Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 1: Einführung

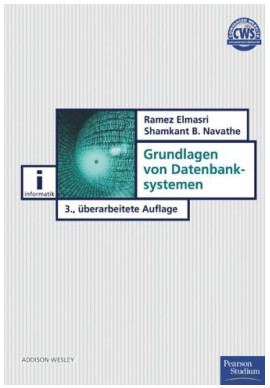
Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Literaturliste

Die Vorlesung orientiert sich nicht an einem bestimmten Lehrbuch. Empfehlenswert sind aber u.a.:

- A. Kemper, A. Eickler:
  Datenbanksysteme
  Oldenbourg, 5. Auflage (2004). 39,80 €
- R. Elmasri, S. B. Navathe:
  Grundlage von Datenbanksystemen
  Pearson Studium, 3. Auflage (2004). 39,95 €
- A. Heuer, G. Saake, K.-U. Sattler:
  Datenbanken kompakt
  mitp, 2. Auflage (2003). 19,95 €
- A. Heuer, G. Saake:
  Datenbanken: Konzepte und Sprachen
  mitp, 2. Auflage (2000). 35,28 €
- R. Ramakrishnan, J. Gehrke:
  Database Management Systems
  McGraw Hill, 3. Auflage (2002).



Das Team

Vorlesung

Prof. Dr. Christian Böhm

Übungen

Annahita Oswald

Bianca Wackersreuther

Tutor

Volker Jarre

Wovon handelt die Vorlesung?

- Bisher (Einführungsvorlesung):
  Nur Betrachtung des Arbeitsspeichers.
  Objekte werden im Arbeitsspeicher erzeugt und nach dem Programmlauf wieder entfernt

- Warum ist dies nicht ausreichend?
  - Viele Anwendungen müssen Daten *permanent* speichern
  - Arbeitsspeicher ist häufig *nicht groß genug*, um z.B. alle Kundendaten einer Bank oder Patientendaten einer Klinik zu speichern

Permanente Datenspeicherung

- Daten können auf dem sog. *Externspeicher* (auch Festplatte genannt) permanent gespeichert werden

- Arbeitsspeicher:
  - rein elektronisch (Transistoren und Kondensatoren)
  - flüchtig
  - schnell: 10 ns/Zugriff *
  - wahlfreier Zugriff
  - teuer: 100-150 € für 1 GByte*

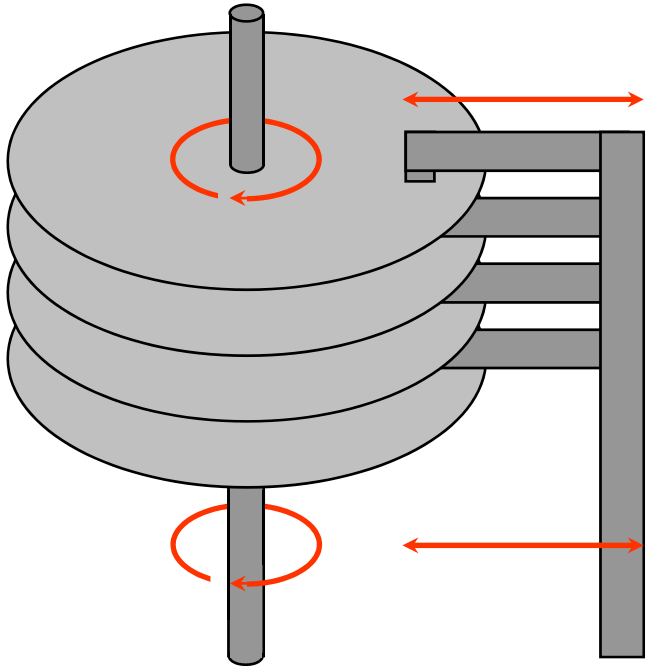
Externspeicher:

- Speicherung auf magnetisierbaren Platten (rotierend)
- nicht flüchtig
- langsam: 5 ms/Zugriff *
- blockweiser Zugriff
- wesentlich billiger: 100 € für ca. 200 GByte*

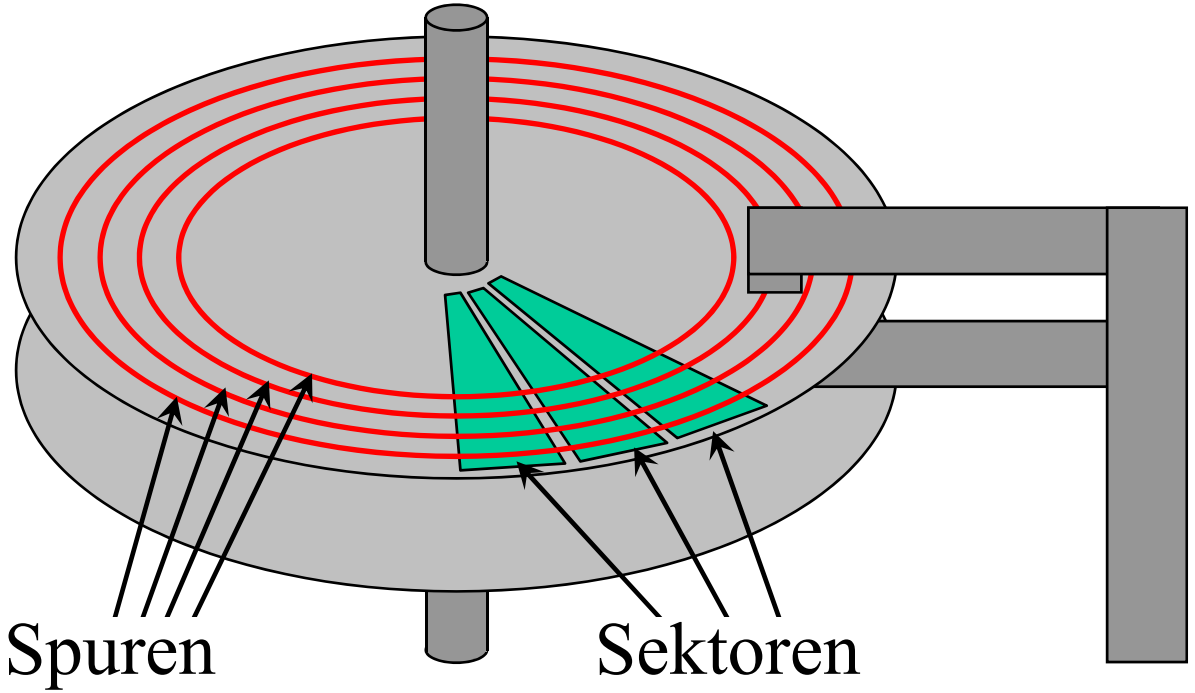
*Oktober 2005 .

Aufbau einer Festplatte

- Mehrere magnetisierbare *Platten* rotieren z.B. mit 7.200 Umdrehungen* pro Minute um eine gemeinsame Achse (*z. Z. 5400, 7200, 10000 upm)
- Ein Kamm mit je zwei *Schreib-/Leseköpfen* pro Platte (unten/oben) bewegt sich in radialer Richtung.



Einteilung der Plattenoberflächen



- (interne) Adressierung einer Information:
  [Platten-Nr | Oberfl.-Nr | Spur-Nr | Sektor-Nr | Byte-Nr]
- Berechnung der Kapazität:
  # Platten * 2 * # Spuren * # Sektoren * Bytes pro Sektor

Lesen/Schreiben eines Sektors

- Positionieren des Kamms mit den Schreib-/Leseköpfen auf der Spur
- Warten bis die Platte so weit rotiert ist, dass der Beginn des richtigen Sektors unter dem Schreib-/Lesekopf liegt
- Übertragung der Information von der Platte in den Arbeitsspeicher (bzw. umgekehrt)

Achtung:

Es ist aus technischen Gründen nicht möglich, einzelne Bytes zu lesen bzw. zu schreiben, sondern mindestens einen ganzen Sektor

Speicherung in Dateien

- Adressierung mit Platten-Nr., Oberfl.-Nr. usw. für den Benutzer nicht sichtbar

- Arbeit mit Dateien:
  - Dateinamen
  - Verzeichnishierarchien
  - Die Speicherzellen einer Datei sind byteweise von 0 aufsteigend durchnummeriert.
  - Die Ein-/Ausgabe in Dateien wird gepuffert, damit nicht der Programmierer verantwortlich ist, immer ganze Sektoren zu schreiben/lesen.

Beispiel: Dateizugriff in Java

```
public static void main (String[] args) {
  try {
    RandomAccessFile f1 = new
      RandomAccessFile("test.file","rw");
    int c = f1.read() ;
    long new_position = .... ;
    f1.seek (new_position) ;
    f1.write (c) ;
    f1.close () ;
  } catch (IOException e) {
    System.out.println ("Fehler: " + e) ;
  }
}
```

Datei-Handle

Datei öffnen

ein Byte lesen

auf neue Position

ein Byte schreiben

Datei schließen

Fehlerbehandlung

Beispiel: Dateizugriff in Java

- Werden die Objekte einer Applikation in eine Datei geschrieben, ist das Dateiformat vom Programmierer festzulegen:

- Wo findet man dieses Datei-Schema im Quelltext z.B. des Java-Programms ?

  Das Dateischema wird nicht explizit durch den Quelltext beschrieben, sondern implizit in den Ein-/Auslese-Prozeduren der Datei

Logische Datenabhängigkeit

- Konsequenzen bei einer Änderung des Dateiformates (z.B. durch zusätzliche Objektattribute in einer neuen Programmversion):
  - Alte Datendateien können nicht mehr verwendet werden oder müssen z.B. durch extra Programme konvertiert werden
  - Die Änderung muss in allen Programmen nachgeführt werden, die mit den Daten arbeiten, auch in solchen, die logisch von Änderung gar nicht betroffen sind

Physische Datenabhängigkeit

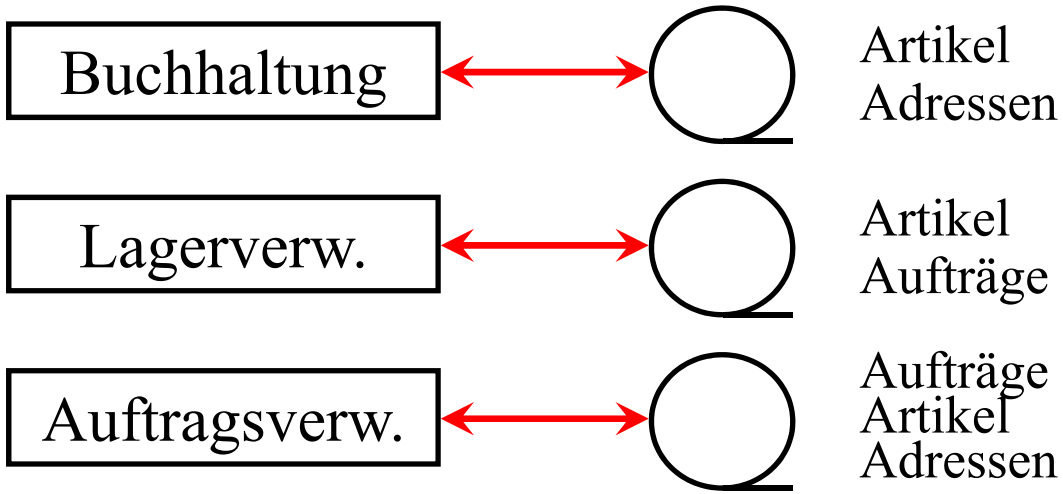
- Meist werden die Datensätze anhand ihrer Position adressiert/separiert:
  - z.B. jeder Satz hat 22 Zeichen:
  - 1. Satz: Adresse 0; 2. Satz: Adresse 22 usw.
- Suche gemäß bestimmten Attributwerten (z.B. Namen des Kunden) muss im Programm codiert werden
- Soll die Datei z.B. mit einem Suchbaum unterstützt werden, dann gleiche Konsequenzen wie bei logischer Änderung

Informationssysteme

- Große Software-Systeme
- Viele einzelne Programme
- Programme arbeiten teils mit gemeinsamen Daten, teils mit unterschiedlichen
- Beispiele für die Programme:
  - Buchhaltung: Artikel- und Adressinformation
  - Lagerverwaltung: Artikel und Aufträge
  - Auftragsverwaltung.: Aufträge, Artikel, Adressen
  - CAD-System: Artikel, techn. Daten, Bausteine
  - Produktion, Bestelleingang, Kalkulation: ...

Redundanz

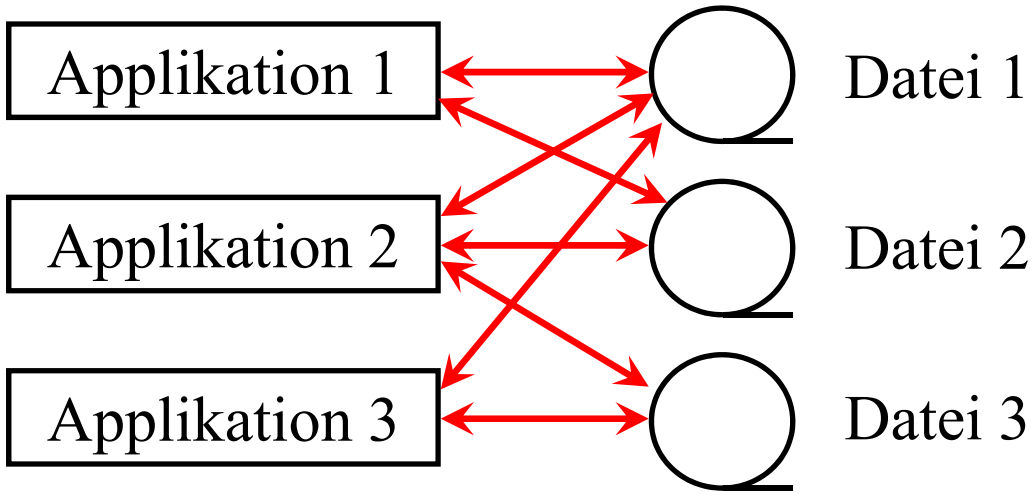
- Daten werden meist mehrfach gespeichert



- Konsequenz: u.a. *Änderungs-Anomalien*
  Bei Änderung einer Adresse müssen viele Dateien nach den Einträgen durchsucht werden
  (hierzu später mehr)

Schnittstellenproblematik

- Alternative Implementierung



- Nachteile:
  - unübersichtlich
  - bei logischen oder physischen Änderungen des Dateischemas müssen viele Programme angepasst werden

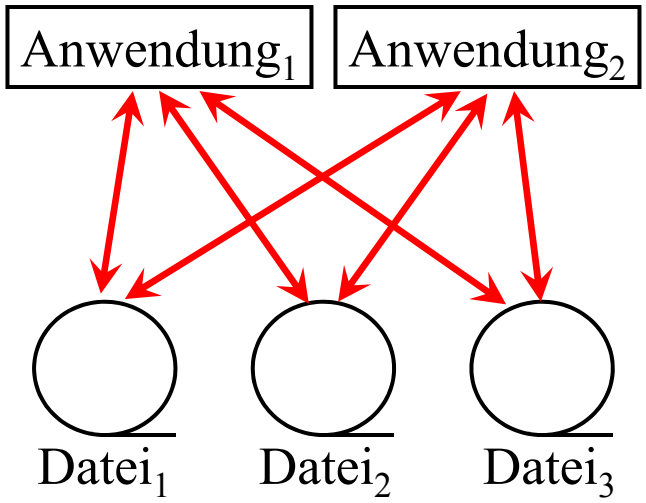
Weitere Probleme von Dateien

- In großen Informationssystemen arbeiten viele Benutzer gleichzeitig mit den Daten: Dateisysteme bieten zu wenige Möglichkeiten, um diese Zugriffe zu synchronisieren
- Dateisysteme schützen nicht in ausreichendem Maß vor Datenverlust im Fall von Systemabstürzen und Defekten
- Dateisysteme bieten nur unflexible Zugriffskontrolle (Datenschutz)

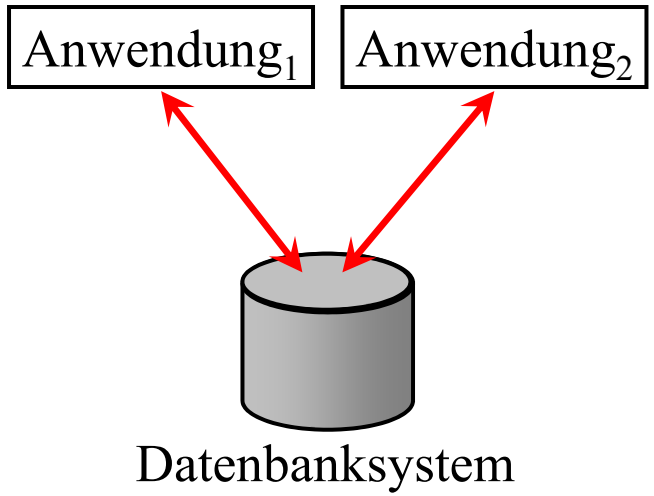
Von Dateien zu Datenbanken

- Um diese Probleme mit einheitlichem Konzept zu behandeln, setzt man Datenbanken ein:

Mit Dateisystem:



Mit Datenbanksystem:



Komponenten eines DBS

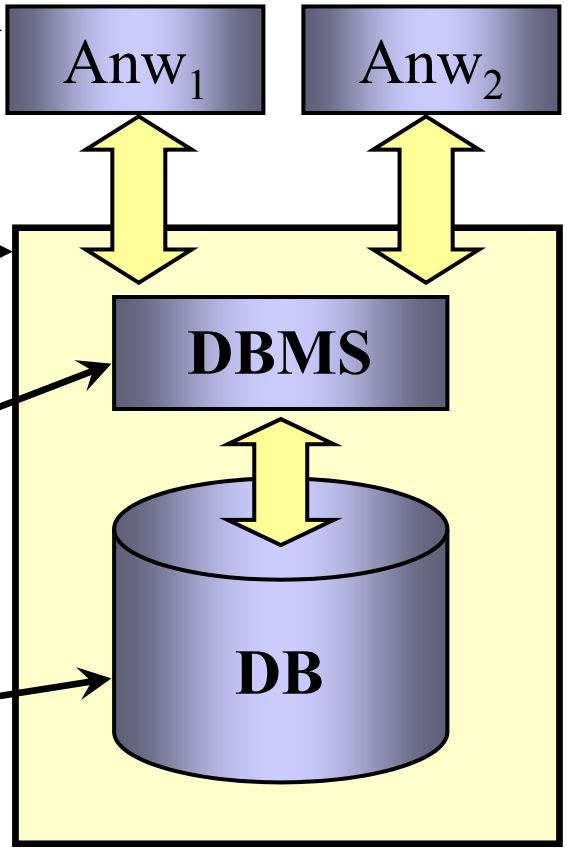
Man unterscheidet zwischen...

DB-Anwendungen
(kommunizieren mit DBMS)

DBS: Datenbanksystem
(DB + DBMS)

DBMS: Datenbank-Management-System
(Software zur Verwaltung)

DB: Datenbank
(eigentliche Datensammlung)



Typische Einsatzbereiche

- Im betriebswirtschaftlichen Bereich:
  - Banken (Kontoführung)
  - Buchhaltung und Rechnungswesen
  - Flugbuchungssysteme
  - Telefongesellschaften (Abrechnung)
  - Lagerverwaltung
- Im technisch-wissenschaftlichen Bereich:
  - CAD/CAM/CIM
  - Medizin
  - Molekularbiologie (Gendatenbanken)

Aufgaben eines DBS

Primäre Aufgabe eines DBS ist die ...

- Beschreibung
- Speicherung und Pflege
- und Wiedergewinnung

umfangreicher Datenmengen, die von verschiedenen Anwendungsprogammen dauerhaft (persistent) genutzt werden

Anforderungen an ein DBS

Liste von 9 Anforderungen (Edgar F. Codd, 1982)

- Integration
  Einheitliche Verwaltung *aller* von Anwendungen benötigten Daten.
  Redundanzfreie Datenhaltung des gesamten Datenbestandes
- Operationen
  Operationen zur Speicherung, zur Recherche und zur Manipulation der Daten müssen vorhanden sein
- Data Dictionary
  Ein Katalog erlaubt Zugriffe auf die Beschreibung der Daten
- Benutzersichten
  Für unterschiedliche Anwendungen unterschiedliche Sicht auf den Bestand
- Konsistenzüberwachung
  Das DBMS überwacht die Korrektheit der Daten bei Änderungen

Anforderungen an ein DBS

- Zugriffskontrolle
  Ausschluss unautorisierter Zugriffe
- Transaktionen
  Zusammenfassung einer Folge von Änderungsoperationen zu einer Einheit, deren Effekt bei Erfolg permanent in DB gespeichert wird
- Synchronisation
  Arbeiten mehrere Benutzer gleichzeitig mit der Datenbank dann vermeidet das DBMS unbeabsichtigte gegenseitige Beeinflussungen
- Datensicherung
  Nach Systemfehlern (d.h. Absturz) oder Medienfehlern (defekte Festplatte) wird die Wiederherstellung ermöglicht (im Ggs. zu Datei-Backup Rekonstruktion des Zustands der letzten erfolgreichen TA)

Inhalte von Datenbanken

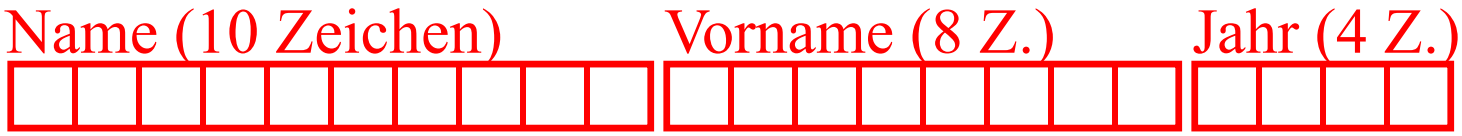
Man unterscheidet zwei Ebenen:

- Intensionale Ebene: **Datenbankschema**
  - beschreibt *möglichen* Inhalt der DB
  - Struktur- und Typinformation der Daten (Metadaten)
  - Art der Beschreibung vorgegeben durch Datenmodell
  - Änderungen möglich, aber selten (Schema-Evolution)
- Extensionale Ebene: **Ausprägung der Datenbank**
  - *tatsächlicher* Inhalt der DB (DB-Zustand)
  - Objektinformation, Attributwerte
  - Struktur vorgegeben durch Datenbankschema
  - Änderungen häufig (Flugbuchung: 10000 TA/min)

Inhalte von Datenbanken

Einfaches Beispiel:

- Schema:



- DB-Zustand:

F	r	a	n	k	l	i	n			A	r	e	t	h	a			1	9	4	2
R	i	t	c	h	i	e				L	i	o	n	e	l			1	9	4	9

- Nicht nur DB-Zustand, sondern auch DB-Schema wird in DB gespeichert.
- Vorteil: Sicherstellung der Korrektheit der DB

Vergleich bzgl. des Schemas

- Datenbanken
  - Explizit modelliert (Textdokument oder grafisch)
  - In Datenbank abgespeichert
  - Benutzer kann Schema-Informationen auch aus der Datenbank ermitteln: *Data Dictionary, Metadaten*
  - DBMS überwacht Übereinstimmung zwischen DB-Schema und DB-Zustand
  - Änderung des Schemas wird durch DBMS unterstützt (Schema-Evolution, Migration)

Vergleich bzgl. des Schemas

- Dateien
  - Kein Zwang, das Schema explizit zu modellieren
  - Schema implizit in den Prozeduren zum Ein-/Auslesen
  - Schema gehört zur Programm-Dokumentation
  - oder es muss aus Programmcode herausgelesen werden.
    Hacker-Jargon: Entwickler-Doku, RTFC (read the f...ing code)
  - Fehler in den Ein-/Auslese-Prozeduren können dazu führen, dass gesamter Datenbestand unbrauchbar wird:

F	r	a	n	k	l	i	n			A	r	e	t	h	a		1	9	4	2	R
i	t	c	h	i	e				L	i	o	n	e	l			1	9	4	9	

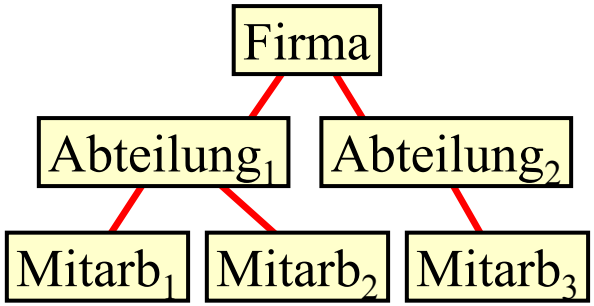
  - Bei Schema-Änderung müssen Migrations-Prozeduren programmiert werden, um bestehende Dateien auf das neue Format umzustellen

Datenmodelle

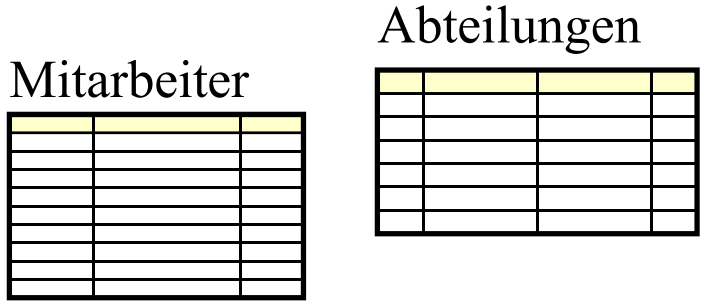
- Formalismen zur Beschreibung des DB-Schemas
  - Objekte der Datenbank
  - Beziehungen zwischen verschiedenen Objekten
  - Integritätsbedingungen

- Verschiedene Datenmodelle unterscheiden sich in der Art und Weise, wie Objekte und Beziehungen dargestellt werden:

Hierarchisch: Baum



Relational: Tabellen



Datenmodelle

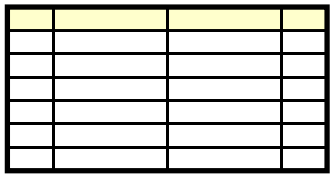
- Weitere Unterschiede zwischen Datenmodellen:
  - angebotene Operationen (insbes. zur Recherche)
  - Integritätsbedingungen

- Die wichtigsten Datenmodelle sind:
  - Hierarchisches Datenmodell
  - Netzwerk-Datenmodell
  - Relationales Datenmodell
  - Objektorientiertes Datenmodell
  - Objekt-relationales Datenmodell

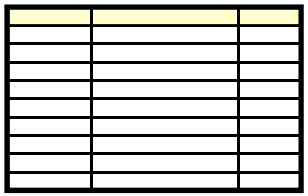
Relationales Modell

- Alle Informationen werden in Form von Tabellen gespeichert
- Die Datenbank besteht aus einer Menge von Tabellen (Relationen)
- Im Beispiel enthält die Tabelle „Mitarbeiter“ Informationen über die Mitarbeiter des Betriebes
- In jeder Zeile (Tupel) Information über einen
- Mitarbeiter (die Zeilen sind strukturell gleich)
- Die Spalten (Attribute) haben einen Namen (z.B. *Personalnr*, *Name*, *Vorname*, *Geburtsdatum*, etc.).
  Sie sind strukturell (Typ, Anzahl Zeichen) verschieden.

