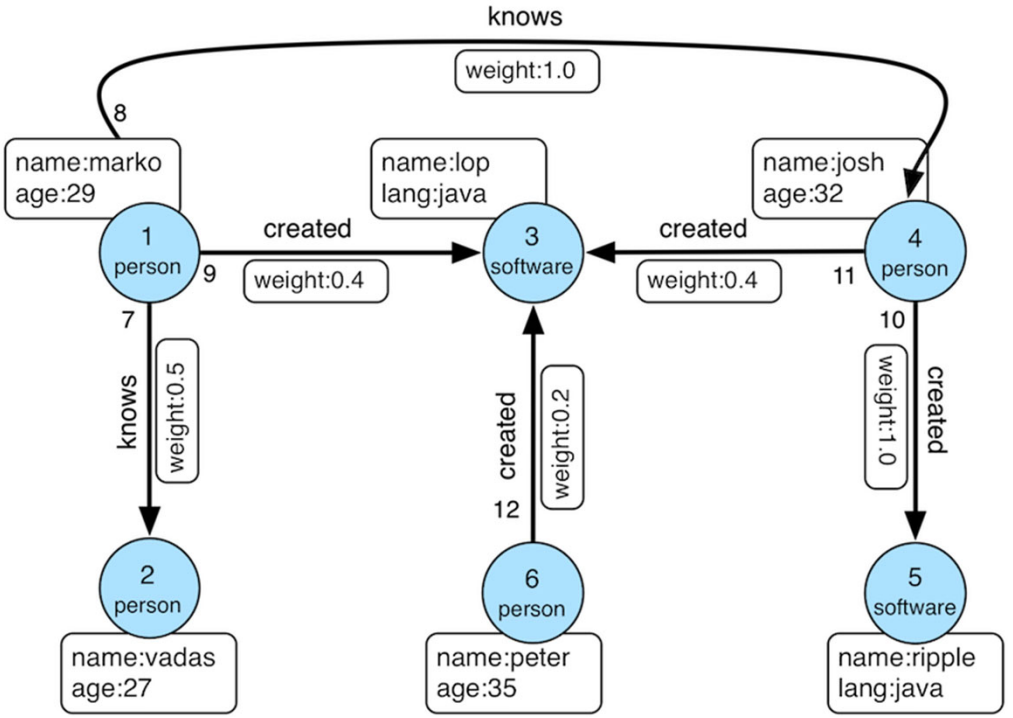
Datenmodell: Property Graph

Knoten

- Repräsentiert eine *Entity*
- Hat 0 oder mehr *Label*
- Hat 0 oder mehr *Properties*

Kanten

- Strukturieren den Graphen (semantischer Kontext)
- Haben einen Typ
- Haben 0 oder mehr Properties
- Setzen Koten zueinander in Relation
- Haben einen Start und Ende-Knoten



Eigenschaften

- Key-Value Paare
- Repräsentieren Daten
- Struktur „String key; typisierter Wert

INTERNATIONAL STANDARD

ISO/IEC 9075-16

First edition
2023-06

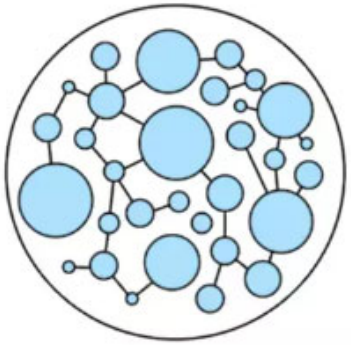
Information technology — Database languages SQL —

Part 16: Property Graph Queries (SQL/PGQ)

Apache Tinkerpop/Gremlin

- Open Source Graph Computing Framework
- Gremlin ist die Graph-Verarbeitungssprache

Struktur



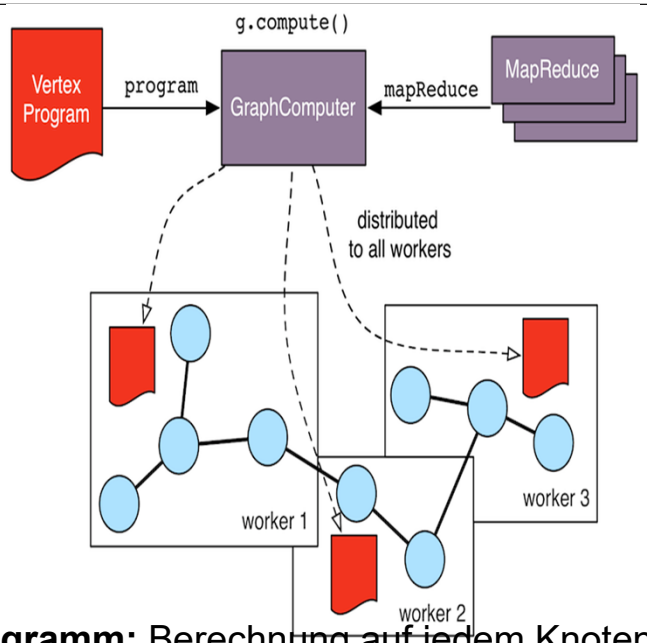
Graph

+

Prozess

Gremlin

https://tinkerpop.apache.org/



Vertex-Programm: Berechnung auf jedem Knoten (Vertex)