Abteilungen



Mitarbeiter



Relationales Modell

- Die Attribute der Tupel haben primitive Datentypen wie z.B. String, Integer oder Date
- Komplexe Sachverhalte werden durch Verknüpfung mehrerer Tabellen dargestellt
- Beispiel:

Mitarbeiter

PNr	Name	Vorname	ANr
001	Huber	Erwin	01
002	Mayer	Hugo	01
003	Müller	Anton	02

Abteilungen

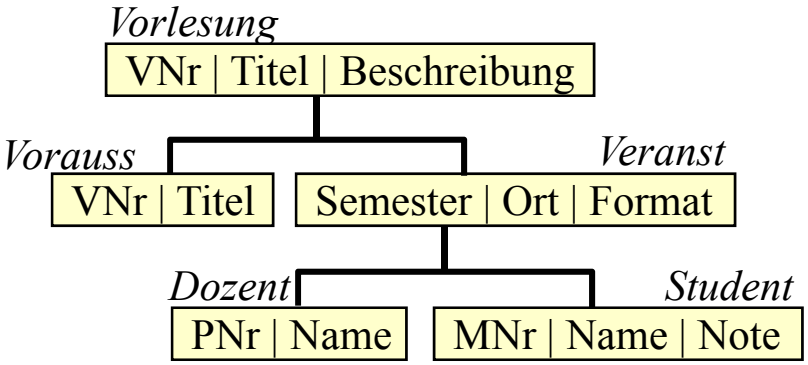
ANr	Abteilungsname
01	Buchhaltung
02	Produktion
03	Marketing

- Später ausführliche Behandlung

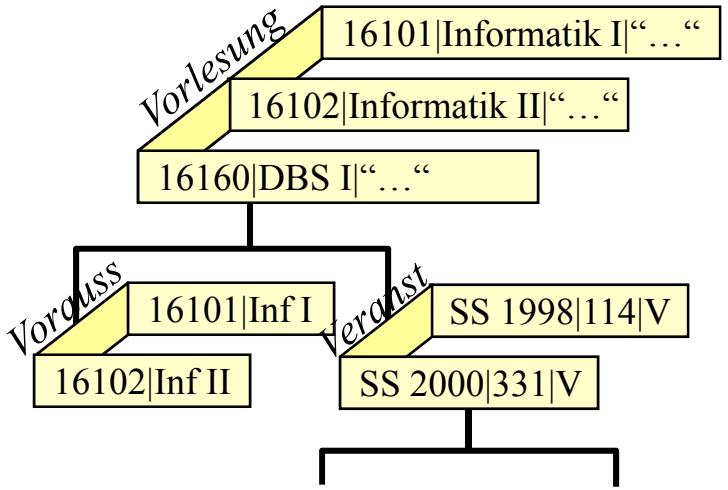
Hierarchisches Datenmodell

- Schema + Daten werden durch Baum strukturiert
- Der gesamte Datenbestand muss hierarchisch repräsentiert werden (oft schwierig)
- Beispiel Lehrveranstaltungen:

Schema:



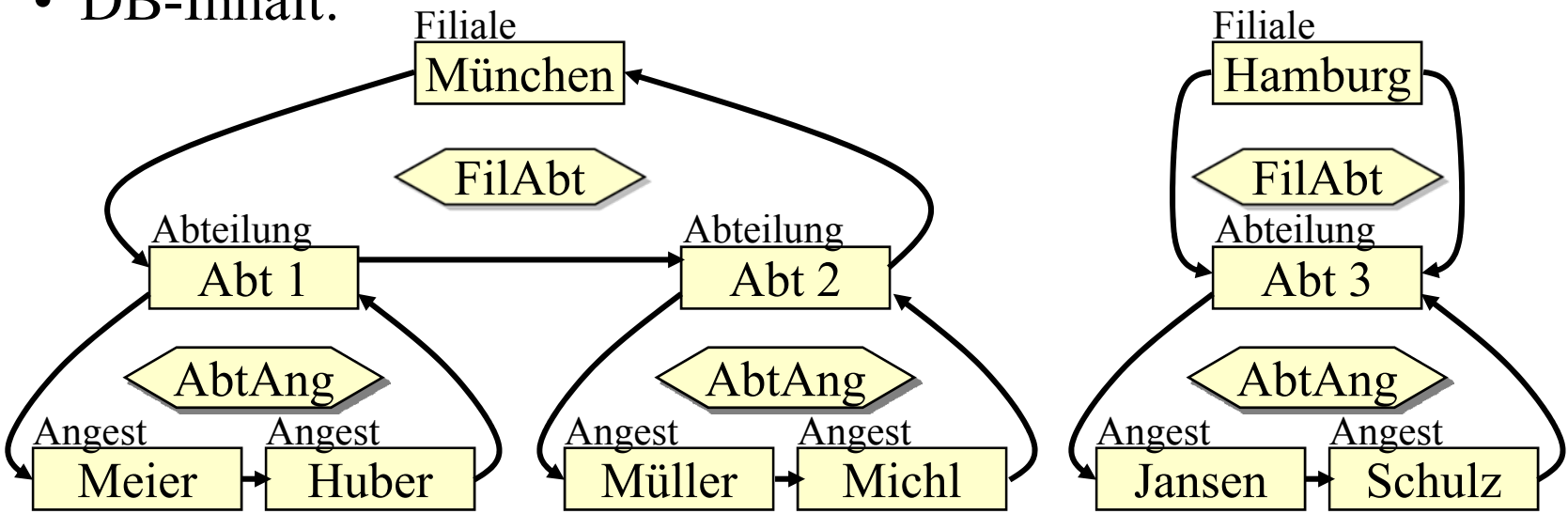
Inhalt:



Netzwerk-Datenmodell

- Schema und Daten werden durch Graphen (Netzwerke) repräsentiert
- Schema:

- DB-Inhalt:



Objekt-Orientiertes Datenmodell

- In der Datenbank werden Objekte, d.h. Ausprägungen von Klassen, die zueinander in verschiedenen Beziehungen stehen (z.B. auch Vererbungsbeziehung), persistent gespeichert.

- Rein objektorientierte Datenbanken haben sich kaum durchgesetzt

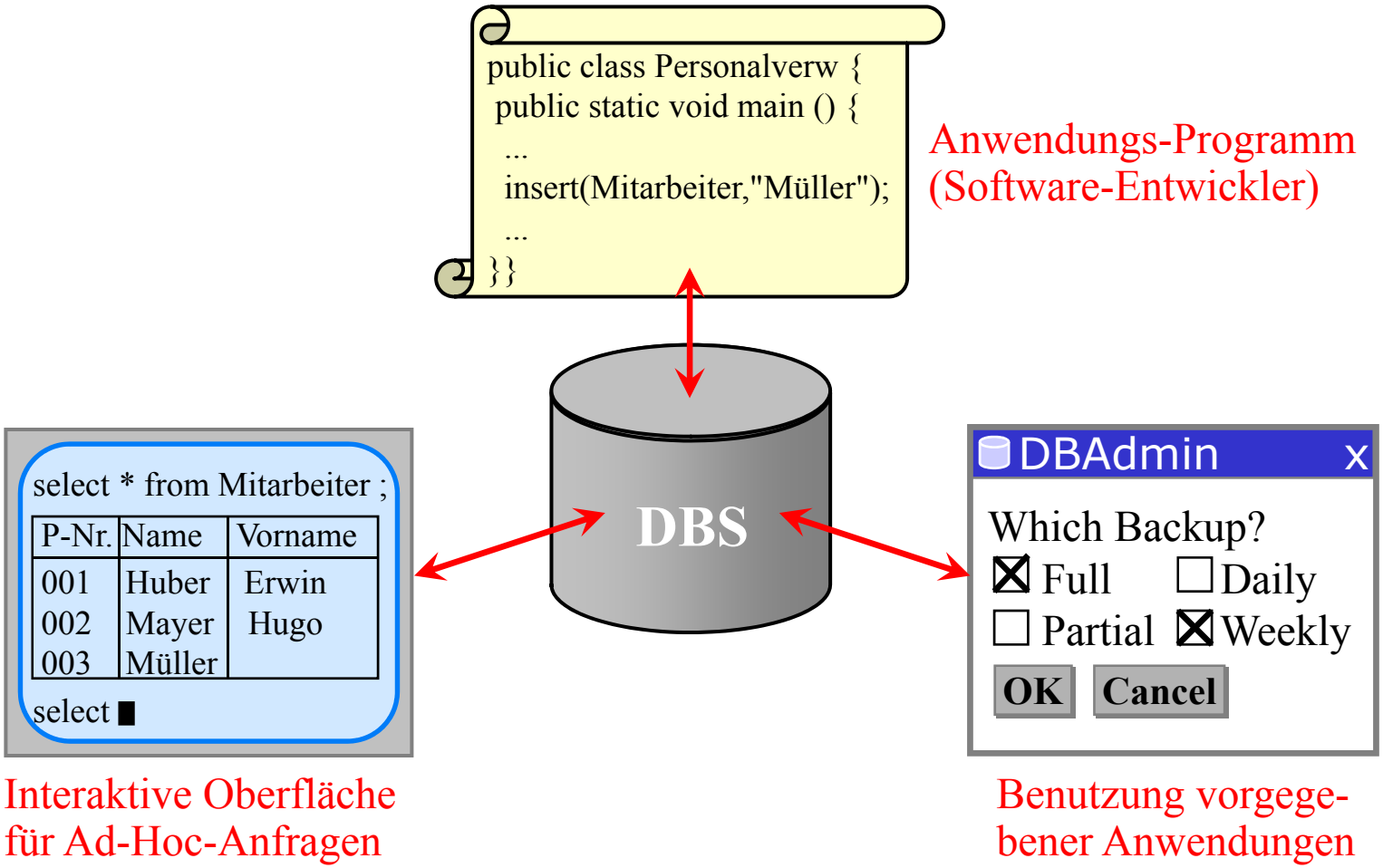
- Relationale Datenbanken haben die Idee aufgenommen und erlauben jetzt auch Speicherung komplexer Objekte (incl. Vererbung) in Relationen

→ Objekt-Relationale Datenbanken

Produkte

- (Objekt-) Relationale Datenbanken:
  - Oracle (Marktführer)
  - IBM DB2
  - Microsoft SQL Server
  - MySQL (Open Source)
  - PostgreSQL (Open Source)
  - keine vollwertigen Datenbanksysteme: dBase, FoxPro, ACCESS
- Nicht-Relationale Datenbanken
  - IMS: Hierarchisches Datenbanksystem (IBM)
  - UDS: Netzwerk-Datenbanksystem (Siemens)
  - Verschiedene objektorientierte DB-Produkte

Verwendung eines DBS

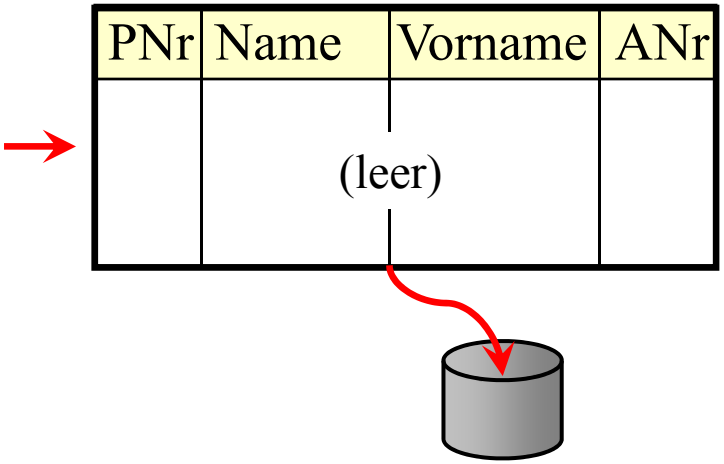


Aus technischer Sicht ist die interaktive Oberfläche ebenfalls ein Anwendungsprogramm, das auf dem DBS aufsetzt

Datenbank-Sprachen

- Data Definition Language (DDL)
  - Deklarationen zur Beschreibung des Schemas
  - Bei relationalen Datenbanken:
    Anlegen und Löschen von Tabellen, Integritätsbedingungen usw.

```
CREATE TABLE Mitarbeiter
  (
    PNr      NUMBER (3),
    Name     CHAR (20),
    Vorname  CHAR (20),
    Anr      NUMBER (2)
  )
```



Datenbank-Sprachen

- Data Manipulation Language (DML)
  - Anweisungen zum Arbeiten mit den Daten in der Datenbank (Datenbank-Zustand)
  - lässt sich weiter unterteilen in Konstrukte
    - zum reinen Lesen der DB (Anfragesprache)
    - zum Manipulieren (Einfügen, Ändern, Löschen) des Datenbankzustands
  - Beispiel: SQL für relationale Datenbanken:

```
SELECT *
FROM Mitarbeiter
WHERE Name = 'Müller'
```

Datenbank-Sprachen

- Wird das folgende Statement (Mitarbeiter-Tab. S. 29)

```
SELECT *
FROM Mitarbeiter
WHERE ANr = 01
```

in die interaktive DB-Schnittstelle eingegeben, dann ermittelt das Datenbanksystem alle Mitarbeiter, die in der Buchhaltungsabteilung (ANr = 01) arbeiten:

PNr	Name	Vorname	ANr
001	Huber	Erwin	01
002	Mayer	Hugo	01

Ergebnis einer Anfrage ist immer eine (neue) Tabelle

Verbindung zur Applikation

- Verwendung einer Programmierbibliothek
  - Dem Programmierer wird eine Bibliothek von Prozeduren/Funktionen zur Verfügung gestellt (Application Programming Interface, API)
  - DDL/DML-Anweisungen als Parameter übergeben
  - Beispiele:
    - OCI: Oracle Call Interface
    - ODBC: Open Database Connectivity
      gemeinsame Schnittstelle an alle Datenbanksysteme
    - JDBC: Java Database Connectivity

Verbindung zur Applikation

- Beispiel: JDBC

```
String q    = "SELECT * FROM Mitarbeiter " +
              "WHERE Name = 'Müller'" ;
Statement s = con.createStatement () ;
ResultSet r = s.executeQuery (q) ;
```

- Die Ergebnistabelle wird an das Java-Programm übergeben.
- Ergebnis-Tupel können dort verarbeitet werden

Verbindung zur Applikation

- Einbettung in eine Wirtssprache
  - DDL/DML-Anweisungen gleichberechtigt neben anderen Sprachkonstrukten
  - Ein eigener Übersetzer (Precompiler) wird benötigt, um die Konstrukte in API-Aufrufe zu übersetzen
  - Beispiele:
    - Embedded SQL für verschiedene Wirtssprachen, z.B. C, C++, COBOL, usw.
    - SQLJ oder JSQL für Java

Verbindung zur Applikation

- Beispiel in SQLJ:

```
public static void main () {
   System.out.println ("Hallöchen") ;
   #sql {SELECT * FROM Mitarbeiter
         WHERE Name = 'Müller'}
   ...
}
```

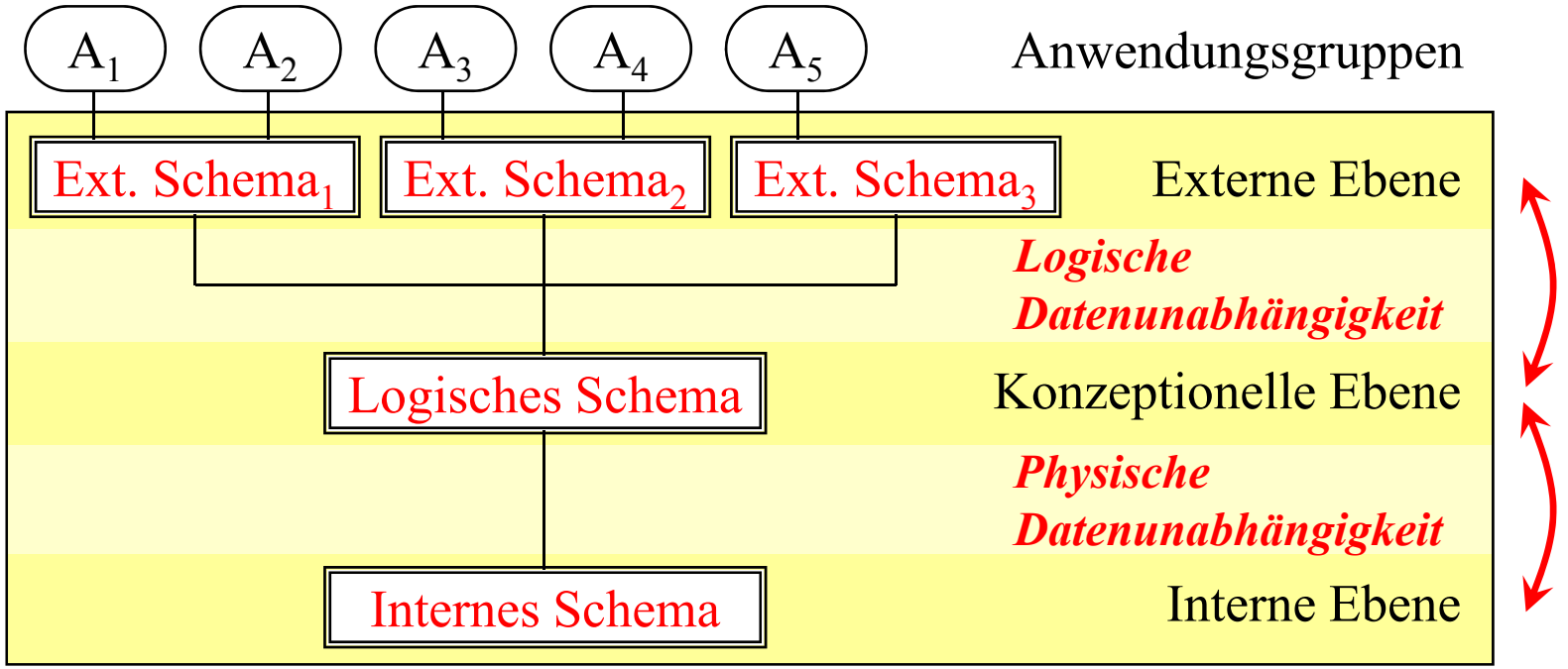
- Die Ergebnistabelle wird an das Java-Programm übergeben.
- Ergebnis-Tupel können dort verarbeitet werden

Architektur eines DBS

Drei-Ebenen-Architektur zur Realisierung von

- **physischer**
- **und logischer**

Datenunabhängigkeit (nach ANSI/SPARC)



Konzeptionelle Ebene

- Logische Gesamtsicht *aller* Daten der DB unabhängig von den einzelnen Applikationen
- Niedergelegt in konzeptionellem (logischem) Schema
- Ergebnis des (logischen) Datenbank-Entwurfs (siehe Kapitel 6)
- Beschreibung aller Objekttypen und Beziehungen
- Keine Details der Speicherung
- Formuliert im Datenmodell des Datenbanksystems
- Spezifiziert mit Hilfe einer Daten-Definitionssprache (Data Definition Language, DDL)

Externe Ebene

- Sammlung der individuellen Sichten aller Benutzer- bzw. Anwendungsgruppen in mehreren externen Schemata
- Ein Benutzer soll keine Daten sehen, die er nicht sehen will (Übersichtlichkeit) oder nicht sehen soll (Datenschutz)
  - Beispiel: Das Klinik-Pflegepersonal benötigt andere Aufbereitung der Daten als die Buchhaltung
- Datenbank wird damit von Änderungen und Erweiterungen der Anwenderschnittstellen abgekoppelt (logische Datenunabhängigkeit)

Interne Ebene

- Das interne Schema beschreibt die systemspezifische Realisierung der DB-Objekte (physische Speicherung), z.B.
  - Aufbau der gespeicherten Datensätze
  - Indexstrukturen wie z.B. Suchbäume
- Das interne Schema bestimmt maßgeblich das Leistungsverhalten des gesamten DBS
- Die Anwendungen sind von Änderungen des internen Schemas nicht betroffen (physische Datenunabhängigkeit)

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 2: Das Relationale Modell

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Charakteristika

- Einführungskapitel:
  Viele Informationen darstellbar als Tabelle
- Die Tabelle (Relation) ist das ausschließliche Strukturierungsmittel des relationalen Datenmodells
- Edgar F. Codd, 1970.
- Grundlage vieler kommerzieller DBS:

ORACLE

IBM Informix

Microsoft SQL Server

Domain

- Ein Wertebereich (oder Typ)
- Logisch zusammengehörige Menge von Werten
- Beispiele:
  - D_1 = Integer
  - D_2 = String
  - D_3 = Date
  - D_4 = {rot, gelb, grün, blau}
  - D_5 = {1, 2, 3}
- Kann *endliche* oder *unendliche* Kardinalität haben

Kartesisches Produkt

- Bedeutung kartesisches Produkt (Kreuzprodukt)?
  Menge von allen möglichen Kombinationen der Elemente der Mengen

- Beispiel ($k = 2$):
  $D_1 = \{1, 2, 3\}$, $D_2 = \{a,b\}$
  $D_1 \times D_2 = \{(1,a), (1,b), (2,a), (2,b), (3,a), (3,b)\}$

- Beispiel ($k = 3$):
  $D_1 = D_2 = D_3 = \mathcal{N}$
  $D_1 \times D_2 \times D_3 = \{(1,1,1),(1,1,2),(1,1,3),...,(1,2,1),...\}$

Relation

- Mathematische Definition:
  Relation R ist Teilmenge des kartesischen Produktes von k Domains $D_1, D_2, ..., D_k$

$$R \subseteq D_1 \times D_2 \times ... \times D_k$$

- Beispiel ($k = 2$):
  $D_1 = \{1, 2, 3\}$, $D_2 = \{a,b\}$

  $R_1 = \{\}$ (leere Menge)
  $R_2 = \{(1,a), (2,b)\}$
  $R_3 = \{(1,a), (2,a), (3,a)\}$
  $R_4 = D_1 \times D_2 = \{(1,a),(1,b),(2,a),(2,b),(3,a),(3,b)\}$

Relation

- Weiteres Beispiel:
  $D_1 = D_2 = \mathcal{N}$
  Relation R_1 = {(1,1),(1,2),(1,3),...,(2,2),(2,3),.....,
  (3,3),(3,4),...(4,4),(4,5),(4,6),....}
  Wie heißt diese mathematische Relation?

$\leq$ $R_1 = \{(x, y) \in \mathcal{N} \times \mathcal{N} \mid x \leq y\}$

- Es gibt endliche und unendliche Relationen
  (wenn mindestens eine Domain unendlich ist)
- In Datenbanksystemen: Nur endliche Relationen
  Unendlich: Nicht darstellbar

Relation

- Die einzelnen Domains lassen sich als Spalten einer Tabelle verstehen und werden als Attribute bezeichnet
- Für $R \subseteq D_1 \times ... \times D_k$ ist k der Grad (Stelligkeit)
- Die Elemente der Relation heißen Tupel: (1,a), (2,a), (3,a) sind drei Tupel vom Grad $k = 2$
- Relation ist Menge von Tupeln d.h. die Reihenfolge der Tupel spielt keine Rolle: {(0,a), (1,b)} = {(1,b), (0,a)}
- Reihenfolge der Attribute ist von Bedeutung: {(a,0), (b,1)} ≠ {(0,a), (1,b)}

Relation

- Relationen sind die mathematische Formalisierung der Tabelle
- Jedes Tupel steht für einen Datensatz
- Die einzelnen Informationen sind in den Attributwerten der Tupel gespeichert
- Die Attribute können auch benannt sein:
  D_1 = Name: String
  D_2 = Vorname: String
- Zeichenketten und Zahlen sind die häufigsten Domains

Relationen-Schema

Alternative Definition:
Relation ist Ausprägung eines Relationen-Schemas

- Geordnetes Relationenschema:
  - k-Tupel aus Domains (Attribute)
  - Attribute können benannt sein
  - Attribute werden anhand ihrer Position im Tupel ref.

  $R = (A_1: D_1, \ldots A_k: D_k)$
- Domänen-Abbildung (ungeordnetes Rel.-Sch.):
  - Relationenschema R ist Menge von Attributnamen:
  - Jedem Attributnamen A_i ist Domäne D_i zugeordnet:
  - Attribute werden anhand ihres Namens referenziert

  $R = \{A_1, \ldots. A_k\}$ mit $\text{dom}(A_i) = D_i,\ 1 \leq i \leq k$

Relationen-Schema

- Beispiel: Städte-Relation

Städte

Name	Einwohner	Land
München	1.211.617	Bayern
Bremen	535.058	Bremen
Passau	49.800	Bayern

- Als geordnetes Relationenschema:

  Schema: R = (Name: String, Einwohner: Integer, Land: String)

  Ausprägung: r = {(München, 1.211.617, Bayern), (Bremen, 535.058, Bremen), (Passau, 49.800, Bayern)}

- Als Relationenschema mit Domänenabbildung:

  Schema: R = {Name, Einwohner, Land}
  mit dom(Name) = String, dom(Einwohner) = Integer, ...

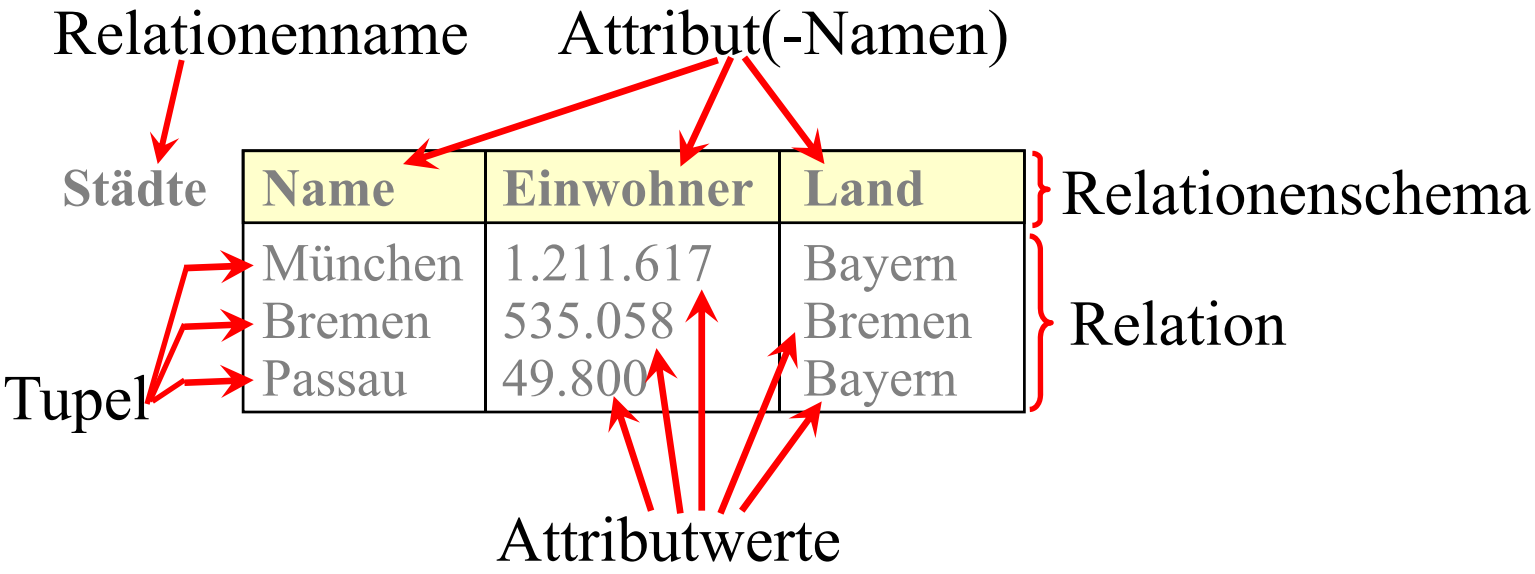
  Ausprägung: $r = \{t_1, t_2, t_3\}$
  mit t_1(Name) = München, t_1(Einwohner) = 1.211.617,...

Diskussion

- Vorteil von geordnetem Relationenschema:
  - Prägnanter aufzuschreiben.
    Wichtig z.B. beim Einfügen neuer Tupel:
    t_3 = (Passau, 49.800, Bayern)
    vergleiche: t_3 (Name) = Passau; t_3 (Einwohner) = ...
- Nachteil von geordnetem Relationenschema:
  - Einschränkungen bei logischer Datenunabhängigkeit:
    Applikationen sensibel bzgl. Einfügung neuer Attribute (nur am Ende!)
- Definitionen gleichwertig
  (lassen sich ineinander überführen)
- Wir verwenden beide Ansätze

Begriffe

- Relation: Ausprägung eines Relationenschemas
- Datenbankschema: Menge von Relationenschemata
- Datenbank: Menge von Relationen (Ausprägungen)



Duplikate

- Relationen sind Mengen von Tupeln.
  Konsequenzen:
  - Reihenfolge der Tupel irrelevant (wie bei math. Def)
  - Es gibt keine Duplikate (gleiche Tupel) in Relationen:
    {(0,a), (0,a), (0,a), (1,b)} = {(0,a), (1,b)}

- Frage: Gilt dies auch für die Spalten beim ungeordneten Relationenschema $R = \{A_1,...,A_k\}$?
  - Reihenfolge der Spalten ist irrelevant
    (das ist gerade das besondere am ungeordneten RS)
  - Duplikate treten nicht auf, weil alle Attribut-Namen verschieden

Schlüssel

- Tupel müssen eindeutig identifiziert werden
- Warum? Z.B. für Verweise:

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	
002	Mayer	Hugo	
003	Müller	Anton	

Abteilungen

ANr	Abteilungsname
01	Buchhaltung
02	Produktion
03	Marketing

- Objektidentifikation in Java:
  Mit Referenz (Adresse im Speicher)
- Im relationalen Modell werden Tupel anhand von Attributwerten identifiziert
- Ein/mehrere Attribute als Schlüssel kennzeichnen
- Konvention: Schlüsselattribut(e) unterstreichen!

Schlüssel

Beispiel: **PNr** und **ANr** werden Primärschlüssel:

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	
002	Mayer	Hugo	
003	Müller	Anton	

Abteilungen

ANr	Abteilungsname
01	Buchhaltung
02	Produktion
03	Marketing

- Damit müssen diese Attributswerte eindeutig sein!
- Verweis durch Wert dieses Schlüsselattributs:

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	01
002	Mayer	Hugo	01
003	Müller	Anton	02

Abteilungen

ANr	Abteilungsname
01	Buchhaltung
02	Produktion
03	Marketing

Zusammengesetzter Schlüssel

- Oft ist ein einzelnes Attribut nicht ausreichend, um die Tupel eindeutig zu identifizieren
- Beispiel:

Lehrveranstaltung

VNr	Titel	Semester
012	Einführung in die Informatik	WS 2001/02
012	Einführung in die Informatik	WS 2002/03
013	Medizinische Informationssyst.	WS 2001/02
...	...	...

- Schlüssel: **(VNr, Semester)**
- Anmerkung: Warum ist dies ein schlechtes DB-Design?
  Nicht redundanzfrei:
  Der Titel ist mehrfach in der Datenbank gespeichert

Schlüssel: Formale Definition

Definition:

- Eine Teilmenge *S* der Attribute eines Relationenschemas *R* heißt Schlüssel, wenn gilt:
  - Eindeutigkeit
    Keine Ausprägung von *R* kann zwei verschiedene Tupel enthalten, die sich in allen Attributen von *S* gleichen.
  - Minimalität
    Keine echte Teilmenge von *S* erfüllt bereits die Bedingung der Eindeutigkeit

Schlüssel: Formale Definition

In noch formalerer Notation:

Definition:

- Sei r eine Relation über dem Relationenschema R und $S \subseteq R$ eine Auswahl von Attributen.
- $t[S]$ bezeichne ein Tupel t eingeschränkt auf die Attribute aus S (alle anderen Attribute gestrichen)
- (1) Eindeutigkeit:
  $\forall$ möglichen Ausprägungen r und Tupel $t_1, t_2 \in r$ gilt:
  $t_1 \neq t_2 \Rightarrow t_1[S] \neq t_2[S]$
- (2) Minimalität:
  Für alle Attributmengen S und T, die (1) erfüllen, gilt:
  $T \subseteq S \Rightarrow T = S$

Schlüssel: Beispiele

- Gegeben sei die folgende Relation:

Lehrveranst
(t_1=)
(t_2=)
(t_3=)

LNr	VNr	Titel	Semester
1	012	Einführung in die Informatik	WS 2001/02
2	012	Einführung in die Informatik	WS 2002/03
3	013	Medizinische Informationssyst.	WS 2001/02
...	...	...	...

- {VNr} ist kein Schlüssel
  Nicht eindeutig: $t_1 \neq t_2$ **aber** $t_1[\text{VNr}] = t_2[\text{VNr}] = 012$
- {Titel} ist kein Schlüssel
  (gleiche Begründung)
- {Semester} ist kein Schlüssel
  Nicht eindeutig: $t_1 \neq t_3$ **aber** $t_1[\text{Semester}] = t_3[\text{Semester}]$

Schlüssel: Beispiele

Lehrveranst

(t_1=)
(t_2=)
(t_3=)

LNr	VNr	Titel	Semester
1	012	Einführung in die Informatik	WS 2001/02
2	012	Einführung in die Informatik	WS 2002/03
3	013	Medizinische Informationssyst.	WS 2001/02
...	...	...	...

- {LNr} ist Schlüssel !!!
  Eindeutigkeit: Alle t_i[LNr] sind paarweise verschieden,
  d.h. $t_1[LNr] \neq t_2[LNr]$, $t_1[LNr] \neq t_3[LNr]$, $t_2[LNr] \neq t_3[LNr]$
  Minimalität: Trivial, weil 1 Attribut kürzeste Möglichkeit

- {LNr, VNr} ist kein Schlüssel
  Eindeutigkeit: Alle t_i[LNr,VNr] paarweise verschieden.
  Nicht minimal, da echte Teilmenge $\{LNr\} \subset \{LNr, VNr\}$ ($\neq$) die Eindeutigkeit bereits gewährleistet, s.o.

Schlüssel: Beispiele

Lehrveranst

	LNr	VNr	Titel	Semester
(t_1=)	1	012	Einführung in die Informatik	WS 2001/02
(t_2=)	2	012	Einführung in die Informatik	WS 2002/03
(t_3=)	3	013	Medizinische Informationssyst.	WS 2001/02
	...	...	...	...

- {VNr, Semester} ist Schlüssel !!!

  Eindeutigkeit: Alle t_i[VNr, Semester] paarw. verschieden:
  - t_1 [VNr, Semester] = (012, WS 2001/02) $\neq$
  - t_2 [VNr, Semester] = (012, WS 2002/03) $\neq$
  - t_3 [VNr, Semester] = (013, WS 2001/02) $\neq$

  Minimalität:
  Weder {VNr} noch {Semester} gewährleisten Eindeutigkeit (siehe vorher). Dies sind alle echten Teilmengen.

Primärschlüssel

- Minimalität bedeutet **nicht**:
  Schlüssel mit den wenigsten Attributen
- Sondern Minimalität bedeutet:
  Keine überflüssigen Attribute sind enthalten
  (d.h. solche, die zur Eindeutikeit nichts beitragen)
- Manchmal gibt es mehrere verschiedene Schlüssel
  - {LNr}
  - {VNr, Semester}
- Diese nennt man auch **Schlüsselkandidaten**
- Man wählt einen dieser Kandidaten aus als sog.
  Primärschlüssel (oft den kürzesten, nicht immer)

Schlüssel: Semantische Eigenschaft

- Die Eindeutigkeit bezieht sich **nicht** auf die aktuelle Ausprägung einer Relation *r*
- Sondern immer auf die **Semantik** der realen Welt

Mitarbeiter

PNr	Name	Gehalt
001	Müller	1700 €
002	Mayer	2172 €
003	Huber	3189 €
004	Schulz	2171 €

  - Bei der aktuellen Relation wären sowohl {PNr} als auch {Name} und {Gehalt} eindeutig.
  - Aber es ist möglich, dass mehrere Mitarbeiter mit gleichem Namen und/oder Gehalt eingestellt werden
  - {PNr} ist **für jede mögliche** Ausprägung eindeutig

Tabellendefinition in SQL

- Definition eines Relationenschemas:

```
CREATE TABLE n
  (
      a1 d1 c1,
      a2 d2 c2,
      ...
      ak dk ck
  )
```

← *n* Name der Relation

← Definition des ersten Attributs

← Definition des Attributs Nr. *k*

- hierbei bedeuten...
  - a_i der Name des Attributs Nr. i
  - d_i der Typ (die Domain) des Attributs
  - c_i ein optionaler Constraint für das Attribut

Basis-Typen in SQL

Der SQL-Standard kennt u.a. folgende Datentypen:

- **integer** oder auch **interger4**, **int**
- **smallint** oder **integer2**
- **float** (*p*) oder auch **float**
- **decimal** (*p*,*q*) und **numeric** (*p*,*q*)
  mit *p* Stellen, davon *q* Nachkommast.
- **character** (*n*), **char** (*n*) für Strings fester Länge *n*
- **character varying** (*n*), **varchar** (*n*): variable Strings
- **date**, **time**, **timestamp** für Datum und Zeit

Zusätze bei Attributdefinitionen

- Einfache Zusätze (Integritätsbedingungen) können unmittelbar hinter einer Attributdefinition stehen:
  - **not null**: Das Attribut darf nicht undefiniert sein in DBS: undefinierte Werte heissen **null**-Werte
  - **primary key**: Das Attribut ist Primärschlüssel (nur bei 1-Attribut-Schlüsseln)
  - **check** *f*:
    Die Formel *f* wird bei jeder Einfügung überprüft
  - **references** t_1 (a_1):
    Ein Verweis auf Attribut a_1 von Tabelle t_1
    Verschiedene Zusatzoptionen:
    - **on delete cascade**
    - **on delete set null**
  - **default** w_1: Wert w_1 ist Default, wenn unbesetzt.

Integritätsbedingungen

- Zusätze, die keinem einzelnen Attribut zugeordnet sind, stehen mit Komma abgetrennt in extra Zeilen
  - **primary key** (s_1, s_2, ...):
    Zusammengesetzter Primärschlüssel
  - **foreign key** (s_1, s_2, ...) **references** t_1 (...)
    Verweis auf zusammengesetzten Schlüssel in t_1
  - **check** f
- Anmerkung:
  SQL ist case-insensitiv:
  - im Ggs. zu Java hat die Groß-/Kleinschreibung weder bei Schlüsselworten noch bei Bezeichnern Bedeutung

Beispiel Tabellendefinition

- Zusammengesetzter Primärschlüssel {VNr, Semester}:

```
create table Lehrveranst
  (
    LNr         integer          not null,
    VNr         integer          not null,
    Titel       varchar(50),
    Semester    varchar(20)      not null,
    primary key (VNr, Semester)
  )
```

- Alternative mit einfachem Primärschlüssel {LNr}:

```
create table Lehrveranst2
  (
    LNr         integer          primary key,
    VNr         integer          not null,
    Titel       varchar(50),
    Semester    varchar(20)      not null
  )
```

Beispiel Tabellendefinition

- Tabelle für Dozenten:

```
create table Dozenten
  (
    DNr        integer          primary key,
    Name       varchar(50),
    Geburt     date,
  )
```

- Verwendung von Fremdschlüsseln:

```
create table Haelt
  (
    Dozent     integer          references Dozenten (DNr)
                                on delete cascade,
    VNr        integer          not null,
    Semester   varchar(20)      not null,
    primary key (Dozent, VNr, Semester),
    foreign key (VNr, Semester) references Lehrveranst
  )
```

Beispiel Tabellendefinition

- Das Schlüsselwort **on delete cascade** in *Haelt* führt dazu, daß bei Löschen eines *Dozenten* auch entsprechende Tupel in *Haelt* gelöscht werden
- Weitere Konstrukte der Data Definition Language:
  - **drop table** n_1
    Relationen-Schema n_1 wird mit allen evtl. vorhandenen Tupeln gelöscht.
  - **alter table** n_1 **add** $(a_1\ d_1\ c_1, a_2\ d_2\ c_2, \ldots)$
    - Zusätzliche Attribute oder Integritätsbedingungen werden (rechts) an die Tabelle angehängt
    - Bei allen vorhandenen Tupeln Null-Werte
  - **alter table** n_1 **drop** $(a_1, a_2, \ldots)$
  - **alter table** n_1 **modify** $(a_1\ d_1\ c_1, a_2\ d_2\ c_2, \ldots)$

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 3: Die Relationale Algebra

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Arbeiten mit Relationen

- Es gibt viele *formale* Modelle, um...
  - mit Relationen zu arbeiten
  - Anfragen zu formulieren
- Wichtigste Beispiele:
  - Relationale Algebra
  - Relationen-Kalkül
- Sie dienen als theoretisches Fundament für konkrete Anfragesprachen wie
  - SQL: Basiert i.w. auf der relationalen Algebra
  - QBE (= Query By Example) und Quel: Basieren auf dem Relationen-Kalkül

Begriff Relationale Algebra

- Mathematik:
  - Algebra ist eine Operanden-Menge mit Operationen
  - Abgeschlossenheit: Werden Elemente der Menge mittels eines Operators verknüpft, ist das Ergebnis wieder ein Element der Menge
  - Beispiele
    - Natürliche Zahlen mit Addition, Multiplikation
    - Zeichenketten mit Konkatenation
    - Boolsche Algebra: Wahrheitswerte mit $\wedge$, $\vee$, $\neg$
    - Mengen-Algebra:
      - Wertebereich: die Menge (*Klasse*) der Mengen
      - Operationen z.B. $\cup$, $\cap$, – (Differenzmenge)

Begriff Relationale Algebra

- Relationale Algebra:
  - „Rechnen mit Relationen“
  - Was sind hier die Operanden? Relationen (Tabellen)
  - Beispiele für Operationen?
    - Selektion von Tupeln nach Kriterien (Anr = 01)
    - Kombination mehrerer Tabellen
  - Abgeschlossenheit:
    Ergebnis einer Anfrage ist immer eine (neue) Relation (oft ohne eigenen Namen)
  - Damit können einfache Terme der relationalen Algebra zu komplexeren zusammengesetzt werden

Grundoperationen

- 5 Grundoperationen der Relationalen Algebra:
  - Vereinigung: $R = S \cup T$
  - Differenz: $R = S - T$
  - Kartesisches Produkt (Kreuzprodukt): $R = S \times T$
  - Selektion: $R = \sigma_F(S)$
  - Projektion: $R = \pi_{A,B,\ldots}(S)$
- Mit den Grundoperationen lassen sich weitere Operationen, (z.B. die Schnittmenge) nachbilden
- Manchmal wird die Umbenennung von Attributen als 6. Grundoperation bezeichnet

Vereinigung und Differenz

- Diese Operationen sind nur anwendbar, wenn die Schemata der beiden Relationen *S* und *T* übereinstimmen
- Die Ergebnis-Relation *R* bekommt Schema von *S*
- Vereinigung: $R = S \cup T = \{t \mid t \in S \vee t \in T\}$
- Differenz: $R' = S - T = \{t \mid t \in S \wedge t \notin T\}$
- Was wissen wir über die *Kardinalität* des Ergebnisses (Anzahl der Tupel von *R*)?

$$|R| = |S \cup T| \leq |S| + |T|$$

$$|R'| = |S - T| \geq |S| - |T|$$

Beispiel

Mitarbeiter:

Name	Vorname
Huber	Egon
Maier	Wolfgang
Schmidt	Helmut

Studenten:

Name	Vorname
Müller	Heinz
Schmidt	Helmut

Alle Personen, die Mitarbeiter oder Studenten sind:

Mitarbeiter ∪ Studenten:

Name	Vorname
Huber	Egon
Maier	Wolfgang
Schmidt	Helmut
Müller	Heinz
~~Schmidt~~	~~Helmut~~

Duplikat-Elimination!

Alle Mitarbeiter ohne diejenigen, die auch Studenten sind:

Mitarbeiter – Studenten:

Name	Vorname
Huber	Egon
Maier	Wolfgang

Kartesisches Produkt

Wie in Kapitel 2 bezeichnet das Kreuzprodukt

$$R = S \times T$$

die Menge aller möglichen Kombinationen von Tupeln aus S und T

- Seien $a_1, a_2, ..., a_s$ die Attribute von S und $b_1, b_2, ..., b_t$ die Attribute von T
- Dann ist $R = S \times T$ die folgende Menge (Relation):
  $\{(a_1,...,a_s,b_1,...,b_t) \mid (a_1,...,a_s) \in S \wedge (b_1,...,b_t) \in T\}$
- Für die Anzahl der Tupel gilt:

$$|S \times T| = |S| \cdot |T|$$

Beispiel

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	01
002	Mayer	Hugo	01
003	Müller	Anton	02

Abteilungen

ANr	Abteilungsname
01	Buchhaltung
02	Produktion

Mitarbeiter × Abteilungen

PNr	Name	Vorname	Abteilung	ANr	Abteilungsname
001	Huber	Erwin	01	01	Buchhaltung
001	Huber	Erwin	01	02	Produktion
002	Mayer	Hugo	01	01	Buchhaltung
002	Mayer	Hugo	01	02	Produktion
003	Müller	Anton	02	01	Buchhaltung
003	Müller	Anton	02	02	Produktion

Frage: Ist dies richtig?

Selektion

- Mit der Selektion $R = \sigma_F(S)$ werden diejenigen Tupel aus einer Relation S ausgewählt, die eine durch die logische Formel F vorgegebene Eigenschaft erfüllen
- R bekommt das gleiche Schema wie S
- Die Formel F besteht aus:
  - Konstanten („Meier“)
  - Attributen: Als Name (PNr) oder Nummer ($1)
  - Vergleichsoperatoren: $=$, $<$, $\leq$, $>$, $\geq$, $\neq$
  - Boole‘sche Operatoren: $\wedge$, $\vee$, $\neg$
- Formel F wird für jedes Tupel von S ausgewertet

Beispiel

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	01
002	Mayer	Hugo	01
003	Müller	Anton	02

Alle Mitarbeiter von Abteilung 01:

$\sigma_{Abteilung=01}(Mitarbeiter)$

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	01
002	Mayer	Hugo	01

Kann jetzt die Frage von S. 9 beantwortet werden?

Beispiel

Mitarbeiter × Abteilungen

PNr	Name	Vorname	Abteilung	ANr	Abteilungsname
001	Huber	Erwin	01	01	Buchhaltung
001	Huber	Erwin	01	02	Produktion
002	Mayer	Hugo	01	01	Buchhaltung
002	Mayer	Hugo	01	02	Produktion
003	Müller	Anton	02	01	Buchhaltung
003	Müller	Anton	02	02	Produktion

$\sigma_{\text{Abteilung=ANr}}$(Mitarbeiter × Abteilungen)

PNr	Name	Vorname	Abteilung	ANr	Abteilungsname
001	Huber	Erwin	01	01	Buchhaltung
002	Mayer	Hugo	01	01	Buchhaltung
003	Müller	Anton	02	02	Produktion

Die Kombination aus Selektion und Kreuzprodukt heißt Join

Projektion

- Die Projektion $R = \pi_{A,B,...}(S)$ erlaubt es,
  - Spalten einer Relation auszuwählen
  - bzw. nicht ausgewählte Spalten zu streichen
  - die Reihenfolge der Spalten zu verändern
- In den Indizes sind die selektierten Attributsnamen oder -Nummern ($1) aufgeführt
- Für die Anzahl der Tupel des Ergebnisses gilt:

$$| \pi_{A,B,...}(S) | \leq | S |$$

Grund: Nach dem Streichen von Spalten können Duplikat-Tupel entstanden sein

Projektion: Beispiel

Mitarbeiter

PNr	Name	Vorname	Abteilung
001	Huber	Erwin	01
002	Mayer	Josef	01
003	Müller	Anton	02
004	Mayer	Maria	01

$\pi_{\text{Name, Abteilung}}(\text{Mitarbeiter}) = \ldots$

Zwischenergebnis:

Name	Abteilung
Huber	01
Mayer	01
Müller	02
Mayer	01

Duplikate

Elimination →

Name	Abteilung
Huber	01
Mayer	01
Müller	02

Duplikatelimination

- Erforderlich nach...
  - Projektion
  - Vereinigung

  } „billige“ Basisoperationen, aber...
- Wie funktioniert Duplikatelimination?

```
for (int i = 0 ; i < R.length ; i++)
    for (int j = 0 ; j < i ; j++)
        if (R[i] == R[j])
            // R[j] aus Array löschen
```

- Aufwand? n=R.length: O(n^2)
- Besserer Algorithmus mit Sortieren: O(n log n)

⇒ An sich billige Operationen werden durch Duplikatelimination teuer

Beispiel-Anfragen

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- Bestimme alle Großstädte (≥500.000) und ihre Einwohner

  $\pi_{SName,SEinw}(\sigma_{SEinw \geq 500.000}(Städte))$

- In welchem Land liegt die Stadt Passau?

  $\pi_{Land}(\sigma_{SName=Passau}(Städte))$

- Bestimme die Namen aller Städte, deren Einwohnerzahl die eines beliebigen Landes übersteigt:

  $\pi_{SName}(\sigma_{SEinw>LEinw}(Städte \times Länder))$

Beispiel-Anfragen

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- Finde alle Städtenamen in CDU-regierten Ländern

  $\pi_{SName}(\sigma_{Land=LName}(Städte \times \sigma_{Partei=CDU}(Länder)))$

  oder auch:

  $\pi_{SName}(\sigma_{Land=Lname \wedge Partei=CDU}(Städte \times Länder))$

- Welche Länder werden von der SPD *allein* regiert

  $\pi_{LName}(\sigma_{Partei=SPD}(Länder)) - \pi_{LName}(\sigma_{Partei\neq SPD}(Länder))$

Abgeleitete Operationen

- Eine Reihe nützlicher Operationen lassen sich mit Hilfe der 5 Grundoperationen ausdrücken:
  - Durchschnitt $R = S \cap T$
  - Quotient $R = S \div T$
  - Join $R = S \bowtie T$

Durchschnitt

- Idee: Finde gemeinsame Elemente in zwei Relationen (Schemata müssen übereinstimmen):

$$R' = S \cap T = \{t \mid t \in S \wedge t \in T\}$$

- Beispiel:
  Welche Personen sind gleichzeitig Mitarbeiter und Student?

Mitarbeiter:

Name	Vorname
Huber	Egon
Maier	Wolfgang
Schmidt	Helmut

Studenten:

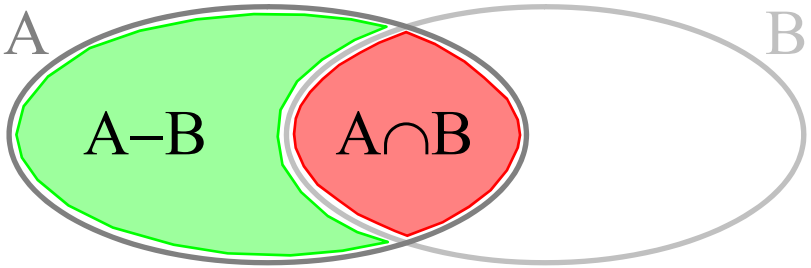
Name	Vorname
Müller	Heinz
Schmidt	Helmut

Mitarbeiter ∩ Studenten:

Name	Vorname
Schmidt	Helmut

Durchschnitt

- Implementierung der Operation „Durchschnitt“ mit Hilfe der Grundoperation „Differenz“:



- $A \cap B = A - (A - B)$
- Achtung! Manche Lehrbücher definieren:
  - Durchschnitt ist Grundoperation
  - Differenz ist abgeleitete Operation

  (Definition gleichwertig, also genauso möglich)

Quotient

- Dient zur Simulation eines Allquantors
- Beispiel:

R_1

Programmierer	Sprache
Müller	Java
Müller	Basic
Müller	C++
Huber	C++
Huber	Java

R_2

Sprache
Basic
C++
Java

- Welche Programmierer programmieren in **allen** Sprachen?

$R_1 \div R_2$

Programmierer
Müller

- Umkehrung des kartesischen Produktes (daher: *Quotient*)

Join

- Wie vorher erwähnt:
  Selektion über Kreuzprodukt zweier Relationen
  - Theta-Join (Θ): $R \bowtie_{A \Theta B} S$

    Allgemeiner Vergleich:
    Θ ist einer der Operatoren $=, <, \leq, >, \geq, \neq$
  - Equi-Join: $R \bowtie_{A = B} S$
  - Natural Join: $R \bowtie S$:
    - Ein Equi-Join bezüglich aller gleichbenannten Attribute in R und S wird durchgeführt.
    - Gleiche Spalten werden gestrichen (Projektion)

Join

- Implementierung mit Hilfe der Grundoperationen

$$R \underset{A\,\Theta\,B}{\bowtie} S = \sigma_{A\,\Theta\,B}(R \times S)$$

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- Finde alle Städtenamen in CDU-regierten Ländern

$$\pi_{SName}(\text{Städte} \underset{Land=LName}{\bowtie} \sigma_{Partei=CDU}(\text{Länder}))$$

- Bestimme die Namen aller Städte, deren Einwohnerzahl die eines beliebigen Landes übersteigt:

$$\pi_{SName}(\text{Städte} \underset{SEinw>LEinw}{\bowtie} \text{Länder}))$$

SQL

- Die wichtigste Datenbank-Anfragesprache SQL beruht wesentlich auf der relationalen Algebra
- Grundform einer Anfrage*:

Projektion	→	SELECT	⟨Liste von Attributsnamen bzw. *⟩
Kreuzprodukt	→	FROM	⟨ein oder mehrere Relationennamen⟩
Selektion	→	[WHERE	⟨Bedingung⟩]

- Mengenoperationen:

```
SELECT ... FROM ... WHERE
UNION
SELECT ... FROM ... WHERE
```

* SQL ist case-insensitive: SELECT = select = SeLeCt

SQL

- Hauptunterschied zwischen SQL und rel. Algebra:
  - Operatoren bei SQL nicht beliebig schachtelbar
  - Jeder Operator hat seinen festen Platz
- Trotzdem:
  - Man kann zeigen, daß jeder Ausdruck der relationalen Algebra gleichwertig in SQL formuliert werden kann
  - Die feste Anordnung der Operatoren ist also keine wirkliche Einschränkung (Übersichtlichkeit)
  - Man sagt, SQL ist *relational vollständig*
- Weitere Unterschiede:
  - Nicht immer werden Duplikate eliminiert (Projektion)
  - zus. Auswertungsmöglichkeiten (Aggregate, Sortieren)

SELECT

- Entspricht Projektion in der relationalen Algebra
- Aber: Duplikatelimination nur, wenn durch das Schlüsselwort DISTINCT explizit verlangt
- Syntax:

```
SELECT * FROM ...                    -- Keine Projektion
SELECT A1, A2, ... FROM ...          -- Projektion ohne
                                     -- Duplikatelimination
SELECT DISTINCT A1, A2, ...          -- Projektion mit
                                     -- Duplikatelimination
```

- Bei der zweiten Form kann die Ergebnis„*relation*“ also u.U. Duplikate enthalten
- Grund: Performanz

SELECT

- Bei den Attributen A_1, A_2, ... lässt sich angeben...
  - Ein Attributname einer beliebigen Relation, die in der FROM-Klausel angegeben ist
  - Ein skalarer Ausdruck, der Attribute und Konstanten mittels arithmetischer Operationen verknüpft
  - Im Extremfall: Nur eine Konstante
  - Aggregationsfunktionen (siehe später)
  - Ein Ausdruck der Form A_1 AS A_2: A_2 wird der neue Attributname (Spaltenüberschrift)
- Beispiel:

```
select  pname
        preis*13.7603 as oespr,
        preis*kurs as usdpr,
        'US$' as currency
from    produkt, waehrungen....
```

pname	oespr	usdpr	currency
nagel	6.88	0.45	US$
dübel	1.37	0.09	US$
...			

FROM

- Enthält mindestens einen Eintrag der Form R_1
- Enthält die FROM-Klausel mehrere Einträge
  - FROM R_1, R_2, ...

  so wird das kartesische Produkt gebildet:
  - $R_1 \times R_2 \times ...$
- Enthalten zwei verschiedene Relationen R_1, R_2 ein Attribut mit gleichem Namen, dann ist dies in der SELECT- und WHERE-Klausel mehrdeutig
- Eindeutigkeit durch vorangestellten Relationennamen:

```
SELECT   Mitarbeiter.Name, Abteilung.Name, ...
FROM     Mitarbeiter, Abteilung
WHERE    ...
```

FROM

- Man kann Schreibarbeit sparen, indem man den Relationen temporär (innerhalb der Anfrage) kurze Namen zuweist (Alias-Namen):

```
SELECT    m.Name, a.Name, ...
FROM      Mitarbeiter m, Abteilung a
WHERE     ...
```

- Dies lässt sich in der SELECT-Klausel auch mit der Sternchen-Notation kombinieren:

```
SELECT    m.*, a.Name AS Abteilungsname, ...
FROM      Mitarbeiter m, Abteilung a
WHERE     ...
```

- Manchmal Self-Join einer Relation mit sich selbst:

```
SELECT    m1.Name, m2.Name, ...
FROM      Mitarbeiter m1, Mitarbeiter m2
WHERE     ...
```

WHERE

- Entspricht der Selektion der relationalen Algebra
- Enthält genau eine logische Formel (Boolean)
- Das logische Prädikat besteht aus
  - Vergleichen zwischen Attributwerten und Konstanten
  - Vergleichen zwischen verschiedenen Attributen (Join)
  - Vergleichsoperatoren* Θ: = , < , <= , > , >= , <>
  - Auch: A_1 BETWEEN x AND y
    (äquivalent zu A_1 >= x AND A_1 <= y)
  - Test auf *Wert undefiniert*: A_1 IS NULL/IS NOT NULL
  - Inexakter Stringvergleich: A_1 LIKE ‘Datenbank%‘
  - A_1 IN (2, 3, 5, 7, 11, 13)

*Der Gleichheitsoperator wird **nicht** etwa wie in Java verdoppelt

WHERE

- Innerhalb eines Prädikates: Skalare Ausdrücke:
  - Numerische Werte/Attribute mit +, –, *, / verknüpfbar
  - Strings: char_length, Konkatenation || und substring
  - Spezielle Operatoren für Datum und Zeit
  - Übliche Klammernsetzung.
- Einzelne Prädikate können mit AND, OR, NOT zu komplexeren zusammengefasst werden
- Idee: Alle Tupel des kartesischen Produktes aus der FROM-Klausel werden getestet, ob sie das log. Prädikat erfüllen.
- Effizientere Ausführung möglich mit Index

WHERE

- Inexakte Stringsuche: A_1 LIKE ‘Datenbank%‘
  - bedeutet: Alle Datensätze, bei denen Attribut A_1 mit dem Präfix *Datenbank* beginnt.
  - Entsprechend: A_1 LIKE ‘%Daten%‘
  - In dem Spezialstring hinter LIKE ...
    - % steht für einen beliebig belegbaren Teilstring
    - _ steht für ein einzelnes frei belegbares Zeichen
- Beispiel:

  Alle Mitarbeiter, deren Nachname auf ‘er‘ endet:

  select * from mitarbeiter
  where name **like** ‘%er ‘

Mitarbeiter

PNr	Name	Vorname	ANr
001	Huber	Erwin	01
002	Mayer	Josef	01
003	Müller	Anton	02
004	Schmidt	Helmut	01

Join

- Normalerweise wird der Join wie bei der relationalen Algebra als Selektionsbedingung über dem kartesischen Produkt formuliert.
- Beispiel: Join zwischen Mitarbeiter und Abteilung
  select * from Mitarbeiter m, Abteilungen a **where** m.ANr = a.ANr
- In neueren SQL-Dialekten auch möglich:
  - **select * from** Mitarbeiter m **join** Abteilungen a **on** a.ANr=m.ANr
  - **select * from** Mitarbeiter **join** Abteilungen **using** (ANr)
  - **select * from** Mitarbeiter **natural join** Abteilungen

  Nach diesem Konstrukt können mit einer WHERE-Klausel weitere Bedingungen an das Ergebnis gestellt werden.