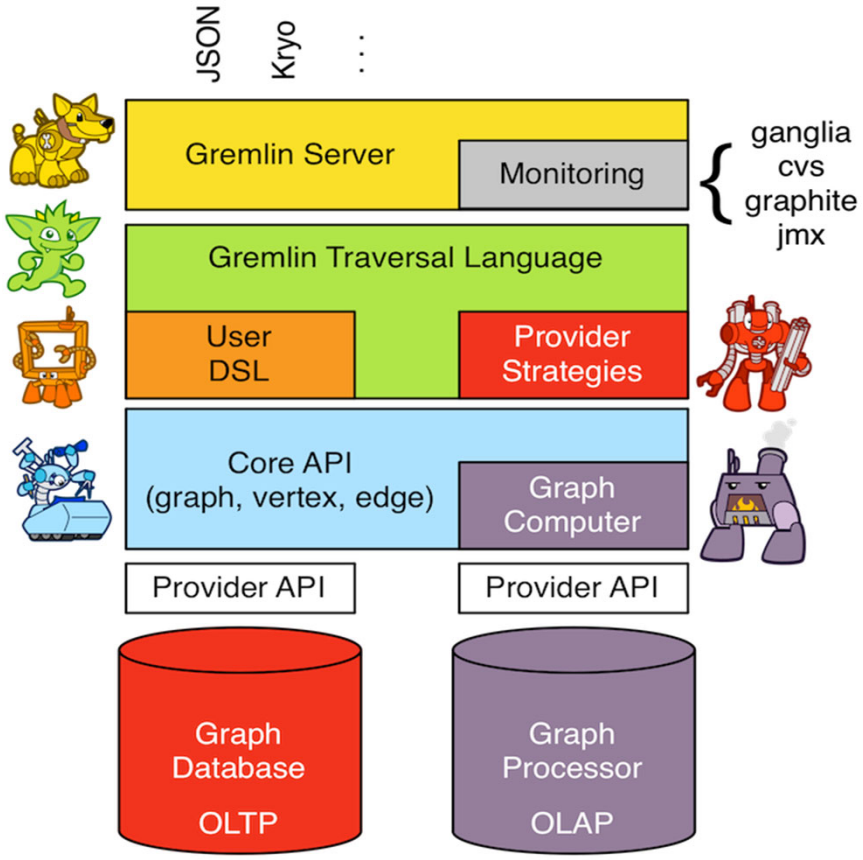
GraphComputer: Steuert Ausführung von Vertex-Programmen

MapReduce: "Map" führt Berechnungen auf Knotenebene aus, und "Reduce" aggregiert diese Ergebnisse

Apache Tinkerpop/Gremlin Architektur - sowohl für OLAP als auch OLTP geeignet

OLTP (Online Transaction Processing):
- Echtzeit-Transaktionsbedürfnisse
- optimiert für schnelle und zuverlässige Verarbeitung kleiner Datenmengen.
- Bankwesen oder Einzelhandelsverkäufe.

OLAP (Online Analytical Processing):
- Systeme für Datenanalyse konzipiert,
- optimiert um komplexe Abfragen auf großen Datenmengen zu bewältigen.
- Verwendet in Business Intelligence und Data Warehousing
- Schwerpunkt auf Datenaggregation, multidimensionaler Analyse



Apache Tinkerpop/Gremlin

- https://tinkerpop.apache.org/providers.html

Alibaba Graph Database
A real-time, reliable, cloud-native graph database service that supports property graph model.

Amazon Neptune
Fully-managed graph database service.

ArcadeDB
Multi-Model Database Supporting Graphs, Key/Value, Documents and Time-Series.

HugeGraph
A high-speed, distributed and scalable OLTP and OLAP graph database with visualized analytics platform.

IBM Db2 Graph
Graph database using IBM® Db2®.

JanusGraph
Distributed OLTP and OLAP graph database with BerkeleyDB, Apache Cassandra and Apache HBase support.

ArangoDB
OLTP Provider for ArangoDB.

Bitsy
A small, fast, embeddable, durable in-memory graph database.

Blazegraph
RDF graph database with OLTP support.

Neo4j
OLTP graph database (embedded and high availability).