Beispiel (Wdh. S. 12)

select * **from** Mitarbeiter m, Abteilungen a...

PNr	Name	Vorname	m.ANr	a.ANr	Abteilungsname
001	Huber	Erwin	01	01	Buchhaltung
001	Huber	Erwin	01	02	Produktion
002	Mayer	Hugo	01	01	Buchhaltung
002	Mayer	Hugo	01	02	Produktion
003	Müller	Anton	02	01	Buchhaltung
003	Müller	Anton	02	02	Produktion

...**where** m.ANr = a.ANr

PNr	Name	Vorname	m.ANr	a.ANr	Abteilungsname
001	Huber	Erwin	01	01	Buchhaltung
002	Mayer	Hugo	01	01	Buchhaltung
003	Müller	Anton	02	02	Produktion

Beispiele:

- Gegeben sei folgendes Datenbankschema:
  - Kunde (KName, KAdr, Kto)
  - Auftrag (KName, Ware, Menge)
  - Lieferant (LName, LAdr, Ware, Preis)
- Welche Lieferanten liefern Mehl oder Milch?

```
select distinct LName
from Lieferant
where Ware = 'Mehl' or Ware = 'Milch'
```

- Welche Lieferanten liefern irgendetwas, das der Kunde Huber bestellt hat?

```
select distinct LName
from Lieferant l, Auftrag a
where l.Ware = a.Ware and KName = 'Huber'
```

Beispiele (Self-Join):

Kunde (KName, KAdr, Kto)
Auftrag (KName, Ware, Menge)
Lieferant (LName, LAdr, Ware, Preis)

- Name und Adressen aller Kunden, deren Kontostand kleiner als der von Huber ist

```
select k1.KName, k1.Adr
from Kunde k1, Kunde k2
where k1.Kto < k2.Kto and k2.KName = 'Huber'
```

- Finde alle Paare von Lieferanten, die eine gleiche Ware liefern

```
select distinct L1.Lname, L2.LName
from Lieferant L1, Lieferant L2
where L1.Ware=L2.Ware and L1.LName<L2.LName
```

?

Beispiele (Self-Join)

Lieferant*

Müller	Mehl
Müller	Haferfl
Bäcker	Mehl

Ohne Zusatzbedingung:

Müller	Mehl	Müller	Mehl
Müller	Mehl	Bäcker	Mehl
Müller	Haferfl	Müller	Haferfl
Bäcker	Mehl	Müller	Mehl
Bäcker	Mehl	Bäcker	Mehl

Nach Projektion:

~~Müller~~	~~Müller~~
~~Müller~~	~~Bäcker~~
Bäcker	Müller
~~Bäcker~~	~~Bäcker~~

L1.LName > L2.LName

L1.LName = L2.LName

UNION, INTERSECT, EXCEPT

- Üblicherweise werden mit diesen Operationen die Ergebnisse zweier SELECT-FROM-WHERE-Blöcke verknüpft:

  select * **from** Mitarbeiter **where** name **like** ‘A%‘
  union -- Vereinigung **mit** Duplikatelimination
  select * **from** Studenten **where** name **like** ‘A%‘

- Bei neueren Datenbanksystemen ist auch möglich:

  select * **from** Mitarbeiter **union** Studenten **where** ...

- Genauso bei:
  - Durchschnitt: INTERSECT
  - Differenz: EXCEPT
  - Vereinigung **ohne** Duplikatelimination: UNION ALL

UNION, INTERSECT, EXCEPT

- Die relationale Algebra verlangt, daß die beiden Relationen, die verknüpft werden, das gleiche Schema besitzen (Namen und Wertebereiche)
- SQL verlangt nur kompatible Wertebereiche, d.h.:
  - beide Wertebereich sind **character** (Länge usw. egal)
  - beide Wertebereiche sind Zahlen (Genauigkeit egal)
  - oder beide Wertebereiche sind gleich

UNION, INTERSECT, EXCEPT

- Mit dem Schlüsselwort **corresponding** beschränken sich die Operationen automatisch auf die gleich benannten Attribute
- Beispiel (aus *Datenbanken kompakt*):

R_1:

A	B	C
1	2	3
2	3	4

R_2:

A	C	D
2	2	3
5	3	2

R_1 **union** R_2:

A	B	C
1	2	3
2	3	4
2	2	3
5	3	2

R_1 **union corresponding** R_2:

A	C
1	3
2	4
2	2
5	3

Änderungs-Operationen

- Bisher: Nur *Anfragen* an das Datenbanksystem
- Änderungsoperationen modifizieren den Inhalt eines oder mehrerer Tupel einer Relation
- Grundsätzlich unterscheiden wir:
  - INSERT: Einfügen von Tupeln in eine Relation
  - DELETE: Löschen von Tupeln aus einer Relation
  - UPDATE: Ändern von Tupeln einer Relation
- Diese Operationen sind verfügbar als...
  - Ein-Tupel-Operationen
    z.B. die Erfassung eines neuen Mitarbeiters
  - Mehr-Tupel-Operationen
    z.B. die Erhöhung aller Gehälter um 2.1%

Die UPDATE-Anweisung

- Syntax:

  update *relation*
  set $attribut_1$ = $ausdruck_1$
  [, … ,
  $attribut_n$ = $ausdruck_n$]*
  [**where** *bedingung*]

- Wirkung:
  In allen Tupeln der Relation, die die Bedingung erfüllen (falls angegeben, sonst in allen Tupeln), werden die Attributwerte wie angegeben gesetzt

*falsch in *Heuer&Saake*: Zuweisungen müssen durch Kommata getrennt werden

Die UPDATE-Anweisung

- UPDATE ist i.a. eine Mehrtupel-Operation
- Beispiel:
  update Angestellte
  set Gehalt = 6000

- Wie kann man sich auf ein einzelnes Tupel beschränken?
  Spezifikation des Schlüssels in WHERE-Bedg.

- Beispiel:
  update Angestellte
  set Gehalt = 6000
  where PNr = 7

Die UPDATE-Anweisung

- Der alte Attribut-Wert kann bei der Berechnung des neuen Attributwertes herangezogen werden
- Beispiel:
  Erhöhe das Gehalt aller Angestellten, die weniger als 3000,-- € verdienen, um 2%

```
update  Angestellte
set     Gehalt = Gehalt * 1.02
where   Gehalt < 3000
```

- UPDATE-Operationen können zur Verletzung von Integritätsbedingungen führen:
  Abbruch der Operation mit Fehlermeldung.

Die DELETE-Anweisung

- Syntax:
  delete from *relation*
  [**where** *bedingung*]
- Wirkung:
  - Löscht alle Tupel, die die Bedingung erfüllen
  - Ist keine Bedingung angegeben, werden *alle* Tupel gelöscht
  - Abbruch der Operation, falls eine Integritätsbedingung verletzt würde (z.B. Fremdschlüssel ohne *cascade*)
- Beispiel: Löschen aller Angestellten mit Gehalt 0

  delete from Angestellte
  where Gehalt = 0

Die INSERT-Anweisung

- Zwei unterschiedliche Formen:
  - Einfügen konstanter Tupel (Ein-Tupel-Operation)
  - Einfügen berechneter Tupel (Mehr-Tupel-Operation)

- Syntax zum Einfügen konstanter Tupel:
  insert into *relation* (*attribut*$_1$, *attribut*$_2$,...)
  values (*konstante*$_1$, *konstante*$_2$, ...)

- oder:
  insert into *relation*
  values (*konstante*$_1$, *konstante*$_2$, ...)

Einfügen konstanter Tupel

- Wirkung:
  Ist die optionale Attributliste hinter dem Relationennamen angegeben, dann...
  - können unvollständige Tupel eingefügt werden:
    Nicht aufgeführte Attribute werden mit NULL belegt
  - werden die Werte durch die Reihenfolge in der Attributsliste zugeordnet
- Beispiel:
  insert into Angestellte (Vorame, Name, PNr)
  values ('Donald', 'Duck', 678)

PNr	Name	Vorname	ANr
678	Duck	Donald	NULL

Einfügen konstanter Tupel

- Wirkung:
  Ist die Attributliste *nicht* angegeben, dann...
  - können unvollständige Tupel nur durch explizite Angabe von NULL eingegeben werden
  - werden die Werte durch die Reihenfolge in der DDL-Definition der Relation zugeordnet (mangelnde Datenunabhängigkeit!)
- Beispiel:
  insert into Angestellte
  values (678, 'Duck', 'Donald', NULL)

PNr	Name	Vorname	ANr
678	Duck	Donald	NULL

Einfügen berechneter Tupel

- Syntax zum Einfügen berechneter Tupel:
  insert into *relation* [**(***attribut*$_1$ **, ...)**]
  (select ... **from** ... **where** ... **)**
- Wirkung:
  - Alle Tupel des Ergebnisses der SELECT-Anweisung werden in die Relation eingefügt
  - Die optionale Attributliste hat dieselbe Bedeutung wie bei der entsprechenden Ein-Tupel-Operation
  - Bei Verletzung von Integritätsbedingungen (z.B. Fremdschlüssel nicht vorhanden) wird die Operation nicht ausgeführt (Fehlermeldung)

Einfügen berechneter Tupel

- Beispiel:
  Füge alle Lieferanten in die Kunden-Relation ein (mit Kontostand 0)

- Datenbankschema:
  - Kunde (KName, KAdr, Kto)
  - Lieferant (LName, LAdr, Ware, Preis)

insert into Kunde
(select distinct LName, LAdr, 0 **from** Lieferant**)**

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 4: Relationen-Kalkül

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Begriff

Kal|kül *das, auch der; -s, -e* <unter Einfluss von gleichbed. *fr.* calcul aus *lat.* calculus «Steinchen, Rechen-, Spielstein; Berechnung», Verkleinerungsform von *lat.* calx «(Spiel)stein; Kalk»>: etwas im Voraus abschätzende, einschätzende Berechnung, Überlegung.

Quelle: DUDEN - Das große Fremdwörterbuch

...das Kalkül

der Kalkül ...

Kal|kül *der; -s, -e* <zu [1]Kalkül>: durch ein System von Regeln festgelegte Methode, mit deren Hilfe bestimmte mathematische Probleme systematisch behandelt u. automatisch gelöst werden können (Math.).

Quelle: DUDEN - Das große Fremdwörterbuch

Begriff

- Mathematik: Prädikatenkalkül
  - Formeln wie $\{x \mid x \in I\!N \wedge x^3 > 0 \wedge x^3 < 1000\}$
- Anwendung solcher Formeln für DB-Anfragen
  - Bezugnahme auf DB-Relationen im Bedingungsteil: $(x_1, y_1, z_1) \in$ Mitarbeiter, $t_1 \in$ Abteilungen
  - Terme werden gebildet aus Variablen, Konstanten usw.
  - Atomare Formeln aus Prädikaten der Datentypen: $=, <, >, \leq$, usw.
  - Atomare Formeln können mit logischen Operatoren zu komplexen Formeln zusammengefasst werden: $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $\exists x\colon F_1$, $\forall x\colon F_1$
- Bsp: Finde alle Großstädte in Bayern:
  $\{t \mid \text{Städte(t)} \wedge t[\text{Land}] = \text{Bayern} \wedge t[\text{SEinw}] \geq 500.000\}$

  Hinweis: Städte(t) gleichbedeutend mit t $\in$ Städte

Unterschied zur Rel. Algebra

- Relationale Algebra ist prozedurale Sprache:
  - Ausdruck gibt an, unter Benutzung welcher Operationen das Ergebnis berechnet werden soll
  - WIE
- Relationen-Kalkül ist deklarative Sprache:
  - Ausdruck beschreibt, welche Eigenschaften die Tupel der Ergebnisrelation haben müssen ohne eine Berechnungsprozedur dafür anzugeben
  - WAS
- Es gibt zwei verschiedene Ansätze:
  - Tupelkalkül: Variablen sind vom Typ *Tupel*
  - Bereichskalkül: Variablen haben *einfachen* Typ

Der Tupelkalkül

- Man arbeitet mit
  - Tupelvariablen: t
  - Formeln: $\psi(t)$
  - Ausdrücken: $\{t \mid \psi(t)\}$
- Idee: Ein Ausdruck beschreibt die Menge aller Tupel, die die Formel ψ erfüllen (wahr machen)
- Ein Kalkül besteht immer aus
  - Syntax: Wie sind Ausdrücke aufgebaut?
  - Semantik: Was bedeuten die Ausdrücke?

Tupelvariablen

- Tupelvariablen haben ein definiertes Schema:
  - Schema(t) = (A_1: D_1, A_2: D_2, ...)
  - Schema(t) = R_1 (t hat dasselbe Schema wie Relation)
- Für Zugriff auf die Komponenten
  - $t[A]$ oder $t.A$ für einen Attributnamen $A \in$ Schema(t)
  - oder auch $t[1]$, $t[2]$ usw.
- Tupelvariable kann in einer Formel ψ frei oder gebunden auftreten (s. unten)

Atome

- Es gibt drei Arten von Atomen:
  - $R(t)$ — R ist Relationenname, t Tupelvariable
    lies: t ist ein Tupel von R
  - $t.A\ \Theta\ s.B$ — t bzw. s sind zwei Tupelvariablen mit passenden Attributen
    lies: $t.A$ steht in Beziehung Θ zu ...
  - $t.A\ \Theta\ c$ — t ist Tupelvariable und c eine passende Konstante

---

Θ Vergleichsoperator: $\Theta \in \{ =, <, \leq, >, \geq, \neq \}$

Formeln

Der Aufbau von Formeln ψ ist rekursiv definiert:

- **Atome**: Jedes Atom ist eine Formel
  Alle vorkommenden Variablen sind frei
- **Verknüpfungen**: Sind ψ_1 und ψ_2 Formeln, dann auch:
  - $\neg \psi_1$ *nicht*
  - $(\psi_1 \wedge \psi_2)$ *und*
  - $(\psi_1 \vee \psi_2)$ *oder*

  Alle Variablen behalten ihren Status.
- **Quantoren**: Ist ψ eine Formel, in der t als freie Variable auftritt, sind auch Formeln...
  - $(\exists t)(\psi)$ es gibt ein t, für das ψ
  - $(\forall t)(\psi)$ für alle t gilt ψ

  die Variable t wird gebunden.

Formeln

- Gebräuchliche vereinfachende Schreibweisen:
  - $\psi_1 \Rightarrow \psi_2$ für $(\neg \psi_1) \vee \psi_2$ (Implikation)
  - $\exists t_1,...,t_k: \psi(t_1,...t_k)$ für $(\exists t_1)\,(...((\exists t_k)\,(\psi(t_1,...t_k)))...)$
  - $(\exists t \in R)\,(\psi(t))$ für $(\exists t)\,(R(t) \wedge \psi(t))$
  - $(\forall t \in R)\,(\psi(t))$ für $(\forall t)\,(R(t) \Rightarrow \psi(t))$
  - Bei Eindeutigkeit können Klammern weggelassen werden
- Beispiel:
  - $(\forall s)\,(s.\mathrm{A} \leq u.\mathrm{B} \vee (\exists u)(\mathrm{R}(u) \wedge u.\mathrm{C} > t.\mathrm{D}))$
  - t ist frei
  - s ist gebunden
  - u ist frei beim ersten Auftreten und dann gebunden

Ausdruck (Anfrage)

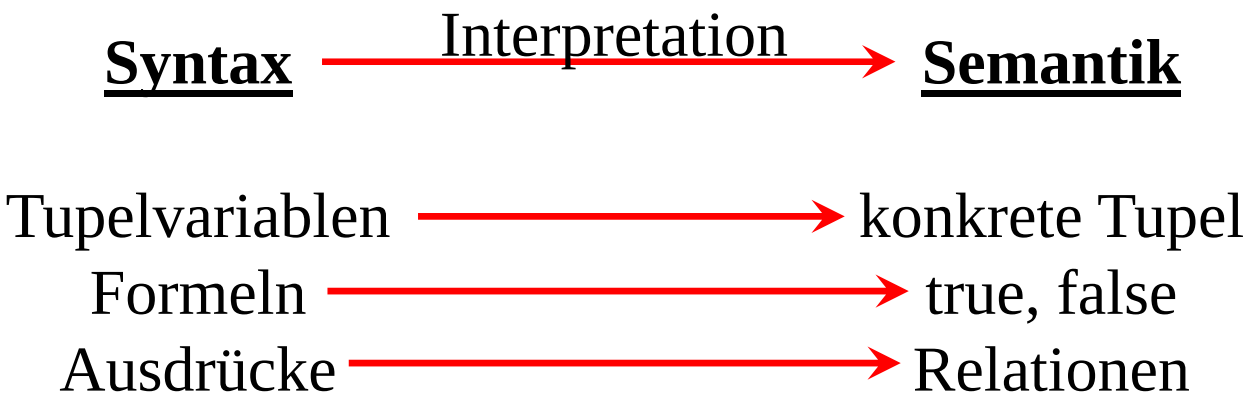
- Ein Ausdruck des Tupelkalküls hat die Form

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

- In Formel ψ ist t die einzige freie Variable

Semantik

Bedeutung, die einem korrekt gebildeten Ausdruck durch eine Interpretation zugeordnet wird:



Belegung von Variablen

- Gegeben:
  - eine Tupelvariable t mit Schema(t) = (D_1, D_2, ...)
  - eine Formel $\psi(t)$, in der t frei vorkommt
  - ein beliebiges konkretes Tupel r (d.h. mit Werten). Es muß nicht zu einer Relation der Datenbank gehören
- Bei der Belegung wird jedes freie Vorkommen von t durch r ersetzt. Insbesondere wird t.A durch den Attributwert von r.A ersetzt.
- Man schreibt: $\psi(r \mid t)$

Beispiel

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- $\psi(t) = (t.\text{Land}=\text{Bayern} \wedge t.\text{SEinw} \geq 500.000)$
  mit Schema(t) = Schema(Städte)

  - r_1 = (Passau, 49800, Bayern):
    $\psi(r_1 \mid t) = (\text{Bayern} = \text{Bayern} \wedge 49800 \geq 500.000)$
  - r_2 = (Bremen, 535.058, Bayern):
    $\psi(r_2 \mid t) = (\text{Bremen} = \text{Bayern} \wedge 535.058 \geq 500.000)$

Interpretation von Formeln

Interpretation $I(\psi)$ analog zu syntaktischem Aufbau

- Anm: Alle Variablen sind durch konkrete Tupel belegt

- Atome:
  - $R(r)$: $I(R(r))$ = **true** $\Leftrightarrow$ r ist in R enthalten
  - $c_i \Theta c_j$: $I(c_i \Theta c_j)$ = **true** $\Leftrightarrow$ der Vergleich ist erfüllt
- Logische Operatoren:
  - $\neg\psi$: $I(\neg\psi)$ = **true** $\Leftrightarrow$ $I(\psi)$ = **false**
  - $\psi_1 \wedge \psi_2$: $I(\psi_1 \wedge \psi_2)$=**true** $\Leftrightarrow$ $I(\psi_1)$=**true** und $I(\psi_2)$=**true**
  - $\psi_1 \vee \psi_2$: $I(\psi_1 \vee \psi_2)$=**true** $\Leftrightarrow$ $I(\psi_1)$=**true** oder $I(\psi_2)$=**true**

Beispiele

- Atome:
  - I(Städte (Passau, 49.800, Bayern)) = **true**
  - I(49.800 ≥ 500.000) = **false**

- Logische Operatoren:
  - $I(\neg$ 49.800 ≥ 500.000) = **true**
  - I(Städte (Passau, 49.800, Bayern) $\vee$ $\vee$ 49.800 ≥ 500.000) = **true**
  - I(Städte (Passau, 49.800, Bayern) $\wedge$ $\wedge$ 49.800 ≥ 500.000) = **false**

Interpretation von Quantoren

- Interpretation $I((\exists s)(\psi))$ bzw. $I((\forall s)(\psi))$:
  - In ψ darf nur s als freie Variable auftreten.
  - $I((\exists s)(\psi))$ = **true** $\Leftrightarrow$ ein Tupel $r \in D_1 \times D_2 \times \ldots$ existiert, daß bei Belegung der Variablen s die Formel ψ gilt:

    $$I(\psi(r \mid s)\ ) = \textbf{true}$$

  - $I((\forall s)(\psi))$ = **true** $\Leftrightarrow$ für alle Tupel $r \in D_1 \times D_2 \times \ldots$ gilt die Formel ψ.
- Beispiele:
  - $I((\exists s)(\text{Städte}(s) \wedge s.\text{Land} = \text{Bayern}))$ = **true**
  - $I((\forall s)(s.\text{Name} = \text{Passau}))$ = **false**

Interpretation von Ausdrücken

- Interpretation von Ausdruck $I(\{t|\psi(t)\})$ stützt sich
  - auf Belegung von Variablen
  - und Interpretation von Formeln
- Gegeben:
  - $E = \{t \mid \psi(t)\}$
  - t die einzige freie Variable in $\psi(t)$
  - Schema(t) = $D_1 \times D_2 \times \ldots$
- Dann ist der Wert von E die Menge aller* (denkbaren) Tupel $r \in D_1 \times D_2 \times \ldots$ für die gilt:

$$I(\psi(r \mid t)) = \textbf{true}$$

*Grundmenge sind hier nicht nur die gespeicherten Tupel aus der DB

Beispiel-Anfragen

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- Finde alle Großstädte (SName, SEinw, Land) in Bayern:

  Schema(t) = Schema(Städte)

  $\{t \mid \text{Städte}(t) \wedge t.\text{Land} = \text{Bayern} \wedge t.\text{SEinw} \geq 500.000\}$
- In welchem Land liegt Passau?

  Schema(t) = (Land:String)

  $\{t \mid (\exists u \in \text{Städte})(u.\text{Sname} = \text{Passau} \wedge u.\text{Land} = t.\text{Land}\}$
- Finde alle Städte in CDU-regierten Ländern:

  Schema(t) = Schema(Städte)

  $\{t \mid \text{Städte}(t) \wedge (\exists u \in \text{Länder})(u.\text{Lname}=t.\text{Land} \wedge u.\text{Partei}=\text{CDU})\}$

Beispiel-Anfragen

Gegeben sei folgendes Relationenschema:

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

* bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

---

- Welche Länder werden von der SPD allein regiert?

  Schema(t) = Schema(Länder)

  $\{t|\text{Länder}(t) \wedge (\forall u \in \text{Länder})(u.\text{LName}=t.\text{LName} \Rightarrow u.\text{Partei}=\text{SPD}\}$

- Gleichbedeutend mit:

  Schema(t) = Schema(Länder)

  $\{t|\text{Länder}(t) \wedge (\forall u \in \text{Länder}) \neg (u.\text{LName}=t.\text{LName} \wedge u.\text{Partei} \neq \text{SPD}\}$

Sichere Ausdrücke

- Mit den bisherigen Definitionen ist es möglich, unendliche Relationen zu beschreiben:
  - Schema(t) = {String, String}
  - $\{t \mid t.1 = t.2 \}$
  - Ergebnis: {(A,A),(B,B),...,(AA,AA),(AB,AB),...}
- Probleme:
  - Ergebnis kann nicht gespeichert werden
  - Ergebnis kann nicht in endlicher Zeit berechnet werden
- Definition:
  Ein Ausdruck heißt *sicher*, wenn jede Tupelvariable nur Werte einer gespeicherten Relation annehmen kann, also positiv in einem Atom R(t) vorkommt.

Der Bereichskalkül

- Tupelkalkül: Tupelvariablen t (ganze Tupel)
- Bereichskalkül: Bereichsvariablen $x_1{:}D_1$, $x_2{:}D_2$, ... für einzelne Attribute
  (Bereich=Wertebereich=Domäne)

Ein **Ausdruck** hat die Form:

$$\{x_1, x_2, \ldots \mid \psi\,(x_1, x_2, \ldots)\}$$

Atome haben die Form:

- $R_1\,(x_1, x_2, \ldots)$: Tupel $(x_1, x_2, \ldots)$ tritt in Relation R_1 auf
- $x\ \Theta\ y$: x,y Bereichsvariablen bzw. Konstanten
  $\Theta \in \{ =, <, \leq, >, \geq, \neq \}$

Formeln analog zum Tupelkalkül

Beispiel-Anfragen

Städte (SName: String, SEinw: Integer, Land: String)

Länder (LName: String, LEinw: Integer, Partei*: String)

*bei Koalitionsregierungen: jeweils eigenes Tupel pro Partei

- In welchem Land liegt Passau?

  $\{x_3 \mid \exists\, x_1, x_2: (\text{Städte}(x_1, x_2, x_3) \wedge x_1 = \text{Passau}) \}$

  oder auch

  $\{x_3 \mid \exists\, x_2: (\text{Städte}(\text{Passau}, x_2, x_3)) \}$
- Finde alle Städte in CDU-regierten Ländern:

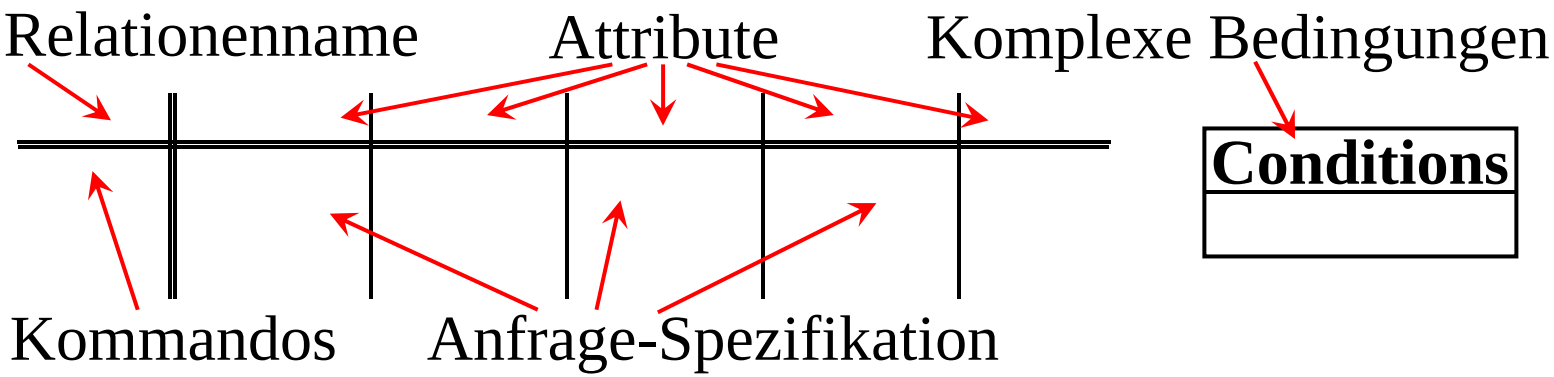
  $\{x_1 \mid \exists\, x_2 , x_3 , y_2 : (\text{Städte}(x_1,x_2,x_3) \wedge \text{Länder}\ (x_1,y_2,\text{CDU}))\}$
- Welche Länder werden von der SPD allein regiert?

  $\{x_1 \mid \exists x_2 : (\text{Länder}(x_1,x_2,\text{SPD}) \wedge \neg \exists y_3 : (\text{Länder}(x_1,x_2,y_3) \wedge y_3 \neq \text{SPD}))\}$

Query By Example (QBE)

- Beruht auf dem Bereichskalkül
- Ausdrücke nicht wie in SQL als Text
- Dem Benutzer wird am Bildschirm ein Tabellen-Gerüst angeboten, das mit Spezial-Editor bearbeitet werden kann
- Nach Eintrag von Werten in das Tabellengerüst (Anfrage) füllt das System die Tabelle
- Zielgruppe: Gelegentliche Benutzer

Query By Example (QBE)



Sprachelemente:

- Kommandos, z.B. **P.** (print), **I.** (insert), **D.** (delete) ...
- Bereichsvariablen (beginnen mit '_'): _x, _y
- Konstanten (Huber, Milch)
- Vergleichsoperatoren und arithmetische Operatoren
- Condition-Box: Zusätzlicher Kasten zum Eintragen einer Liste von Bedingungen (**AND**, **OR**, kein **NOT**)

Beispiel-Dialog

- Beginn: leeres Tabellengerüst

- *Benutzer* gibt interessierende Relation und **P.** ein

Kunde P.			

- *System* trägt Attributsnamen der Relation ein

Kunde	KName	KAdr	Kto

- *Benutzer* stellt Anfrage

Kunde	KName	KAdr	Kto
	P.	**P.**	**<0**

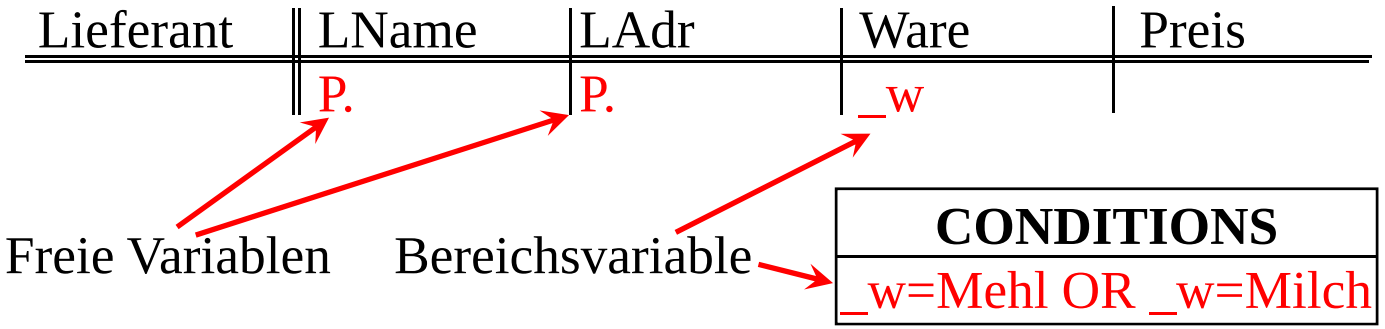
- *System* füllt Tabelle mit Ergebnis-Werten

Kunde	KName	KAdr
	Huber	Innsbruck
	Maier	München

evtl. weitere Tabelle (Join)

Anfragen mit Bedingungen

- Welche Lieferanten liefern Mehl oder Milch?



- Bedeutung:

  $\{x_1, x_2 \mid \exists w, x_4 : \text{Lieferant}(x_1, x_2, w, x_4) \wedge (w = \text{Mehl} \vee w = \text{Milch})\}$

- Kommando **P.** für print bzw. auch für die Projektion

Anfragen mit Bedingungen

- Welche Lieferanten liefern Brie und Perrier, wobei Gesamtpreis 7,00 € nicht übersteigt?

Lieferant	LName	LAdr	Ware	Preis
	P. _L		Brie	_y
	_L		Perrier	_z

CONDITIONS
_y + _z <= 7.00

- Bedeutung:

$\{l \mid \exists\, x_1, x_2, y, z$: Lieferant $(l, x_1,$ Brie, $y) \wedge$
Lieferant $(l, x_2,$ Perrier, $z) \wedge y + z \leq 7.00\}$

Join-Anfragen

- Welcher Lieferant liefert etwas das Huber bestellt hat?

Lieferant	LName	LAdr	Ware	Preis
	P.		_w	

Auftrag	KName	Ware	Menge
	Huber	_w	

- Bedeutung:

  $\{x_1 \mid \exists\, x_2, w, x_4, y_3$: Lieferant $(x_1, x_2, w, x_4) \wedge$ Auftrag (Huber, w, $y_3)\}$

- Beachte:
  Automatische Duplikat-Elimination in QBE

Join-Anfragen

Meist ist für Ergebnis neues Tabellengerüst nötig:

- Beispiel: Bestellungen mit Kontostand des Kunden
- Falsch (leider nicht möglich):

Kunde	KName	KAdr	Kto
	P. _n		P.

Auftrag	KName	Ware	Menge
	_n	P.	P.

- Richtig:

Kunde	KName	KAdr	Kto
	_n		_k

Auftrag	KName	Ware	Menge
	_n	_w	_m

Bestellung	Name	Was	Wieviel	Kontostand
P.	P. _n	P. _w	P. _m	P. _k

Abkürzung!

Anfragen mit Ungleichung

- Wer liefert Milch zu Preis zw. 0,50 € und 0,60 € ?
- Variante mit zwei Zeilen:

Lieferant	LName	LAdr	Ware	Preis
P.	_L		Milch	>= 0.5
	_L		Milch	<= 0.6

- Variante mit Condition-Box

Lieferant	LName	LAdr	Ware	Preis
P.			Milch	_p

CONDITIONS
_p >= 0.5 AND _p <= 0.6

Anfragen mit Negation

- Finde für jede Ware den billigsten Lieferanten

Lieferant	LName	LAdr	Ware	Preis
P.			_w	_p
¬			_w	< _p

- Das Symbol ¬ in der ersten Spalte bedeutet:
  Es gibt kein solches Tupel

- Bedeutung:

$\{x_1, x_2, w, p \mid \neg\exists\, y_1, y_2, y_3$: Lieferant $(x_1, x_2, w, p) \wedge$
Lieferant $(y_1, y_2, w, y_3) \wedge y_3 < p\}$

Einfügen

- Einfügen von einzelnen Tupeln
  - Kommando **I.** für INSERT

Kunde	KName	KAdr	Kto
I.	Schulz	Wien	0

- Einfügen von Tupeln aus einem Anfrageergebnis
  - Beispiel: Alle Lieferanten in Kundentabelle übernehmen

Kunde	KName	KAdr	Kto
I.	_n	_a	0

Lieferant	LName	LAdr	Ware	Preis
	_n	_a		

Löschen und Ändern

- Löschen aller Kunden mit negativem Kontostand

Kunde	KName	KAdr	Kto
D.			< 0

- Ändern eines Tupels (**U.** für UPDATE)

Kunde	KName	KAdr	Kto
	Schulz	Wien	U. 100

- oder auch:

Kunde	KName	KAdr	Kto
	Meier	_a	_k
U.	Meier	_a	_k – 110

- oder auch mit Condition-Box

Vergleich

QBE	Bereichskalkül
Konstanten	Konstanten
Bereichsvariablen	Bereichsvariablen
leere Spalten	paarweise verschiedene Bereichsvariablen, ∃-quantifiziert
Spalten mit **P.**	freie Variablen
Spalten ohne **P.**	∃-quantifizierte Variablen

Anmerkung: QBE ist relational vollständig, jedoch ist für manche Anfragen der relationalen Algebra eine Folge von QBE-Anfragen nötig

Umsetzung einer QBE-Anfrage

(ohne Negation)

- Erzeuge für alle Attribute A_i aller vorkommenden Tabellen-Zeilen der Anfrage eine Bereichsvariable x_i
- Steht bei Attribut A_i das Kommando **P.**
  dann schreibe x_i zu den freien Variablen ({... x_i, ... | ...}),
  sonst binde x_i mit einem ∃-Quantor ({... | ∃..., x_i, ...})
- Binde alle Variablen der Anfrage mit einem ∃-Quantor

Lieferant	LName	LAdr	Ware	Preis
	P.	P.	P. _w	

Auftrag	KName	Ware	Menge
	Huber	_w	>= 5

$\{x_1, x_2, x_3 \mid \exists\ x_4, x_5, x_6, x_7, w{:}\ldots$

Umsetzung einer QBE-Anfrage

Lieferant	LName	LAdr	Ware	Preis
	P.	P.	P. _w	

Auftrag	KName	Ware	Menge
	Huber	_w	>= 5

- Füge für jede vorkommende Relation R ein Atom der Form $R(x_i, x_{i+1}, ...)$ mit $\wedge$ an die Formel Ψ an

$$\{x_1,x_2,x_3 \mid \exists\, x_4,x_5,x_6,x_7,w\text{: Lieferant}(x_1,x_2,x_3,x_4) \wedge \text{Auftrag}(x_5,x_6,x_7) \ ...$$

- Steht bei A_i ein Zusatz der Form **Const** bzw. **≤ Const** etc., dann hänge x_i = Const bzw. $x_i \leq$ Const mit $\wedge$ an Formel.

$$\{x_1,x_2,x_3 \mid \exists\, x_4,x_5,x_6,x_7,w\text{: Lieferant}(x_1,x_2,x_3,x_4) \wedge \text{Auftrag}(x_5,x_6,x_7) \wedge x_5 = \text{Huber} \wedge x_7 \geq 5$$

Umsetzung einer QBE-Anfrage

Lieferant	LName	LAdr	Ware	Preis
	P.	P.	P. _w	

Auftrag	KName	Ware	Menge
	Huber	_w	>= 5

- Gleiches Vorgehen bei Zusätzen der Form _Variable bzw. ≤ _Variable usw:

$$\{x_1,x_2,x_3 \mid \exists\, x_4,x_5,x_6,x_7,w\colon \text{Lieferant}(x_1,x_2,x_3,x_4) \wedge \text{Auftrag}(x_5,x_6,x_7) \wedge x_5=\text{Huber} \wedge x_7 \geq 5 \wedge w = x_3 \wedge w = x_6\}$$

- Ggf. wird der Inhalt der Condition-Box mit ∧ angehängt.
- Meist lässt sich der Term noch vereinfachen:

$$\{x_1,x_2,w \mid \exists\, x_4,x_5,x_7\colon \text{Lieferant}(x_1,x_2,w,x_4) \wedge \text{Auftrag}(\text{Huber},w,x_7) \wedge x_7 \geq 5\}$$

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 5: Mehr zu SQL

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Überblick

- Bisher (Kapitel 3):
  Operatoren der relationalen Algebra in SQL:
  - SELECT *Attribute*
    FROM *Relationen*
    WHERE *Bedingung*
  - sowie die Mengenoperationen (UNION, EXCEPT, ...)
- In diesem Kapitel:
  Erweiterungen, die effektives Arbeiten ermöglichen
  - Verschiedene Formen von Quantoren (rel. Kalkül)
  - Aggregationen
  - Sortieren und Gruppieren von Tupeln
  - Sichten

Outer Join

- Problem:
  Beim gewöhnlichen („inner“) Join gehen diejenigen Tupel verloren, die keine Joinpartner in der jeweiligen anderen Relation haben
- Beispiel:
  Auflistung aller Kunden mit ihren aktuellen Bestellungen:
  select * from kunde k, auftrag a where k.kname = a.kname
  Kunden ohne aktuellen Auftrag erscheinen nicht.

Kunde:

KName	KAdr	Kto
Maier	Hall	10
Huber	Ibk	25
Geizhals	Ibk	0

Auftrag:

KName	Ware	...
Maier	Brot	
Maier	Milch	
Huber	Mehl	

Kunde ⋈ Auftrag:

KName	KAdr	Kto	Ware	...
Maier	Hall	10	Brot	
Maier	Hall	10	Milch	
Huber	Ibk	25	Mehl	

Geizhals erscheint nicht mehr in der erweiterten Liste

Outer Join

- Ein Outer Join ergänzt das Joinergebnis um die Tupel, die keinen Joinpartner in der anderen Relation haben.
- Das Ergebnis wird mit NULL-Werten aufgefüllt:
  select * from kunde natural outer join auftrag

Kunde:

KName	KAdr	Kto
Maier	Hall	10
Huber	Ibk	25
Geizhals	Ibk	0

Auftrag:

KName	Ware	...
Maier	Brot	
Maier	Milch	
Huber	Mehl	

Kunde n.o.j. Auftrag:

KName	KAdr	Kto	Ware	...
Maier	Hall	10	Brot	
Maier	Hall	10	Milch	
Huber	Ibk	25	Mehl	
Geizhals	Ibk	0	NULL	

Outer Join

- Aufstellung aller Möglichkeiten:
  - [inner] join: keine verlustfreie Relation, Normalfall
  - left outer join: die linke Relation ist verlustfrei
  - right outer join: die rechte Relation ist verlustfrei
  - [full] outer join: beide Relationen verlustfrei
- Kombinierbar mit Schlüsselworten natural, on, ..

L

A	B
1	2
2	3

R

B	C
3	4
4	5

inner

A	B	C
2	3	4

left

A	B	C
1	2	⊥
2	3	4

right

A	B	C
2	3	4
⊥	4	5

full

A	B	C
1	2	⊥
2	3	4
⊥	4	5

Quantoren und Subqueries

- Quantoren sind Konzept des Relationenkalküls
- In relationaler Algebra nicht vorhanden
- Können zwar simuliert werden:
  - Existenzquantor implizit durch Join und Projektion:
    $\{x \in R \mid \exists y \in S\colon \ldots\} \quad \equiv \quad \pi_{R.*}\,(\sigma\ldots\,(R \times S)\,)$
  - Allquantor mit Hilfe des Quotienten
    $\{x \in R \mid \forall y \in S\colon \ldots\} \quad \equiv \quad (\sigma\ldots\,(R)) \div S$
- Häufig Formulierung mit Quantoren natürlicher
- SQL: Quantifizierter Ausdruck in einer Subquery

Quantoren und Subqueries

- Beispiel für eine Subquery
  select * from Kunde **where exists** (**select**...**from**...**where**...)

  Subquery
- In Where-Klausel der Subquery auch Zugriff auf Relationen/Attribute der Hauptquery
- Eindeutigkeit ggf. durch Aliasnamen für Relationen (wie bei Self-Join):

```
select *
from kunde k1
where exists     ( select *
                   from Kunde k2
                   where k1.Adr=k2.Adr and...
                 )
```

Existenz-Quantor

- Realisiert mit dem Schlüsselwort exists
- Der ∃-quantifizierte Ausdruck wird in einer Subquery notiert.
- Term **true** gdw. Ergebnis der Subquery **nicht leer**
- Beispiel:
  KAdr der Kunden, zu denen ein Auftrag existiert:

```
select KAdr from Kunde k
where exists
      ( select * from Auftrag a
        where a.KName = k.KName
      )
```

Äquivalent mit Join ??

Allquantor

- Keine direkte Unterstützung in SQL
- Aber leicht ausdrückbar durch die Äquivalenz:

$$\forall x\colon \psi(x) \Leftrightarrow \neg\exists x\colon \neg\psi(x)$$

- Also Notation in SQL:
  ...where not exists (select...from...where not...)
- Beispiel:
  Die Länder, die von der SPD allein regiert werden

```
select * from Länder L1
where not exists
  ( select * from Länder L2
    where L1.LName=L2.LName and not L1.Partei='SPD'
  )
```

Direkte Subquery

- An jeder Stelle in der **select**- und **where**-Klausel, an der ein konstanter Wert stehen kann, kann auch eine Subquery **(select...from...where...)** stehen.
- Einschränkungen:
  - Subquery darf nur ein Attribut ermitteln (Projektion)
  - Subquery darf nur ein Tupel ermitteln (Selektion)
- Beispiel: Dollarkurs aus Kurstabelle

```
select    Preis,
          Preis  *    ( select Kurs from Devisen
                        where DName = 'US$'     ) as USPreis
from Waren where ...
```

- Oft schwierig, Eindeutigkeit zu gewährleisten...

Weitere Quantoren

- Quantoren bei Standard-Vergleichen in WHERE
- Formen:
  - $A_i\ \Theta$ all (select...from...where...) — $\forall$-Quantor
  - $A_i\ \Theta$ some (select...from...where...) — $\exists$-Quantor
  - $A_i\ \Theta$ any (select...from...where...) — $\exists$-Quantor

  Vergleichsoperatoren $\Theta \in \{ =, <, <=, >, >=, <> \}$
- Bedeutung:
  - $A_i\ \Theta$ all (Subquery) $\equiv \{...|\ \forall t \in$ Subquery: $A_i\ \Theta\ t\}$
  - ist größer als alle Werte, die sich aus Subquery ergeben
- Einschränkung bezüglich Subquery:
  - Darf nur ein Ergebnis-Attribut ermitteln — Menge nicht Relation
  - Aber mehrere Tupel sind erlaubt — Menge nicht Relation

Beispiel

- Ermittle den Kunden mit dem höchsten Kontostand

```
select KName, KAdr
from Kunde
where Kto >= all        (   select Kto
                            from Kunde
                        )
```

- Äquivalent zu folgendem Ausdruck mit EXISTS:

```
select KName, KAdr
from Kunde
where not exists        (   select *
                            from Kunde k2
                            where not k1.Kto >= k2.Kto
                        )
```

Subquery mit IN

- Nach dem Ausdruck A_i **[not] in** ... kann stehen:
  - Explizite Aufzählung von Werten: A_i **in** (2,3,5,7,11,13)
  - Eine Subquery:

    A_i **in** (**select** wert **from** Primzahlen **where** wert<=13)

    Auswertung:
    - Erst Subquery auswerten
    - In explizite Form (2,3,5,7,11,13) umschreiben
    - Dann einsetzen
    - Zuletzt Hauptquery auswerten

Beispiele

- Gegeben:
  - MagicNumbers (Name: String, Wert: Int)
  - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die prim sind

  select * **from** MagicNumbers **where** Wert **in**

  (**select** Zahl **from** Primzahlen)
- ist äquivalent zu folgender Anfrage mit EXISTS:

  select * **from** MagicNumbers **where exists**

  (**select** * **from** Primzahlen **where** Wert = Zahl)
- und zu folgender Anfrage mit SOME/ANY/ALL:

  select * **from** MagicNumbers **where**

  Wert = **some** (**select** Zahl **from** Primzahlen)

Beispiele

- Gegeben:
  - MagicNumbers (Name: String, Wert: Int)
  - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die **nicht** prim sind

  select * **from** MagicNumbers **where**
  Wert **not in** (**select** Zahl **from** Primzahlen)
- ist äquivalent zu folgender Anfrage mit EXISTS:

  select * **from** MagicNumbers **where**
  not exists (**select** * **from** Primzahlen **where** Wert = Zahl)
- und zu folgender Anfrage mit SOME/ANY/ALL:

  select * **from** MagicNumbers **where**
  Wert <> **all** (**select** Zahl **from** Primzahlen)

Sortieren

- In SQL mit ORDER BY A_1, A_2, ...
- Bei mehreren Attributen: Lexikographisch

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

order by A, B

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

order by B, A

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

- Steht am Schluß der Anfrage
- Nach Attribut kann man ASC für aufsteigend (Default) oder DESC für absteigend angeben
- Nur Attribute der SELECT-Klausel verwendbar

Beispiel

- Gegeben:
  - MagicNumbers (Name: String, Wert: Int)
  - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die prim sind, sortiert nach dem Wert beginnend mit größtem

```
select * from MagicNumbers where Wert in
 (select Zahl from Primzahlen)
  order by Wert desc
```

- Nicht möglich:

```
select Name from MagicNumbers order by Wert
```

Aggregation

- Berechnet Eigenschaften ganzer Tupel-Mengen
- Arbeitet also Tupel-übergreifend
- Aggregatfunktionen in SQL:
  - **count** Anzahl der Tupel bzw. Werte
  - **sum** Summe der Werte einer Spalte
  - **avg** Durchschnitt der Werte einer Spalte
  - **max** größter vorkommender Wert der Spalte
  - **min** kleinster vorkommender Wert
- Aggregate können sich erstrecken:
  - auf das gesamte Anfrageergebnis
  - auf einzelne Teilgruppen von Tupeln (siehe später)

Aggregation

- Aggregatfunktionen stehen in der Select-Klausel
- Beispiel:
  Gesamtzahl und Durchschnitt der Einwohnerzahl aller Länder, die mit ‘B‘ beginnen:

```
select sum (Einw), avg (Einw)
from länder
where LName like ‘B%‘
```

- Ergebnis ist immer ein einzelnes Tupel:
  Keine Mischung aggregierte/nicht aggregierte Attribute
- Aggregate wie **min** oder **max** sind ein einfaches Mittel, um Eindeutigkeit bei Subqueries herzustellen (vgl. S. 10)

Aggregation

- NULL-Werte werden ignoriert (auch bei **count**)
- Eine Duplikatelimination kann erzwungen werden
  - **count** (**distinct** KName) zählt verschiedene Kunden
  - **count** (**all** KName) zählt alle Einträge (außer **NULL**)
  - **count** (KName) ist identisch mit **count** (**all** KName)
  - **count** (*) zählt die Tupel des Anfrageergebnisses (macht nur bei NULL-Werten einen Unterschied)
- Beispiel:

  Produkt (PName, Preis, ...)

  Alle Produkte, mit unterdurchschnittlichem Preis:

```
select *
from Produkt
where Preis < (select avg (Preis) from Produkt)
```

Gruppierung

- Aufteilung der Ergebnis-Tupel in Gruppen
- Ziel: Aggregationen
- Beispiel:
  Gesamtgehalt und Anzahl Mitarbeiter pro Abteilung

Mitarbeiter

PNr	Name	Vorname	Abteilung	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

Aggregationen:

Σ Gehalt	COUNT
6300	3
4200	2

- Beachte: So in SQL nicht möglich!
  Anfrage-Ergebnis soll wieder eine Relation sein

Gruppierung

Mitarbeiter

PNr	Name	Vorname	Abteilung	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

- In SQL:
  select Abteilung, **sum** (Gehalt), **count** (*)
  from Mitarbeiter
  group by Abteilung

Abteilung	sum (Gehalt)	count (*)
01	6300	3
02	4200	2

Gruppierung

- Syntax in SQL:

select ... ← siehe unten
from ...
[**where** ...]
[**group by** $A_1, A_2, \ldots$
[**having** ...]] siehe Seite 27*ff.*
[**order by** ...] ←

- Wegen Relationen-Eigenschaft des Ergebnisses Einschränkung der **select**-Klausel. Erlaubt sind:
  - Attribute aus der Gruppierungsklausel (incl. arithmetischer Ausdrücke etc.)
  - Aggregationsfunktionen auch über andere Attribute, count (*)
  - in der Regel kein **select** * **from**...

Gruppierung

- Beispiel: Nicht möglich!!!

  Mitarbeiter

PNr	Name	Vorname	Abteilung	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

- **select** ~~PNr,~~ Abteilung, **sum** (Gehalt)
  from Mitarbeiter
  group by Abteilung

~~„PNr“~~	Abteilung	Gehalt
~~„001,002,003“~~	01	6300
~~„004,005“~~	02	4200

Gruppierung mehrerer Attribute

- Etwa sinnvoll in folgender Situation:

PNr	Name	Vorname	Abteilung	Gruppe	Gehalt
001	Huber	Erwin	01	01	2000
002	Mayer	Hugo	01	02	2500
003	Müller	Anton	01	02	1800
004	Schulz	Egon	02	01	2500
005	Bauer	Gustav	02	01	1700

Debitoren
Kreditoren
Fernsehgeräte
Buchhaltung
Produktion

Gesamtgehalt in jeder Gruppe:

select Abteilung, Gruppe,
sum(Gehalt)
from Mitarbeiter
group by Abteilung, Gruppe

Abt.	Gr.	Σ Geh.
01	01	2000
01	02	4300
02	01	4200

Gruppierung mehrerer Attribute

Oft künstlich wegen **select**-Einschränkung:

Mitarbeiter ⋈ Abteilungen

PNr	Name	Vorname	ANr	AName	Gehalt
001	Huber	Erwin	01	Buchhaltung	2000
002	Mayer	Hugo	01	Buchhaltung	2500
003	Müller	Anton	01	Buchhaltung	1800
004	Schulz	Egon	02	Produktion	2500
005	Bauer	Gustav	02	Produktion	1700

- Nicht möglich, obwohl AName von ANr funktional abh.:
  ~~**select** ANr, AName, sum(Gehalt) **from** ... **where** ... **group by** ANr~~
- Aber wegen der funktionalen Abhängigkeit identisch mit:
  select ANr, AName, sum(...) **from** ... **where** ... **group by** ANr, AName
- Weitere Möglichkeit (ebenfalls wegen Abhängigkeit):
  select ANr, **max** (AName), sum(...) **from** ... **where** ... **group by** ANr

Die Having-Klausel

- Motivation:
  Ermittle das Gesamt-Einkommen in jeder Abteilung, die mindestens 5 Mitarbeiter hat
- In SQL nicht möglich:

```
select      ANr, sum (Gehalt)
from        Mitarbeiter
where       count (*) >= 5        GEHT NICHT !
group by    ANr                   STATT DESSEN:
having      count (*) >= 5
```

- Grund: Gruppierung wird erst nach den algebraischen Operationen ausgeführt

Auswertung der Gruppierung

An folgendem Beispiel:

select A, **sum**(D)
from ... **where** ...
group by A, B
having sum (D) < 10 **and max** (C) = 4

1. Schritt:

from/where

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7

2. Schritt:

Gruppenbildung

A	B	C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7

3. Schritt:

Aggregation

A	B	sum(D)	max(C)
1	2	9	4
2	3	4	3
3	3	12	6

temporäre „nested relation“

Auswertung der Gruppierung

An folgendem Beispiel:

```
select A, sum(D)
from ... where ...
group by A, B
having sum (D) < 10 and max (C) = 4
```

3. Schritt:

Aggregation

A	B	sum(D)	max(C)
1	2	9	4
2	3	4	3
3	3	12	6

4. Schritt:

having (=Selektion)

A	B	sum(D)	max(C)
1	2	9	4

5. Schritt:

Projektion

A	sum(D)
1	9

Generierung eindeutiger Schlüssel

- Keine Standard-Konstrukte in SQL
- ORACLE: Sequenz als eigenes DB-Objekt

  create sequence SName ;
  insert into Mitarbeiter values (SName.nextval, ‘Müller‘, ...) ;
- MySQL: Auto-Increment-Attribute einer Tabelle

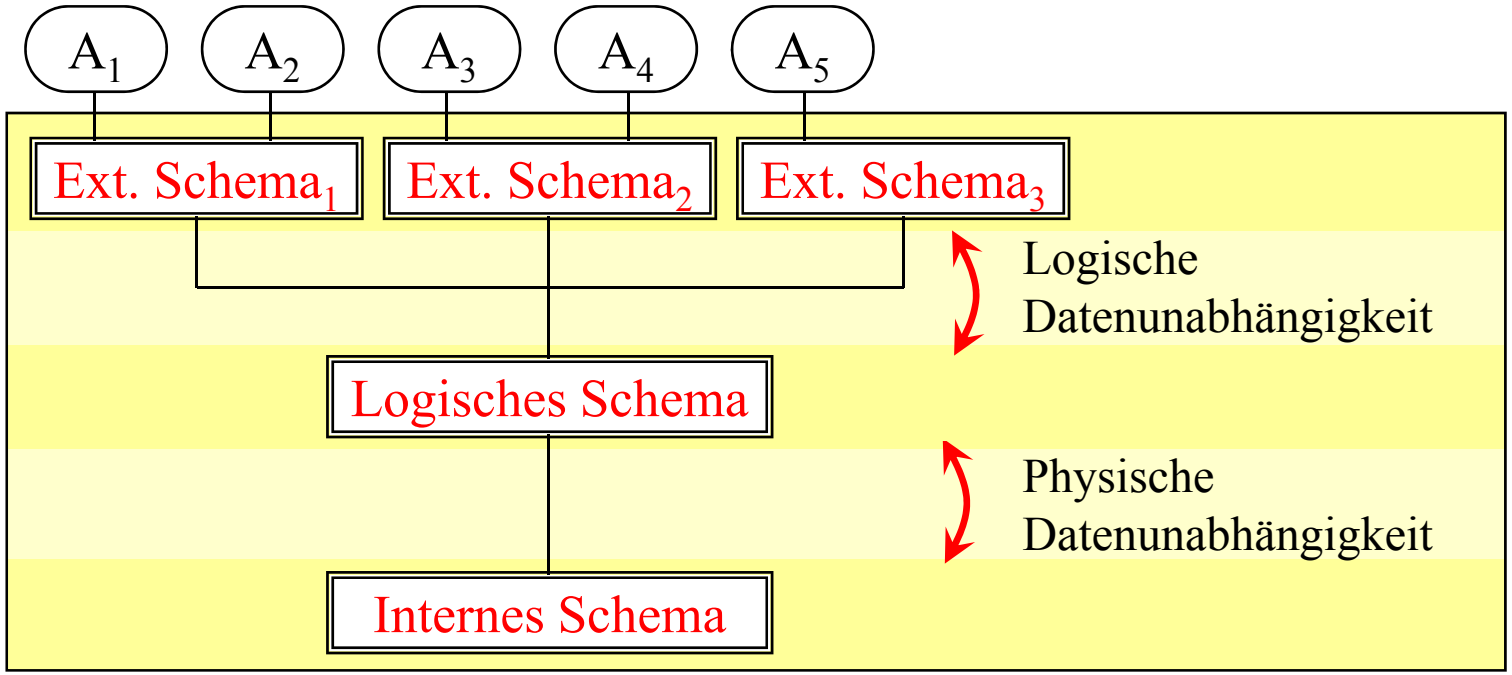
  create table Mitarbeiter
  (PNr integer not null auto_increment,
  ...
) ;
  insert into Mitarbeiter values (NULL, ‘Müller ‘, ...) ;
  Bei NULL wird automatisch Maximalwert+1 gesetzt

Architektur eines DBS

Drei-Ebenen-Architektur zur Realisierung von

- **physischer**
- **und logischer**

Datenunabhängigkeit (nach ANSI/SPARC)



Externe Ebene

- Gesamt-Datenbestand ist angepasst, so dass jede Anwendungsgruppe nur die Daten sieht, die sie...
  - sehen will (Übersichtlichkeit)
  - sehen soll (Datenschutz)
- Logische Datenunabhängigkeit

- In SQL:
  Realisiert mit dem Konzept der Sicht (View)

Was ist eine Sicht (View)?