OrientDB
OLTP graph database

OverflowDB
In-memory graph database with low memory footprint

CosmosDB
Microsoft's distributed OLTP graph database.

ChronoGraph
A versioned graph database.

DSEGraph
DataStax graph database with OLTP and OLAP support.

Apache S2Graph
OLTP graph database running on Apache HBase.

Sqlg
OLTP implementation on SQL databases.

Stardog
RDF graph database with OLTP and OLAP support.

Hadoop (Spark)
OLAP graph processor using Spark.

HGraphDB
OLTP graph database running on Apache HBase.

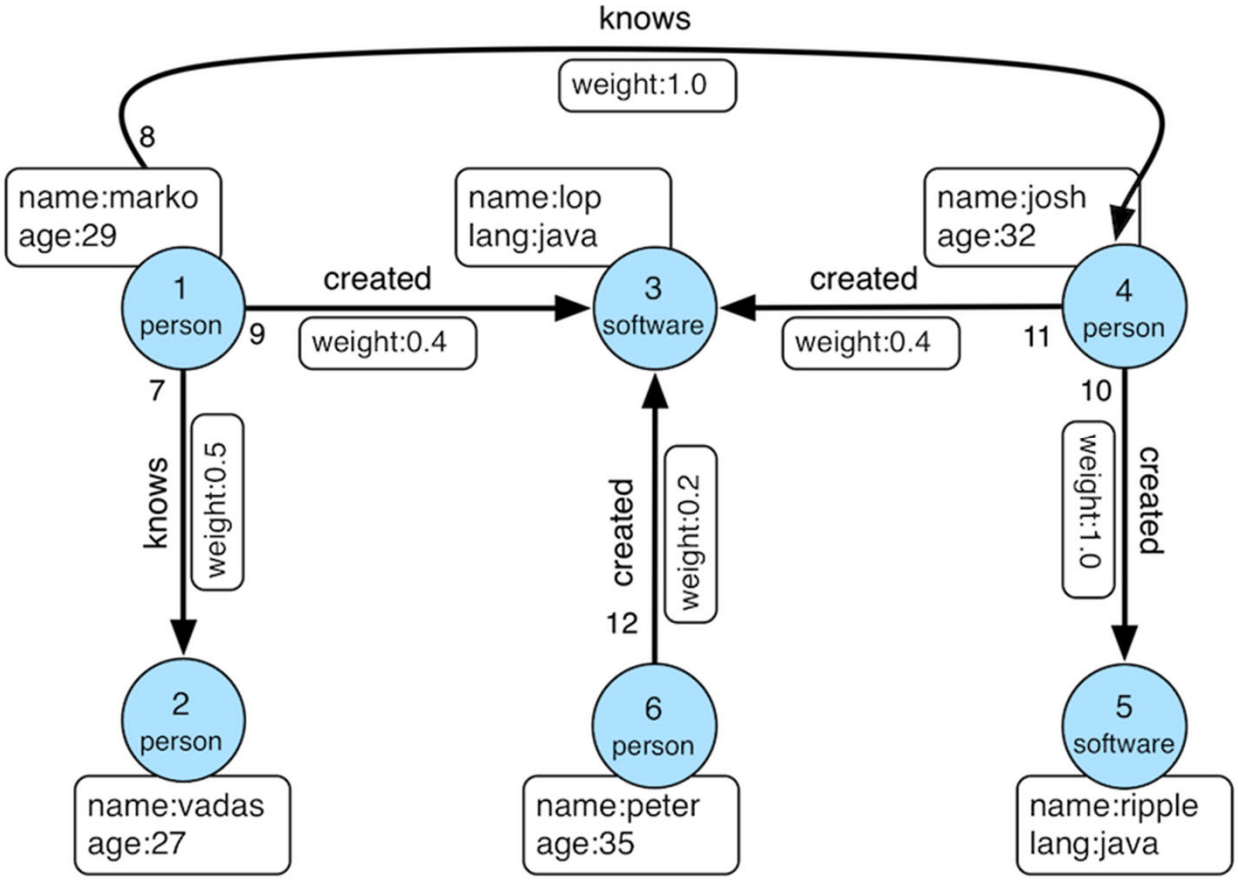
Huawei Graph Engine Service
Fully-managed, distributed, at-scale graph query and analysis service that provides a visualized interactive analytics platform.

Tibco Graph Database
Combined OLTP and OLAP features in a single enterprise-grade database.

TinkerGraph
In-memory OLTP and OLAP reference implementation.

Unipop
OLTP Elasticsearch and JDBC backed graph.

Der apache tinkerpop „modern“ Beispiel Graph



Graph Definition

G=(V,E)

V=Menge der Vertices (Knoten)
E=Menge Edges (Kanten)

Grundprinzip: Graph Tranversals

Graph Traversal