- Virtuelle Relation
- Was bedeutet virtuell?
  - Die View sieht für den Benutzer aus wie eine Relation:
    - **select** ... **from** $View_1$, $Relation_2$, ... **where** ...
    - mit Einschränkung auch: **insert, delete** und **update**
  - Aber die Relation ist nicht real existent/gespeichert; Inhalt ergibt sich durch Berechnung aus anderen Relationen
- Besteht aus zwei Teilen:
  - Relationenschema für die View (nur rudimentär)
  - Berechnungsvorschrift, die den Inhalt festlegt: SQL-Anfrage mit **select** ... **from** ... **where**

Viewdefinition in SQL

- Das folgende DDL-Kommando erzeugt eine View
  create [or replace] view *VName* [(A_1, A_2, ...)]* **as select** ...
- Beispiel: Eine virtuelle Relation Buchhalter, nur mit den Mitarbeitern der Buchhaltungsabteilung:
  create view Buchhalter **as**
  select PNr,Name,Gehalt **from** Mitarbeiter **where** ANr=01
- Die View *Buchhalter* wird erzeugt:

Mitarbeiter

PNr	Name	Vorname	ANr	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

Buchhalter

PNr	Name	Gehalt
001	Huber	2000
002	Mayer	2500
003	Müller	1800

*relativ unüblich

Konsequenzen

- Automatisch sind in dieser View alle Tupel der Basisrelation, die die Selektionsbedingung erfüllen
- An diese können beliebige Anfragen gestellt werden, auch in Kombination mit anderen Tabellen (Join) etc:

  select * from Buchhalter where Name like ‘B%‘
- In Wirklichkeit wird lediglich die View-Definition in die Anfrage eingesetzt und dann ausgewertet:

Buchhalter:

select PNr,Name,Gehalt
from Mitarbeiter **where** ANr=01

select * from Buchhalter where Name like ‘B%‘

select * from (select PNr, Name, Gehalt
from Mitarbeiter where ANr=01)
where Name like ‘B%‘

Konsequenzen

- Bei Updates in der Basisrelation (Mitarbeiter) ändert sich auch die virtuelle Relation (Buchhalter)
- Umgekehrt können (mit Einschränkungen) auch Änderungen an der View durchgeführt werden, die sich dann auf die Basisrelation auswirken
- Eine View kann selbst wieder Basisrelation einer neuen View sein (View-Hierarchie)
- Views sind ein wichtiges Strukturierungsmittel für Anfragen und die gesamte Datenbank

Löschen einer View:

drop view *VName*

In Views erlaubte Konstrukte

- Folgende Konstrukte sind in Views erlaubt:
  - Selektion und Projektion
    (incl. Umbenennung von Attributen, Arithmetik)
  - Kreuzprodukt und Join
  - Vereinigung, Differenz, Schnitt
  - Gruppierung und Aggregation
  - Die verschiedenen Arten von Subqueries
- Nicht erlaubt:
  - Sortieren

Insert/Delete/Update auf Views

- Logische Datenunabhängigkeit:
  - Die einzelnen Benutzer-/Anwendungsgruppen sollen ausschließlich über das externe Schema (d.h. Views) auf die Datenbank zugreifen (Übersicht, Datenschutz)
  - Insert, Delete und Update auf Views erforderlich
- Effekt-Konformität
  - View soll sich verhalten wie gewöhnliche Relation
  - z.B. nach dem Einfügen eines Tupels muß das Tupel in der View auch wieder zu finden sein, usw.
- Mächtigkeit des View-Mechanismus
  - Join, Aggregation, Gruppierung usw.
  - Bei komplexen Views Effekt-Konformität unmöglich

Insert/Delete/Update auf Views

- Wir untersuchen die wichtigsten Operationen in der View-Definition auf diese Effekt-Konformität
  - Projektion
  - Selektion
  - Join
  - Aggregation und Gruppierung
- Wir sprechen von Projektions-Sichten usw.
  - Änderung auf Projektionssicht muß in Änderung der Basisrelation(en) transformiert werden
- Laufendes Beispiel:
  - MGA (Mitarbeiter, Gehalt, Abteilung)
  - AL (Abteilung, Leiter)

Projektionssichten

- Beispiel:

```
create view MA as
select Mitarbeiter, Abteilung
from MGA
```

- Keine Probleme beim Löschen und Update:

  delete from MA **where** Mitarbeiter = ...
  → **delete from** MGA **where** Mitarbeiter = ...

- Bei Insert müssen wegprojizierte Attribute durch NULL-Werte oder bei der Tabellendefinition festgelegte Default-Werte belegt werden:

  insert into MA **values** (‘Weber‘, 001)
  → **insert into** MGA **values** (‘Weber‘, NULL, 001)

Projektionssichten

- Problem bei Duplikatelimination (**select distinct**):
  Keine eindeutige Zuordnung zwischen Tupeln der View und der Basisrelation:
- Bei Arithmetik in der Select-Klausel: Rückrechnung wäre erforderlich:
  create view P **as select** 3*x*x*x+2*x*x+x+1 **as** y **from** A
- Der folgende Update wäre z.B. problematisch:
  insert into P **set** y = 0 **where** ...
- womit müsste x besetzt werden?
  Mit der Nullstelle des Polynoms $f(x) = 3x^3 + 2x^2 + x + 1$
  Nullstellensuche kein triviales mathematisches Problem

Kein **insert/delete/update** bei **distinct**/Arithmetik

Selektionssichten

- Beispiel:

```
create view MG as
select * from MGA
where Gehalt >= 20
```

- Beim Ändern (und Einfügen) kann es passieren, dass ein Tupel aus der View verschwindet, weil es die Selektionsbedingung nicht mehr erfüllt:
  update MG **set** Gehalt = 19 **where** Mitarbeiter = ‘Huber‘
- Huber ist danach nicht mehr in MG
- Dies bezeichnet man als Tupel-Migration:
  Tupel verschwindet, taucht aber vielleicht dafür in anderer View auf

Selektionssichten

- Dies ist manchmal erwünscht
  - Mitarbeiter wechselt den zuständigen Sachbearbeiter, jeder Sachbearbeiter arbeitet mit „seiner“ View
- Manchmal unerwünscht
  - Datenschutz
- Deshalb in SQL folgende Möglichkeit:

```
create view MG as
select * from MGA
where Gehalt >= 20
with check option
```

- Die Tupel-Migration wird dann unterbunden
  Fehlermeldung bei: **update** MG **set** Gehalt = 19 **where** ...

Join-Views

- Beispiel:

```
create view MGAL as
select Mitarbeiter, Gehalt, MGA.Abteilung, Leiter
from MGA, AL
where MGA.Abteilung = AL.Abteilung
```

- Insert in diese View nicht eindeutig übersetzbar:
  insert into MGAL **values** (‘Schuster‘, 30, 001, ‘Boss‘)
  → **insert into** MGA **values** (‘Schuster‘, 30, 001)

  wenn kein Tupel (001, ‘Boss‘) in AL existiert:
  → **insert into** AL **values** (001, ‘Boss‘)
  → **update** AL **set** Leiter=‘Boss‘ **where** Abteilung=001
  oder Fehlermeldung ?
- Daher: Join-View in SQL nicht updatable

Aggregation, group by, Subquery

- Auch bei Aggregation und Gruppierung ist es nicht möglich, eindeutig auf die Änderung in der Basisrelation zu schließen
- Subqueries sind unproblematisch, sofern sie keinen Selbstbezug aufweisen (Tabelle in from-Klausel der View wird nochmals in Subquery verwendet)

Eine View, die keiner der angesprochenen Problemklassen angehört, heisst **Updatable View**.
Insert, delete und update sind möglich.

Materialisierte View

Hier nur Begriffserklärung:

- Eine sog. materialisierte View ist keine virtuelle Relation sondern eine real gespeicherte
- Der Inhalt der Relation wurde aber durch eine Anfrage an andere Relationen und Views ermittelt
- In SQL einfach erreichbar durch Anlage einer Tabelle *MVName* und Einfügen der Tupel mit:

  insert into *MVName* (**select ... from ... where**)
- Bei Änderungen an den Basisrelationen keine automatische Änderung in *MVName* und umgekehrt
- DBS bieten oft auch spezielle Konstrukte zur Aktualisierung (Snapshot, Trigger)

Rechtevergabe

- Basiert in SQL auf Relationen bzw. Views
- Syntax:

  grant *Rechteliste*
  on *Relation* bzw. *View*
  to *Benutzerliste*
  [**with grant option**]

- *Rechteliste*:
  - all [privileges]
  - select, insert, delete (mit Kommas sep.)
  - update (optional in Klammern: Attributnamen)

Rechtevergabe

- *Benutzerliste*:
  - Benutzernamen (mit Passwort identifiziert)
  - **to public** (an alle)
- Grant Option:
  Recht, das entsprechende Privileg selbst weiterzugeben
- Rücknahme von Rechten:
  revoke *Rechteliste*
  on *Relation*
  from *Benutzerliste*
  [**restrict**] *Abbruch, falls Recht bereits weitergegeben*
  [**cascade**] *ggf. Propagierung der Revoke-Anweisung*

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

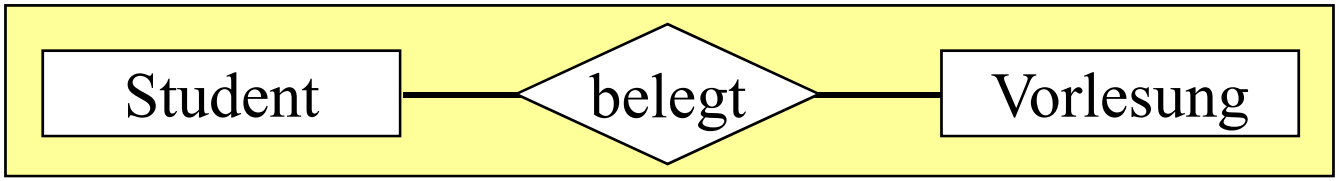
Kapitel 6: Das E/R-Modell

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Schema-Entwurf

- Generelle Aufgabe:
  Finde eine formale Beschreibung (Modell) für einen zu modellierenden Teil der realen Welt
- Zwischenstufen:
  - Beschreibung durch natürliche Sprache (Pflichtenheft):
    Beispiel: *In der Datenbank sollen alle Studierenden mit den durch sie belegten Lehrveranstaltungen gespeichert sein*
  - Beschreibung durch abstrakte grafische Darstellungen:

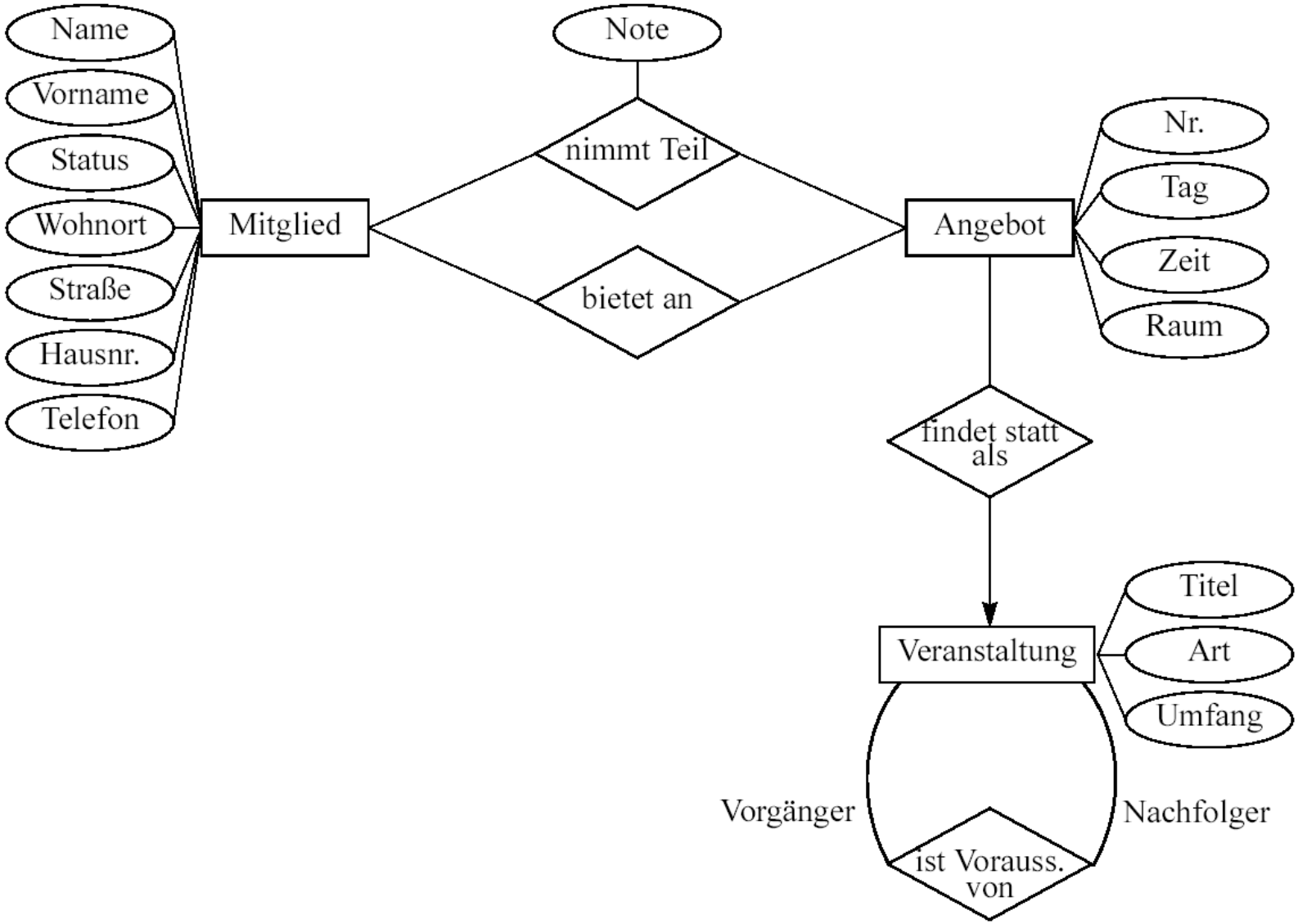


  - Beschreibung im relationalen Modell:
    create table student (...) ;
    create table vorlesung (...) ;

Das Entity/Relationship Modell

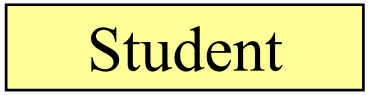
- Dient dazu, für einen Ausschnitt der realen Welt ein konzeptionelles Schema zu erstellen
- Grafische Darstellung: E/R-Diagramm
- Maschinenfernes Datenmodell
- Hohes Abstraktionsniveau
- Überlegungen zur Effizienz spielen keine Rolle
- Das E/R-Modell muss in ein relationales Schema überführt werden
  - Einfache Grundregeln zur Transformation
  - Gewinnung eines *effizienten* Schemas erfordert tiefes Verständnis vom Zielmodell

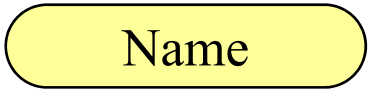
Beispiel: Lehrveranstaltungen



Elemente des E/R-Modells

- Entities:
  (eigentlich: Entity Sets)
  Objekttypen
- Attribute:
  Eigenschaften
- Relationships:
  Beziehungen zw. Entities

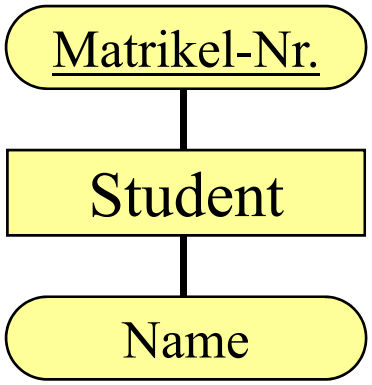




Entscheidende Aufgabe des Schema-Entwurfs:

- Finde *geeignete* Entities, Attribute und Relationships

Entities und Attribute



Entities

- Objekte, Typen, "Seiendes"
- Objekte der realen Welt, unterscheidbar
- Bsp: Mensch, Haus, Vorlesung, Bestellung, ...

Attribute

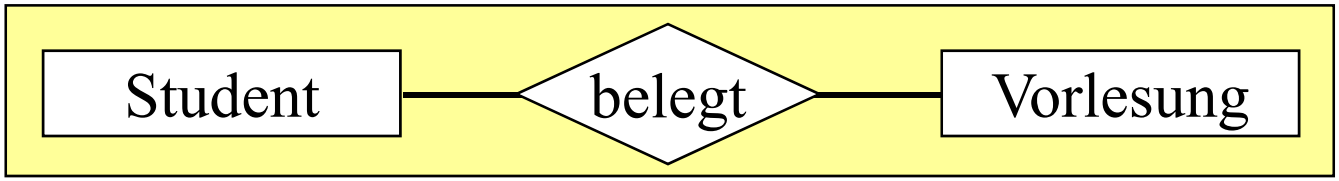
- Entities durch charakterisierende Eigenschaften beschrieben
- Einfache Datentypen wie INT, STRING usw.
- Bsp: Farbe, Gewicht, Name, Titel, ...
- Häufig beschränkt man sich auf die wichtigsten Attribute

Schlüssel

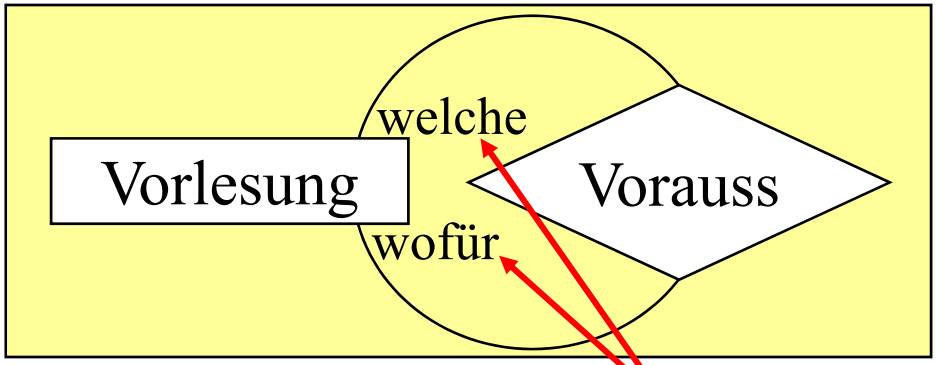
- ähnlich definiert wie im relationalen Modell
- Primärschlüssel-Attribute werden unterstrichen

Relationships (Beziehungen)

- Stellen Zusammenhänge zwischen Entities dar
- Beispiele:



Ausprägung: belegt (Anton, Informatik 1)
belegt (Berta, Informatik 1)
belegt (Caesar, Wissensrepräsentation)
belegt (Anton, Datenbanksysteme 1)

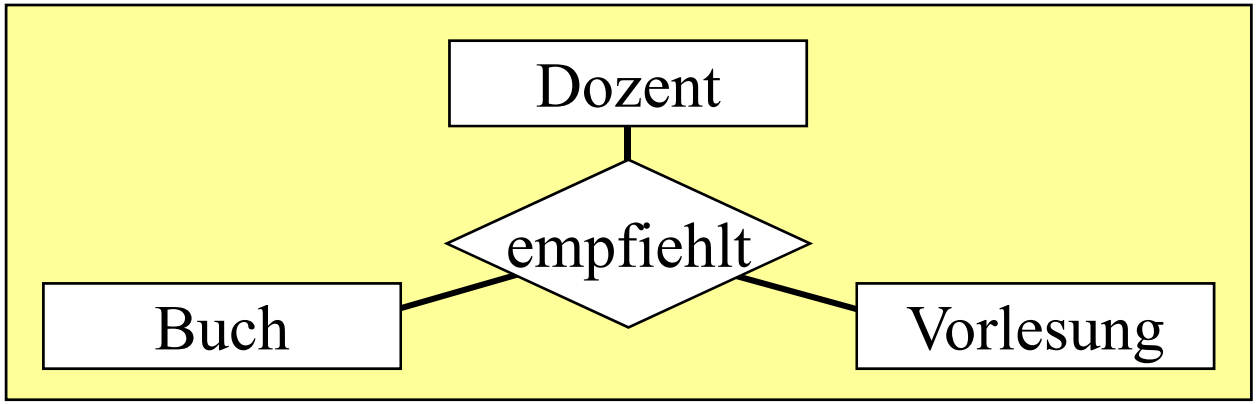


Ausprägung:
Vorauss (I 1, DBS 1)
Vorauss (I 1, Software-Eng.)
Vorauss (DBS 1, DBS 2)
Vorauss (I 1, Wissensrepr.)

unterschiedliche Rollen

Relationships (Beziehungen)

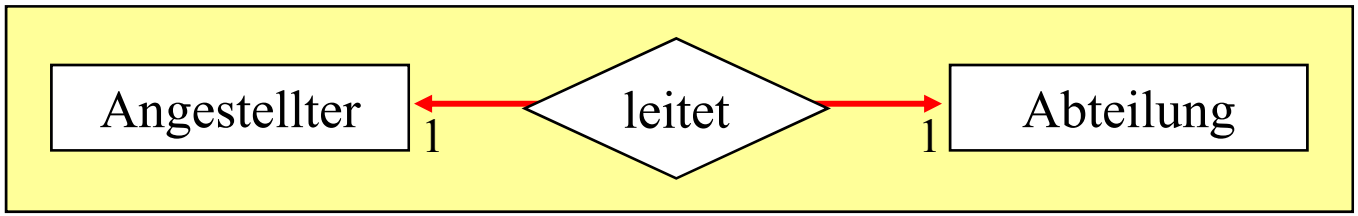
- Relationships können eigene Attribute haben.
  Beispiel: Student *belegt* Vorlesung
  belegt hat Attribut *Note*
- Mehrstellige (ternäre usw.) Relationships:



Ausprägung: empfiehlt (Böhm, Heuer&Saake, DBS 1)
empfiehlt (Böhm, Kemper&Eickler, DBS 1)
empfiehlt (Böhm, Bishop, Informatik 1)
empfiehlt (Böhm, Bishop, Programmierkurs)

Funktionalität von Relationships

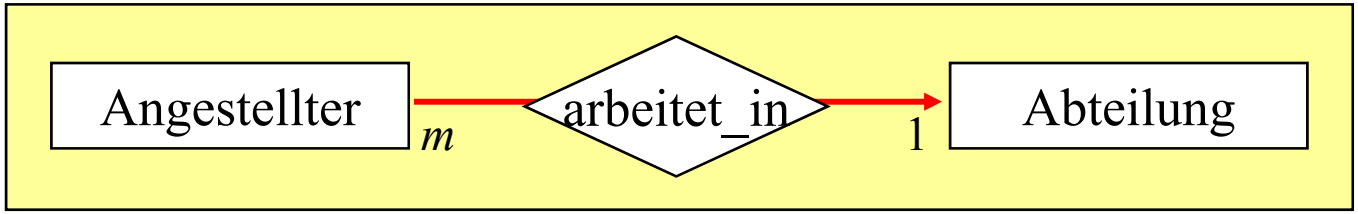
- 1:1-Beziehung (one-to-one-relationship):



- Charakteristik:
  Jedes Objekt aus dem linken Entity steht in Beziehung zu höchstens einem Objekt aus dem rechten Entity und umgekehrt.
- Grafische Notation: Pfeile auf jeder Seite
- Beispiel gilt unter der Voraussetzung, dass jede Abteilung max. einen Leiter hat und kein Angestellter mehrere Abteilungen leitet

Funktionalität von Relationships

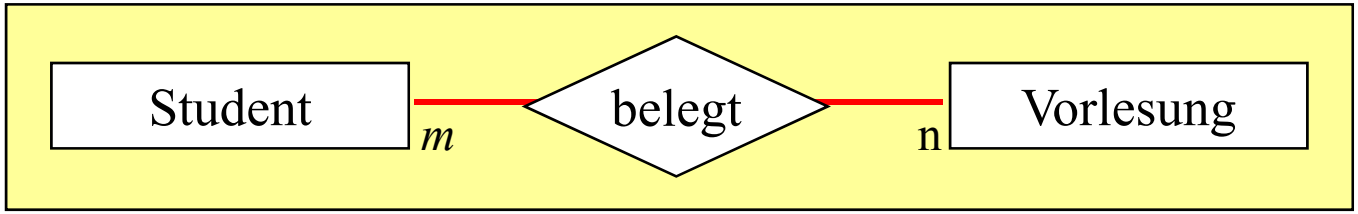
- *m*:1-Beziehung (many-to-one-relationship)



- Charakteristik:
  Jedes Objekt auf der *"many"*-Seite steht in Beziehung zu höchstens einem Objekt auf der *"one"*-Seite (im Allgemeinen nicht umgekehrt)
- Grafische Notation:
  Pfeil in Richtung zur *"one"*-Seite

Funktionalität von Relationships

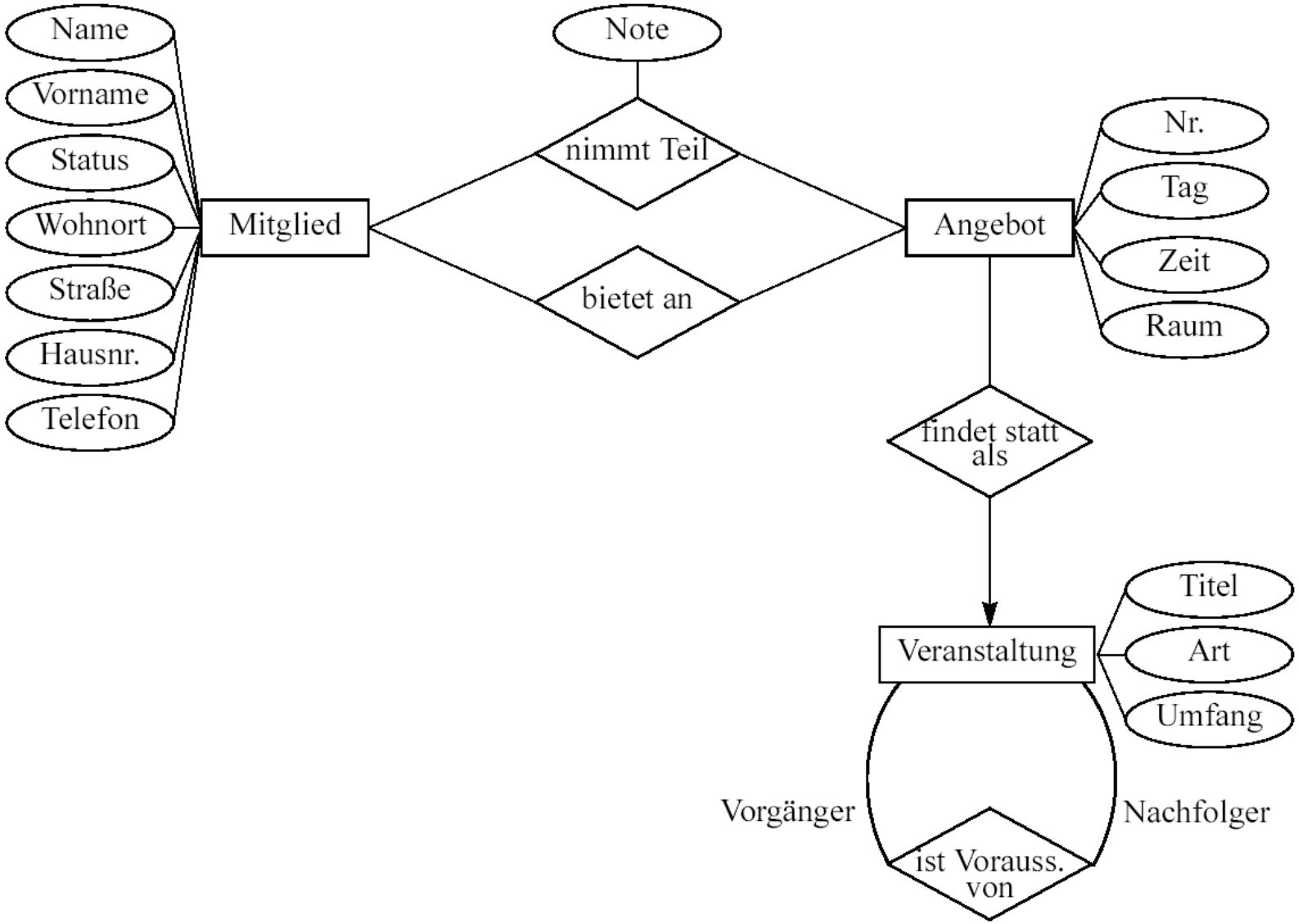
- *m*:*n*-Beziehung (many-to-many-relationship)



- Charakteristik:
  Jedes Objekt auf der linken Seite kann zu mehreren Objekten auf der rechten-Seite in Beziehung stehen (d.h. keine Einschränkung)
- Grafische Notation:
  Kein Pfeil

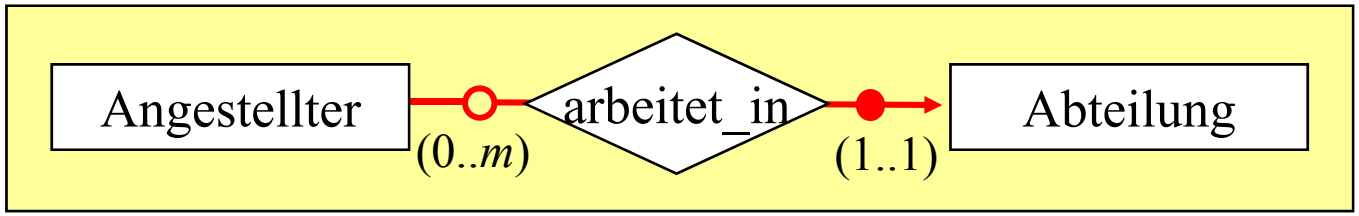
Beispiel-Ausprägung:
belegt (Anton, Informatik 1)
belegt (Berta, Informatik 1)
belegt (Caesar, Wissensrepräsentation)
belegt (Anton, Datenbanksysteme 1)

Beispiel: Lehrveranstaltungen



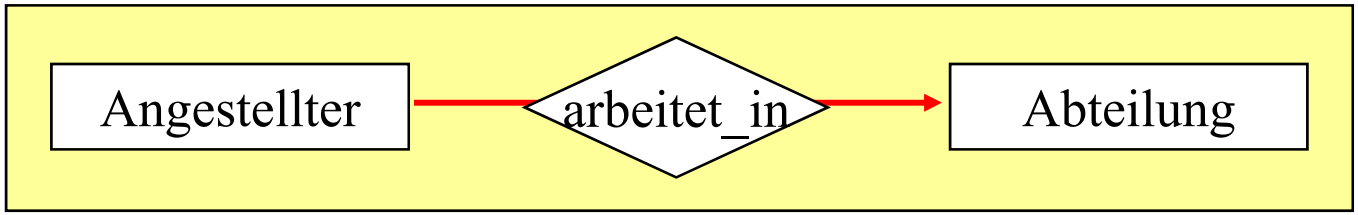
Optionalität

- Das E/R-Modell trifft lediglich Aussagen darüber, ob ein Objekt zu mehreren Objekten in Beziehung stehen darf oder zu max. einem
- Darüber, ob es zu mindestens einem Objekt in Beziehung stehen muss, keine Aussage
- Erweiterung der Notation mit:
  - Geschlossener Kreis: Verpflichtend mindestens eins.
  - Offener Kreis: Optional
  - Gilt auch für Attribute (vgl. NOT NULL)

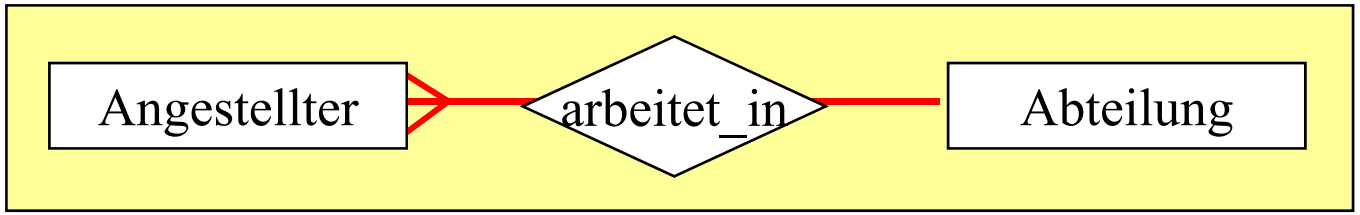


Verschiedene Notationen

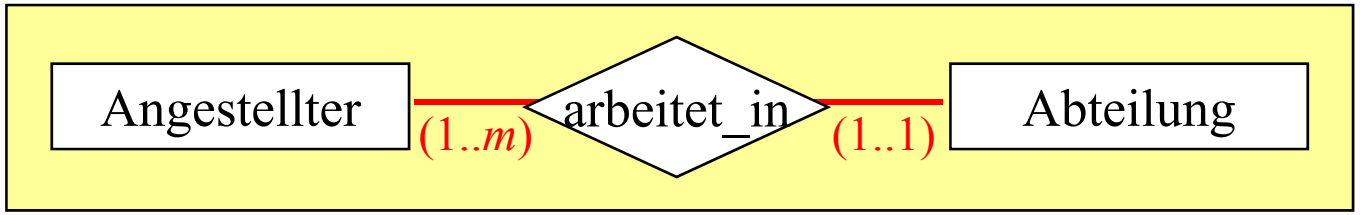
- Pfeil-Notation der Funktionalität:



- Krähenfuß-Notation:



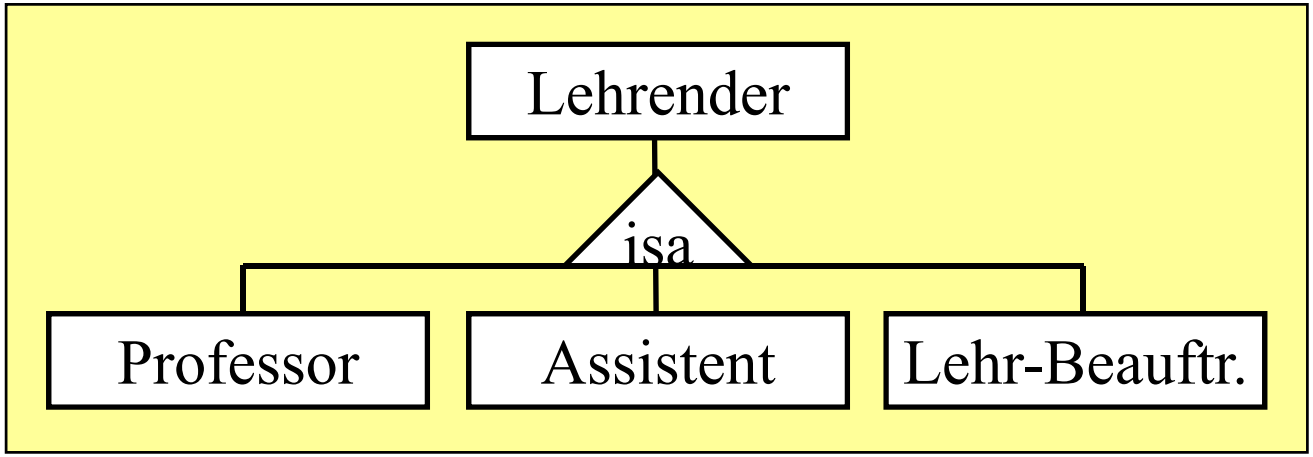
- Kardinalitäts-Notation:



auch *m*, (1..∞), (1..*), verschiedene Kombinationsformen

ISA-Beziehung/Vererbung

- Das Erweiterte E/R-Modell kennt Vererbungs-Beziehungen für Entities
- Beispiel:

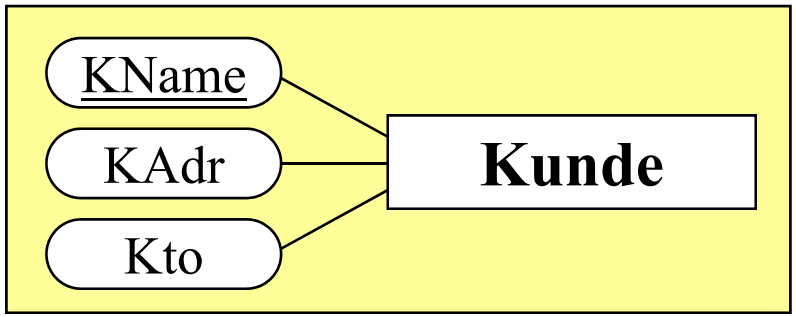


- Charakteristik und Bedeutung:
  - Assistent *ist ein* Lehrender (engl.: *is a*)
  - Vererbung aller Attribute und Beziehungen

Vom E/R-Modell zur Relation

Einfache Umsetzungsregeln:

- Entities und Attribute:



  - Jedem Entity-Typ wird eine Relation zugeordnet
  - Jedes Attribut des Entity wird ein Attribut der Relation
  - Der Primärschlüssel des Entity wird Primärschlüssel der Relation
  - Attribute können im weiteren Verlauf dazukommen

Kunde (KName, KAdr, Kto)

Vom E/R-Modell zur Relation

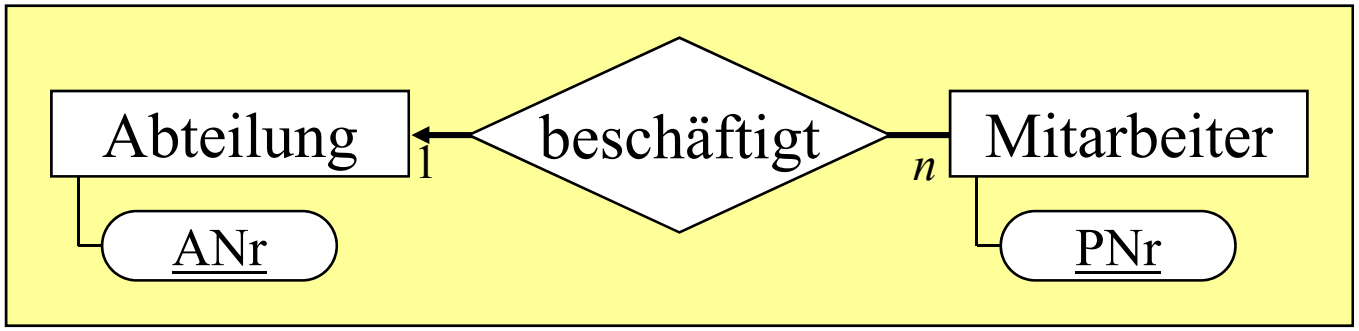
- Bei Relationships:
  Umsetzung abh. von Funktionalität/Kardinalität:
  - 1:1
  - 1:*n*

  } Zusätzliche Attribute in bestehende Relationen
  - *n*:*m* Erzeugung einer zusätzlichen Relation

  Die ersten beiden Funktionalitäten sind Spezialfälle der dritten.
  Deshalb ist es immer *auch* möglich, zusätzliche Relationen einzuführen, jedoch nicht erforderlich

Vom E/R-Modell zur Relation

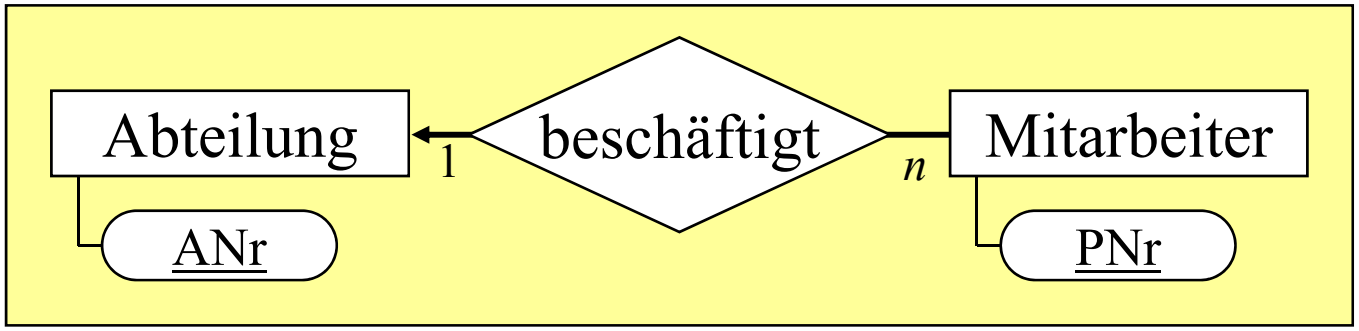
- One-To-Many Relationships:



  - Keine zusätzliche Tabelle wird angelegt
  - Der Primärschlüssel der Relation auf der **one**-Seite der Relationship kommt in die Relation der **many**-Seite (Umbenennung bei Namenskonflikten)
  - Die neu eingeführten Attribute werden Fremdschlüssel
  - Die Primärschlüssel der Relationen ändern sich nicht
  - Attribute der Relationship werden ebenfalls in die Relation der **many**-Seite genommen (kein Fremdschl.)

Vom E/R-Modell zur Relation

- Beispiel One-To-Many-Relationship:



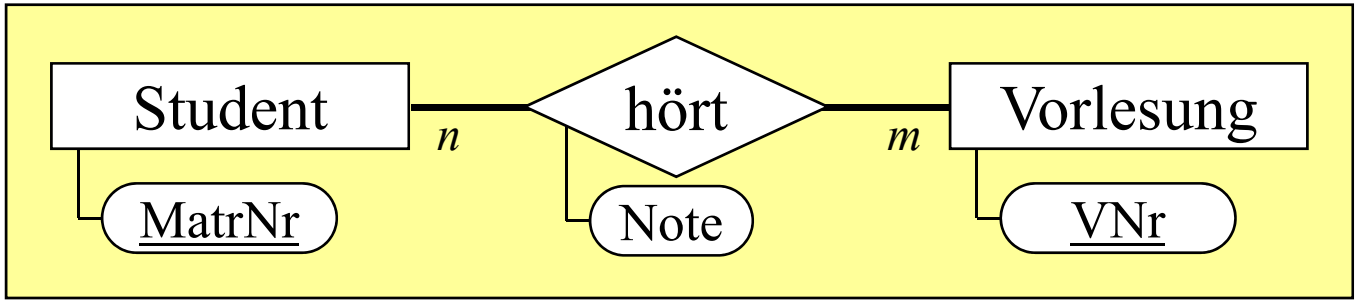
Abteilung (ANr, Bezeichnung, ...)

Mitarbeiter (PNr, Name, Vorname, ..., ANr)

```
create table Mitarbeiter (
        PNr char(3) primary key,
        ...
        ANr char(3) references Abteilung (ANr) ) ;
```

Vom E/R-Modell zur Relation

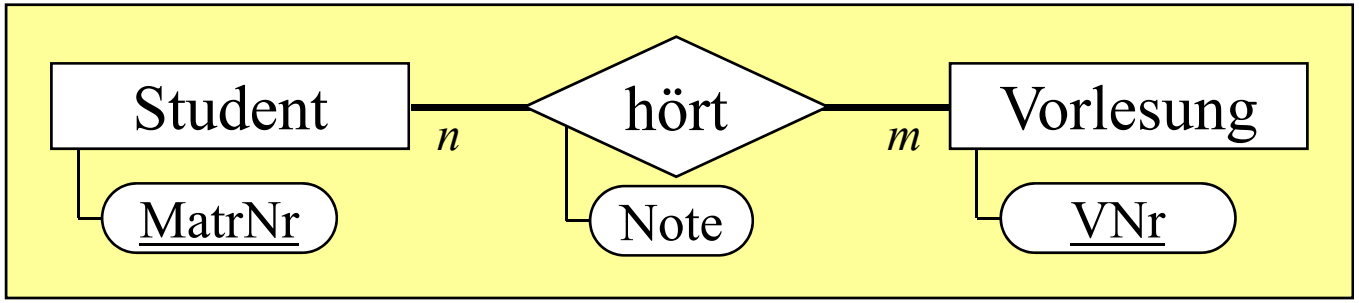
- Many-To-Many-Relationships:



  - Einführung einer zusätzlichen Relation mit dem Namen der Relationship
  - Attribute: Die Primärschlüssel-Attribute der Relationen, den Entities beider Seiten zugeordnet sind
  - Diese Attribute sind jeweils Fremdschlüssel
  - Zusammen sind diese Attribute Primärschlüssel der neuen Relation
  - Attribute der Relationship ebenfalls in die neue Rel.

Vom E/R-Modell zur Relation

- Beispiel: Many-To-Many-Relationships



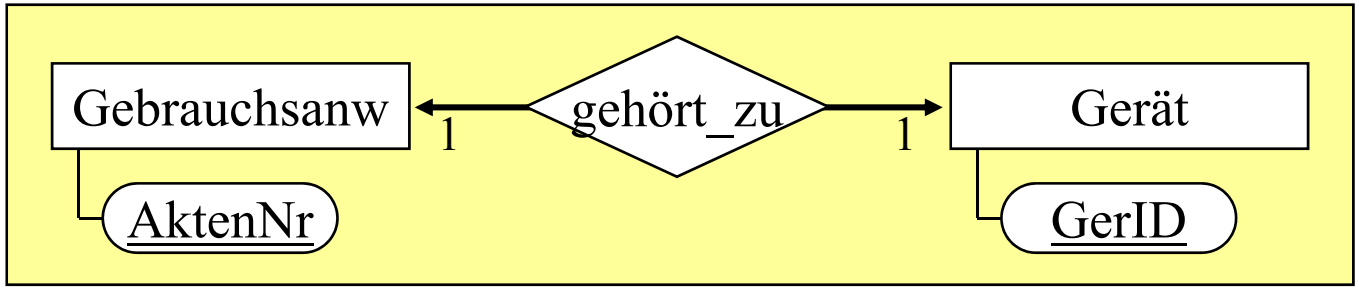
Student (MatrNr, ...) Vorlesung (VNr, ...)

Hoert (MatrNr, VNr, Note)

...
primary key (MatrNr, VNr),
foreign key MatrNr **references** Student,
foreign key VNr **references** Vorlesung...

Vom E/R-Modell zur Relation

- One-To-One-Relationships:



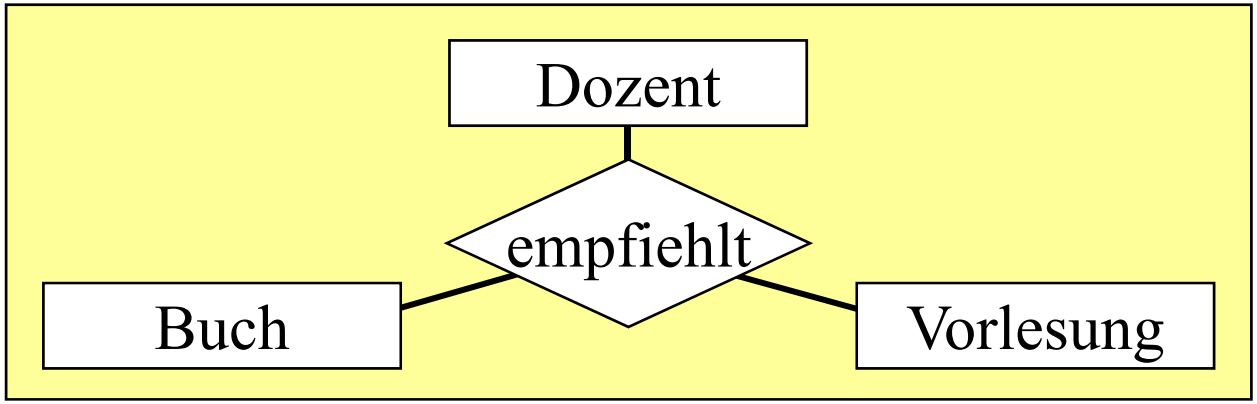
  - Die beiden Entities werden zu einer Relation zusammengefasst
  - Einer der Primärschlüssel der Entities wird Primärschlüssel der Relation

    Geraet (GerID, ..., AktenNr, ...)

  - Häufig auch Umsetzung wie bei 1:n-Beziehung (insbes. wenn eine Seite optional ist), wobei die Rollen der beteiligten Relationen austauschbar sind

Vom E/R-Modell zur Relation

- Mehrstellige Relationen

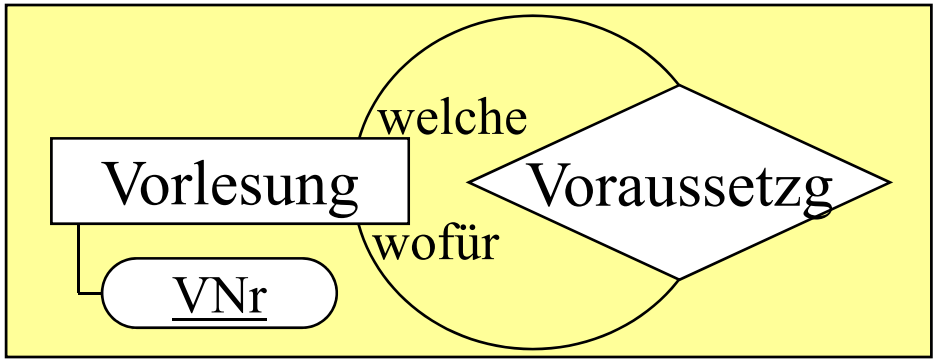


  - Eigene Relation für *empfiehlt*, falls mehr als eine Funktionalität **many** ist:

Dozent (PNr, ...)
Buch (ISBN, ...)
Vorlesung (VNr, ...)
empfiehlt (PNr, ISBN, VNr)

Vom E/R-Modell zur Relation

- Selbstbezug

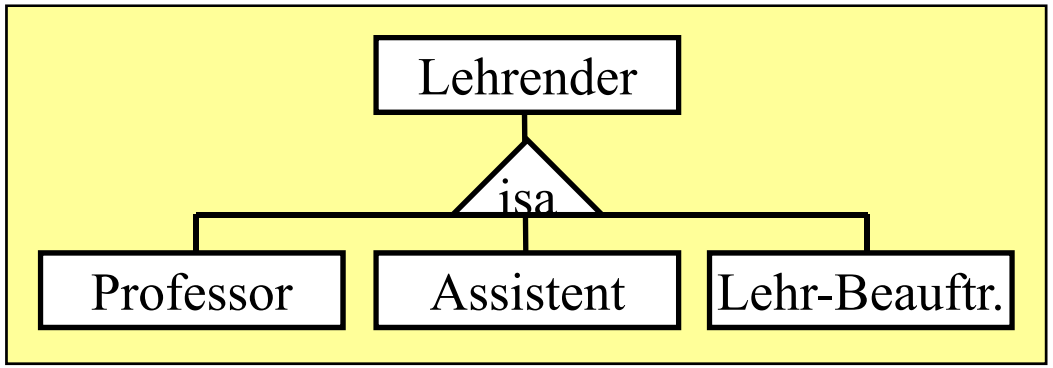


Keine Besonderheiten:
Vorgehen je nach Funktionalität.

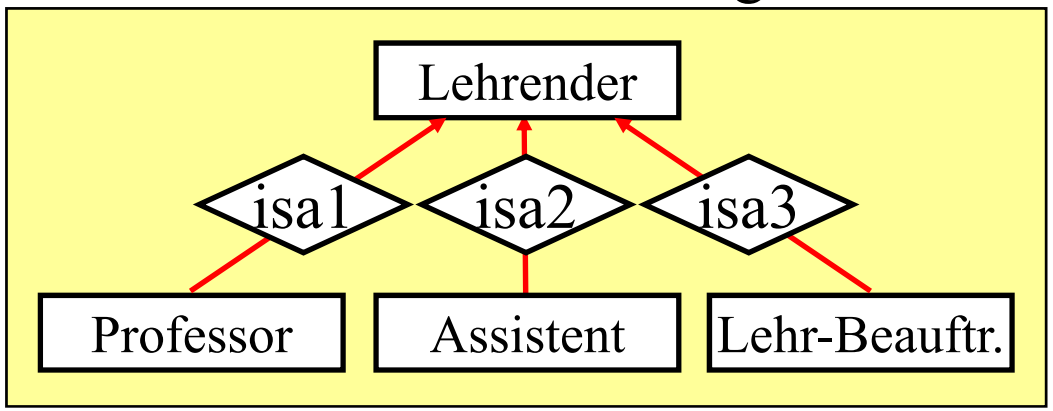
Vorlesung (VNr, ...)
Voraussetzg (Welche, Wofuer)
foreign key Welche **references** Vorlesung (VNr),
foreign key Wofuer **references** Vorlesung (VNr)

Vom E/R-Modell zur Relation

- Umsetzung der ISA-Beziehung:



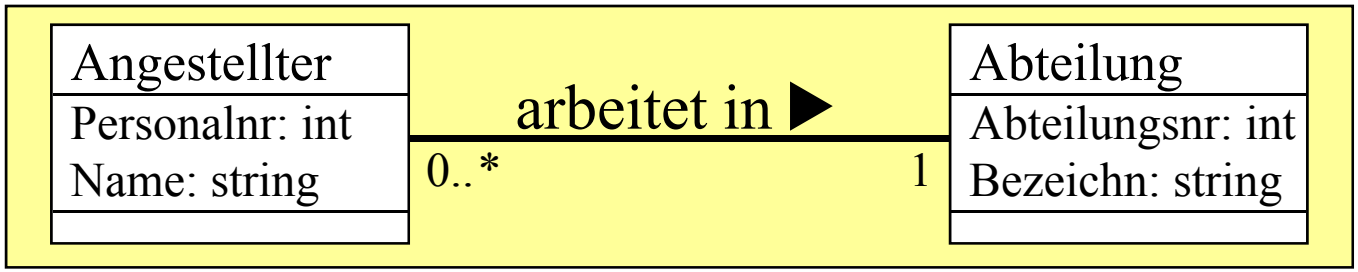
- Meist wie bei 1:*m*-Beziehungen:



- Alternative: Attribute und Relationships von *Lehrender* explizit in *Professor* (...) übernehmen

UML

- Wegen Unübersichtlichkeit und Einschränkungen Verdrängung der E/R-Diagramme durch UML
- Unterschiede:
  - Attribute werden direkt im Entity-Kasten notiert
  - Relationships ohne eigenes Symbol (nur Verbindung) Ausnahme: Ternäre Relationships mit Raute
  - Verschiedene Vererbungsbeziehungen, Part-of-Bez.
  - (Methoden: Nicht gebraucht bei DB-Modellierung)



Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 7: Normalformen

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

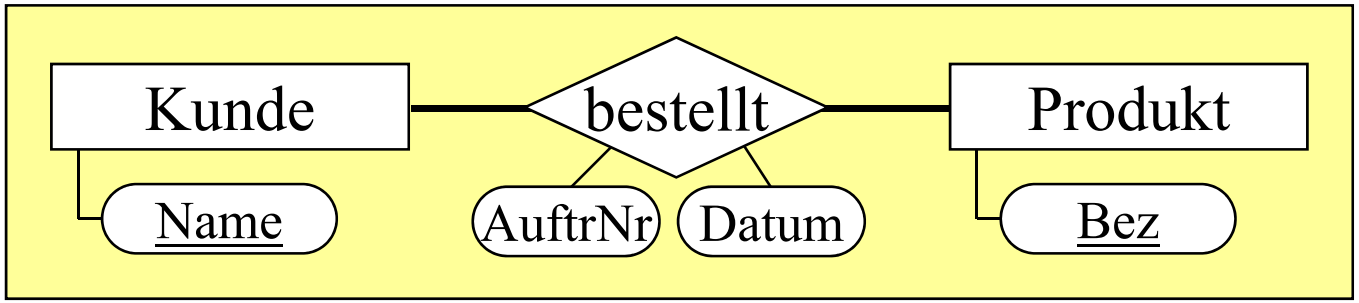
http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Relationaler Datenbank-Entwurf

- Schrittweises Vorgehen:
  - Informelle Beschreibung: Pflichtenheft
  - Konzeptioneller Entwurf: E/R-Diagramm
  - Relationaler DB-Entwurf: Relationenschema
- In diesem Kapitel:
  Normalisierungstheorie als formale Grundlage für den relationalen DB-Entwurf
- Zentrale Fragestellungen:
  - Wie können Objekte und deren Beziehungen ins relationale Modell überführt werden
  - Bewertungsgrundlagen zur Unterscheidung zwischen „guten“ und „schlechten“ relationalen DB-Schemata

Motivation Normalisierung

- Nicht immer liefert das E/R-Modell ein redundanzfreies Datenbankschema:



Schema:

Kunde	(Name, ...)
Produkt	(Bez, ...)
bestellt	(Name, Bez, AuftrNr, Datum)

Redundanz: Kundenauftrag für mehrere Produkte

Motivation Normalisierung

- Tabelleninhalt Bestellt:

Name	Bez	AuftrNr	Datum
Huber	Schraube	01	01.01.02
Huber	Nagel	01	01.01.02
Huber	Schraube	02	01.02.02
Meier	Schraube	03	05.01.02

- Hier gibt es offensichtlich einige Redundanzen:
  - →zwei verschiedene Datums zu einem Auftrag möglich
  - →zwei verschiedene Kunden zu einem Auftrag möglich
- Redundanzen durch funktionale Abhängigkeiten
  - Datum funktional abhängig von AuftrNr
  - Name funktional abhängig von AuftrNr

Weiteres Beispiel

Datenbankschema aus Kapitel 3:

Kunde	(KName, KAdr, Kto)
Auftrag	(KName, Ware, Menge)
Lieferant	(LName, LAdr, Ware, Preis)

Das Schema **Lieferant** hat folgende Nachteile:

- Redundanz
  - für jede Ware wird die Adresse des Lieferanten gespeichert, d.h. die Adresse ist mehrfach vorhanden
- Insert-/Delete-/Update-Anomalien
  - **update**: Adressänderung in 1 Tupel
  - **insert**: Einfügen eines Lieferanten erfordert Ware
  - **delete**: Löschen der letzten Ware löscht die Adresse

Verbesserung

Datenbankschema aus Kapitel 3:

Kunde	(KName, KAdr, Kto)
Auftrag	(KName, Ware, Menge)
LiefAdr	(LName, LAdr)
Angebot	(LName, Ware, Preis)

- Vorteile:
  - keine Redundanz
  - keine Anomalien
- Nachteil:
  - Um zu einer Ware die Adressen der Lieferanten zu finden, ist Join nötig (teuer auszuwerten und umständlich zu formulieren)

Ursprüngliche Relation

- Die ursprüngliche Relation Lieferant kann mit Hilfe einer View simuliert werden:

```
create view Lieferant as
    select   L.LName, LAdr, Ware, Preis
    from     LieferantAdr L, Angebot A
    where    L.LName = A.LName
```

Schema-Zerlegung

- Anomalien entstehen durch Redundanzen
- Entwurfsziele:
  - Vermeidung von Redundanzen
  - Vermeidung von Anomalien
  - evtl. Einbeziehung von Effizienzüberlegungen
- Vorgehen:
  Schrittweises Zerlegen des gegebenen Schemas (Normalisierung) in ein äquivalentes Schema ohne Redundanz und Anomalien
- Formalisierung von Redundanz und Anomalien:
  Funktionale Abhängigkeit

Funktionale Abhängigkeit

(engl. Functional Dependency, FD)

- beschreibt Beziehungen zwischen den Attributen einer Relation
- Schränkt das Auftreten gleicher bzw. ungleicher Attributwerte innerhalb einer Relation ein
  → spezielle Integritätsbedingung (nicht in SQL)

Wiederholung Integritätsbedingungen in SQL:

- Primärschlüssel
- Fremdschlüssel (referenzielle Integrität)
- **not null**
- **check**

Wiederholung *Schlüssel*

Definition:

- Eine Teilmenge *S* der Attribute eines Relationenschemas *R* heißt Schlüssel, wenn gilt:
  - Eindeutigkeit
    Keine Ausprägung von *R* kann zwei verschiedene Tupel enthalten, die sich in allen Attributen von *S* gleichen.
  - Minimalität
    Keine echte Teilmenge von *S* erfüllt bereits die Bedingung der Eindeutigkeit

- Ein Teilmenge S der Attribute von R heißt Superschlüssel, wenn nur die Eindeutigkeit gilt.

Definition: *funktional abhängig*

- Gegeben:
  - Ein Relationenschema R
  - A, B: Zwei *Mengen* von Attributen von R ($A,B \subseteq R$)
- Definition:
  B ist von A funktional abhängig (A $\rightarrow$ B) gdw.
  für alle möglichen Ausprägungen von R gilt:

$$falls \quad \forall r,s \in R \quad mit \quad r.A = s.A \quad gilt: r.B = s.B$$

  Zu jedem Wert in A exist. genau ein Wert von B.

- Beispiel **Lieferant** (LName, LAdr, Ware, Preis):
  - {LName} $\rightarrow$ {LAdr}
  - {LName, Ware} $\rightarrow$ {LAdr}
  - {LName, Ware} $\rightarrow$ {Preis}

üblicherweise schreibt man keine Klammern

Bei mehreren Attributen

- Steht auf der linken Seite mehr als ein Attribut:

$$A_1, A_2, \ldots, A_n \rightarrow B$$

dann gilt: *B* ist von der *Kombination* aus Attributen f.a.:

$$r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n \Rightarrow r.B = s.B$$

- Steht auf der rechten Seite mehr als ein Attribut:

$$A \rightarrow B_1, B_2, \ldots B_n$$

dann ist dies eine abkürzende Schreibweise für:

$$A \rightarrow B_1, A \rightarrow B_2, \ldots, A \rightarrow B_n$$

(wenn die Kombination von *B*-Werten von *A* f.a. ist, dann ist auch jeder einzelne *B*-Wert von *A* f.a. und umgekehrt)

Vergleich mit *Schlüssel*

- Gemeinsamkeiten zwischen dem *Schlüssel* im relationalen Modell und *Funktionaler Abhängigkeit*:
  - Definitionen ähnlich
  - Für alle Schlüsselkandidaten $S = \{A,B,...\}$ gilt:
    Alle Attribute der Rel. sind von S funktional abhängig:
    $$A, B, ... \rightarrow \mathrm{R}$$
    (das ist Folge der *Eindeutigkeits-Eigenschaft* von S)
- Unterschied:
  - Aber es gibt u.U. weitere funktionale Abhängigkeiten:
    Ein Attribut B kann z.B. auch funktional abhängig sein
    - von Nicht-Schlüssel-Attributen
    - von nur einem Teil des Schlüssels (nicht vom gesamten Schlüssel)

Vergleich mit *Schlüssel*

- Die funktionale Abhängigkeit ist also eine

  Verallgemeinerung des Schlüssel-Konzepts

- Wie der Schlüssel ist auch die funktionale Abhängigkeit eine semantische Eigenschaft des Schemas:
  - FD nicht aus aktueller DB-Ausprägung entscheidbar
  - sondern muss für alle möglichen Ausprägungen gelten

Triviale Funktionale Abhängigkeit

- Ein Attribut ist immer funktional abhängig:
  - von sich selbst
  - und von jeder Obermenge von sich selbst

Solche Abhängigkeiten bezeichnet man als

triviale funktionale Abhängigkeit

Partielle und *volle* FD

- Ist ein Attribut B funktional von A abhängig, dann auch von jeder Obermenge von A.
  Man ist interessiert, minimale Mengen zu finden, von denen B abhängt (vgl. Schlüsseldefinition)
- Definition:
  - Gegeben: Eine funktionale Abhängigkeit $A \rightarrow B$
  - Wenn es keine echte Teilmenge $A' \subset A$ gibt, von der B ebenfalls funktional abhängt,
  - dann heißt $A \rightarrow B$ eine volle funktionale Abhängigkeit
  - andernfalls eine partielle funktionale Abhängigkeit

(Anmerkung: Steht auf der linken Seite nur ein Attribut, so ist die FD immer eine volle FD)

Partielle und *volle* FD

- Beispiele:
  - LName → LAdr — voll funktional abhängig
  - LName, Ware → LAdr — partiell funktional abhängig
  - Ware ? Preis — nicht funktional abhängig
  - LName, Ware → Preis — voll funktional abhängig

Prime Attribute

- Definition:
  Ein Attribut heißt prim,
  wenn es Teil eines Schlüsselkandidaten ist

Volle FD und minimaler Schlüssel

- Aus der Eindeutigkeits-Eigenschaft ergibt sich, dass alle Attribute von einem Schlüssel *funktional abhängig* sind.
- Frage: Ergibt sich aus der Minimalität des Schlüssels auch, dass alle Attribute von S *voll funktional abhängig* sind?
- Dies würde nahe liegen, denn Definitionen sind ähnlich: „Es gibt keine echte Teilmenge, so dass …“
  - Eindeutigkeit erhalten bleibt (Minimalität Schlüssel)
  - Funktionale Abhängigkeit erhalten bleibt (volle FD)
- Trotzdem gilt: Einzelne Attribute können partiell von einem Schlüssel abhängig sein:
  - LName, Ware → LAdr: partiell funktional abhängig
- Das „Tupel als Ganzes“ ist aber vom Schlüssel *voll f.a.*

Herleitung funktionaler Abhängigkeit

Armstrong Axiome

- Reflexivität: Falls β eine Teilmenge von α ist ($\beta \subseteq \alpha$) dann gilt immer $\alpha \rightarrow \beta$. Inbesondere gilt also immer $\alpha \rightarrow \alpha$.
- Verstärkung: Falls $\alpha \rightarrow \beta$ gilt, dann gilt auch $\alpha\gamma \rightarrow \beta\gamma$. Hierbei steht $\alpha\gamma$ für $\alpha \cup \gamma$.
- Transitivität: Falls $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gilt, dann gilt auch $\alpha \rightarrow \gamma$.

Diese Axiome sind vollständig und korrekt :

Sei F eine Menge von FDs:

- es lassen sich nur FDs von F ableiten, die von jeder Relationenausprägung erfüllt werden, für die auch F erfüllt ist.
- alle FDs ableitbar, die durch F impliziert sind.

Hülle einer Attributmenge

- Eingabe: eine Menge F von FDs und eine Menge von Attributen α.
- Ausgabe: die vollständige Menge von Attributen α^+, für die gilt $\alpha \rightarrow \alpha^+$.

```
AttrHülle(F,α)
   Erg := α
   while( Änderungen an Erg) do
     foreach FD β→γ  in F do
         if β ⊆ Erg then Erg:=Erg ∪ γ
   Ausgabe α + = Erg
```

Verlustlose Zerlegung

- Eine Zerlegung von R in $R_1,\ldots, R_n$ ist *verlustlos*, falls sich jede mögliche Ausprägung r von R durch den natürlichen Join der Ausprägungen $r_1,\ldots,r_n$ rekonstruieren läßt:

$$r = r_1 \bowtie \ldots \bowtie r_n$$

- Beispiel für eine nicht-verlustlose Zerlegung:

  In der Relation *Einkauf* wird beschrieben, welche Waren ein Kunde (exklusiv) bei welchem Anbieter bezieht (d.h. es gelte *Kunde, Ware* $\rightarrow$ *Anbieter*):

Einkauf

Anbieter	Ware	Kunde
Meier	Eier	Schmidt
Meier	Milch	Huber
Bauer	Milch	Schmidt

Verlustlose Zerlegung

- Eine mögliche Zerlegung in die Relationen *Lieferant* und *Bedarf* ergibt:

Lieferant

Anbieter	**Kunde**
Meier	Schmidt
Meier	Huber
Bauer	Schmidt

Bedarf

Ware	**Kunde**
Eier	Schmidt
Milch	Huber
Milch	Schmidt

- Diese Zerlegung ist **nicht** verlustlos, da die Rekonstruktion von Einkauf als natürlicher Join von Lieferant und Bedarf misslingt, d.h.

$$Lieferant \bowtie Bedarf \neq Einkauf$$

Verlustlose Zerlegung

- Im konkreten Beispiel erhält man zusätzliche (unerwünschte) Tupel:

Lieferant ⋈ Bedarf

Anbieter	Ware	Kunde
Meier	Eier	Schmidt
Meier	*Milch*	*Schmidt*
Meier	Milch	Huber
Bauer	Milch	Schmidt
Bauer	*Eier*	*Schmidt*

Verlustlose Zerlegung

- Hinreichendes Kriterium für Verlustlosigkeit:

  Eine (binäre) Zerlegung von R mit den funktionalen Abhängigkeiten F in R_1 und R_2 ist verlustlos, wenn mindestens eine der folgenden funktionalen Abhängigkeiten auf der Basis von F herleitbar ist:

$$R_1 \cap R_2 \rightarrow R_1$$
$$R_1 \cap R_2 \rightarrow R_2$$

- Im Beispiel gilt nur die nicht-triviale Abhängigkeit

$$\text{Kunde, Ware} \rightarrow \text{Anbieter}$$

  nicht aber eine der beiden Abhängigkeiten, welche die Verlustlosigkeit garantieren würden:

$$\text{Kunde} \rightarrow \text{Anbieter}$$
$$\text{Kunde} \rightarrow \text{Ware}$$

Abhängigkeitserhaltende Zerlegung

- Eine Zerlegung von R in $R_1,...,R_n$ ist *abhängigkeitserhaltend*, wenn die Überprüfung aller funktionalen Abhängigkeiten F auf R lokal auf den R_i erfolgen kann, ohne dass Joins berechnet werden müssen.
- Es gibt dann keine übergreifenden Abhängigkeiten F' über die lokalen F_i hinaus und für die Menge der funktionalen Abhängigkeiten F auf R gilt:

$$F = F_1 \cup ... \cup F_n$$

Abhängigkeitserhaltende Zerlegung

- Beispiel: **Bank** (Filiale, Kunde, Betreuer)

  Funktionale Abhängigkeiten:

  Betreuer → Filiale

  Kunde, Filiale → Betreuer

- Mögliche Zerlegung :

  Personal (Filiale, Betreuer)

  Kunde (Kunde, Betreuer)

- Diese Zerlegung ist ...
  - *verlustlos* (d.h. *Personal* ⋈ *Kunden* = *Bank*), da Betreuer → Betreuer, Filiale gilt.
  - *nicht abhängigkeitserhaltend*, da Kunde, Filiale → Betreuer verlorengegangen ist.

Normalisierung

- In einem Relationenschema sollen möglichst keine funktionalen Abhängigkeiten bestehen, außer vom gesamten Schlüssel
- Verschiedene Normalformen beseitigen unterschiedliche Arten von funktionalen Abhängigkeiten bzw. Redundanzen/Anomalien
  - 1. Normalform
  - 2. Normalform
  - 3. Normalform
  - Boyce-Codd-Normalform
  - 4. Normalform
- Herstellung einer Normalform durch verlustlose Zerlegung des Relationenschemas

1. Normalform

- Keine Einschränkung bezüglich der FDs
- Ein Relationenschema ist in erster Normalform, wenn alle Attributwerte *atomar* sind
- In relationalen Datenbankem sind nicht-atomare Attribute ohnehin nicht möglich
- Nicht-atomare Attribute z.B. durch **group by**

A	B	C	D
1	2	3 4	4 5
2	3	3	4
3	3	4 6	5 7

„nested relation“
non first normal form
In SQL nur temporär erlaubt

2. Normalform

- Motivation:
  Man möchte verhindern, dass Attribute nicht vom gesamten Schlüssel voll funktional abhängig sind, sondern nur von einem Teil davon.
- Beispiel:

Lieferant (LName, LAdr, Ware, Preis)

Bäcker	Ibk	Brot	3,00
Bäcker	Ibk	Semmel	0,30
Bäcker	Ibk	Breze	0,40
Metzger	Hall	Filet	5,00
Metzger	Hall	Wurst	4,00

Anomalien?

- Konsequenz: In den abhängigen Attributen muss dieselbe Information immer wiederholt werden

2. Normalform

- Dies fordert man vorerst nur für Nicht-Schlüssel-Attribute (für die anderen z.T. schwieriger)
- Definition
  Ein Schema ist in zweiter Normalform, wenn jedes Attribut
  - voll funktional abhängig von allen Schlüsselkandidaten
  - oder prim ist
- Beobachtung:
  Zweite Normalform kann nur verletzt sein, wenn...
  ...ein Schlüssel(-Kandidat) zusammengesetzt ist

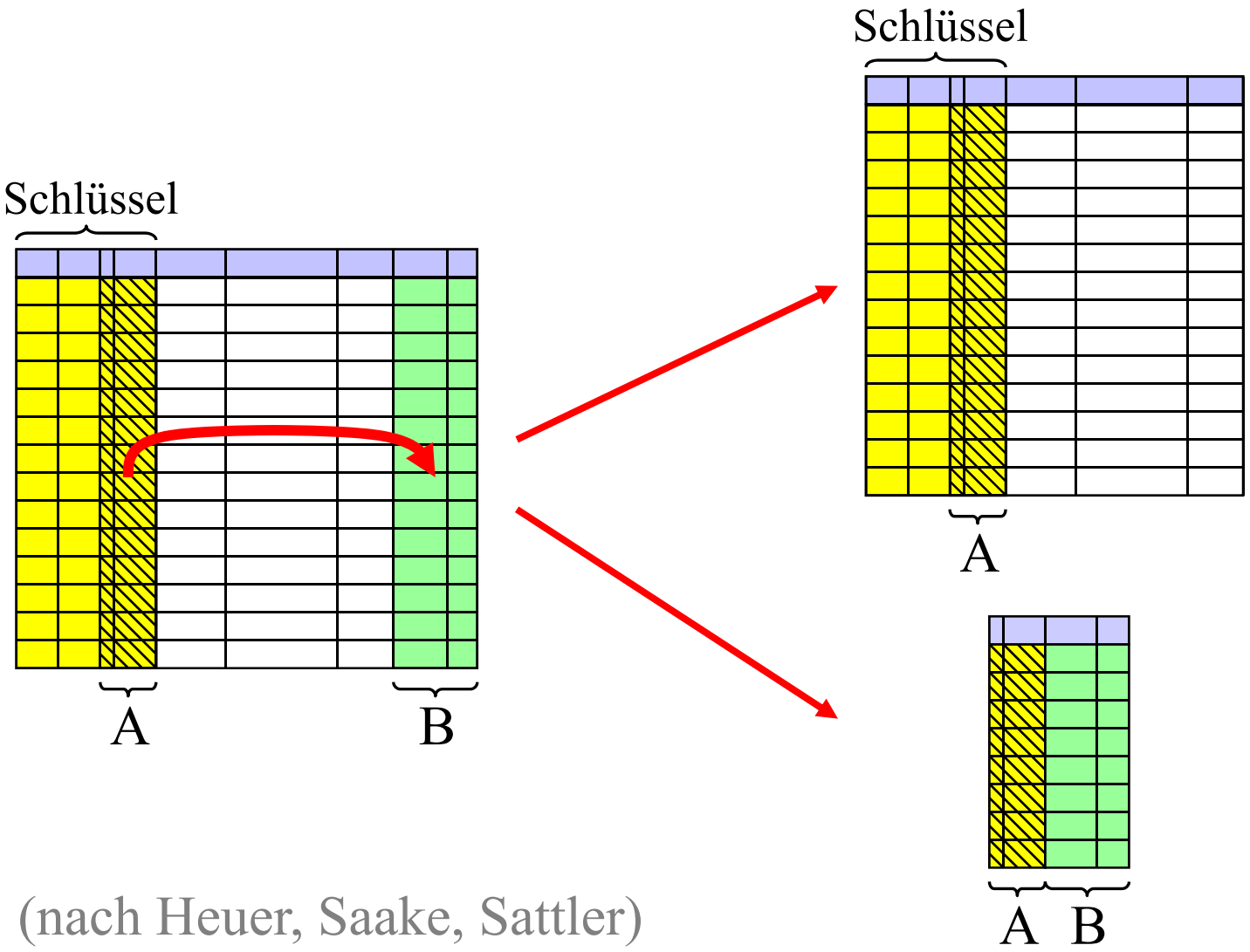
2. Normalform

- Zur Transformation in 2. Normalform spaltet man das Relationenschema auf:
  - Attribute, die voll funktional abhängig vom Schlüssel sind, bleiben in der Ursprungsrelation R
  - Für alle Abhängigkeiten $A_i \rightarrow B_i$ von einem Teil eines Schlüssels ($A_i \subset S$) geht man folgendermaßen vor:
    - Lösche die Attribute B_i aus R
    - Gruppiere die Abhängigkeiten nach gleichen linken Seiten A_i
    - Für jede Gruppe führe eine neue Relation ein mit allen enthaltenen Attributen aus A_i und B_i
    - A_i wird Schlüssel in der neuen Relation

2. Normalform

- Beispiel:
  Lieferant (LName, LAdr, Ware, Preis)
- Vorgehen:
  - LAdr wird aus Lieferant gelöscht
  - Gruppierung:
    Nur eine Gruppe mit LName auf der linken Seite
    - es könnten aber noch weitere Attribute von LName abhängig sein (selbe Gruppe)
    - es könnten Attribute von Ware abh. (2. Gruppe)
  - Erzeugen einer Relation mit LName und LAdr
- Ergebnis: **Lieferant** (LName, Ware, Preis)
  LieferAdr (LName, LAdr)

Grafische Darstellung



(nach Heuer, Saake, Sattler)

3. Normalform

- Motivation:
  Man möchte zusätzlich verhindern, dass Attribute von nicht-primen Attributen funktional abhängen.
- Beispiel:
  Bestellung (AuftrNr, Datum, KName, KAdresse)

001	24.04.02	Meier	Innsbruck
002	25.04.02	Meier	Innsbruck
003	25.04.02	Huber	Hall
004	25.04.02	Huber	Hall
005	26.04.02	Huber	Hall

- Redundanz: Kundenadresse mehrfach gespeichert
- Anomalien?

3. Normalform

- Abhängigkeit von Nicht-Schlüssel-Attribut bezeichnet man häufig auch als *transitive Abhängigkeit* vom Primärschlüssel
  - weil Abhängigkeit *über* ein drittes Attribut besteht:

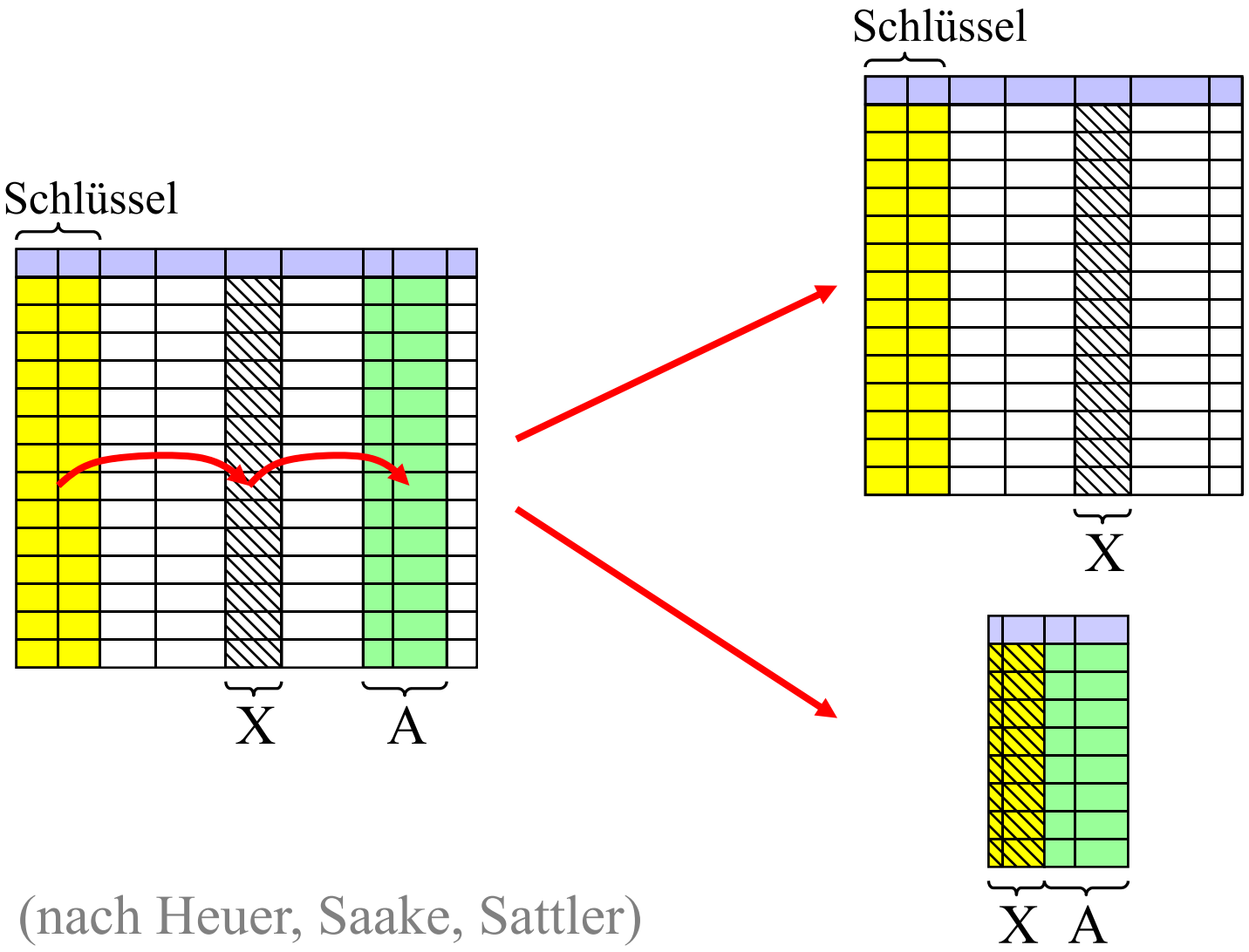
    $\underline{\text{AuftrNr}} \rightarrow \text{KName} \rightarrow \text{KAdr}$

- Definition:
  Ein Relationenschema ist in 3. Normalform, wenn für jede nichttriviale Abhängigkeit $X \rightarrow A$ gilt:
  - X enthält einen Schlüsselkandidaten
  - oder A ist prim
- Beobachtung: 2. Normalform ist mit impliziert

3. Normalform

- Transformation in 3. Normalform wie vorher
  - Attribute, die voll funktional abhängig vom Schlüssel sind, und nicht abhängig von Nicht-Schlüssel-Attributen sind, bleiben in der Ursprungsrelation R
  - Für alle Abhängigkeiten $A_i \rightarrow B_i$ von einem Teil eines Schlüssels ($A_i \subset S$) oder von Nicht-Schlüssel-Attribut:
    - Lösche die Attribute B_i aus R
    - Gruppiere die Abhängigkeiten nach gleichen linken Seiten A_i
    - Für jede Gruppe führe eine neue Relation ein mit allen enthaltenen Attributen aus A_i und B_i
    - A_i wird Schlüssel in der neuen Relation

Grafische Darstellung



(nach Heuer, Saake, Sattler)

Synthesealgorithmus für 3NF

Synthesealgorithmus für 3NF

- Der sogenannte *Synthesealgorithmus* ermittelt zu einem gegebenen Relationenschema R mit funktionalen Abhängigkeiten F eine Zerlegung in Relationen $R_1, \ldots, R_n$, die folgende Kriterien erfüllt:
- $R_1, \ldots, R_n$ ist eine verlustlose Zerlegung von R.
- Die Zerlegung ist abhängigkeitserhaltend.
- Alle R_i $(1 \leq i \leq n)$ sind in dritter Normalform.

Synthesealgorithmus für 3NF

Der Synthese-Algorithmus arbeitet in 4 Schritten:

1. Bestimme die kanonische Überdeckung F_c zu F, d.h. eine minimale Menge von FDs, die dieselben (partiellen und transitiven) Abhängigkeiten wie F beschreiben
2. Erzeugung eines neuen Relationenschemas aus F_c
3. Rekonstruktion eines Schlüsselkandidaten
4. Elimination überflüssiger Relationen

Synthesealgorithmus für 3NF

Bestimmung der kanonischen Überdeckung der Menge der funktionalen Abhängigkeiten:

$AB \rightarrow C$
$A \rightarrow C$
$\Downarrow$
$A \rightarrow C$

1. **Linksreduktion** der FDs $A \rightarrow B$, um *partielle* Abhängigkeiten zu entfernen:
   Für jedes $\alpha \in A$, ersetze die Abhängigkeit $A \rightarrow B$ durch $(A-\alpha) \rightarrow B$, falls α auf der linken Seite überflüssig ist, d.h. falls *B* schon durch $(A-\alpha)$ determiniert ist.

$A \rightarrow B \rightarrow C$
$A \rightarrow C$
$\Downarrow$
$A \rightarrow B \rightarrow C$

2. **Rechtsreduktion** der (verbliebenen) FDs $A \rightarrow B$ zur Entfernung *transitiver* Abhängigkeiten:
   Für jedes $\beta \in B$, ersetze die Abhängigkeit $A \rightarrow B$ durch $A \rightarrow (B-\beta)$, falls β auf der rechten Seite überflüssig ist, d.h. falls $A \rightarrow \beta$ eine transitive Abhängigkeit ist.
3. **Entfernung** von rechts-leeren funktionalen Abhängigkeiten $A \rightarrow \varnothing$, die bei der Rechtsreduktion möglicherweise entstanden sind.
4. **Zusammenfassen** von Abhängigkeiten mit gleichen linken Seiten, so daß jede linke Seite nur einmal vorkommt:
   Ersetze die Abhängigkeiten $A \rightarrow B_1, \ldots, A \rightarrow B_m$ durch $A \rightarrow (B_1 \cup \ldots \cup B_m)$.

Synthesealgorithmus für 3NF

2. Erzeugung eines neuen Relationenschemas aus F_c:
   - Erzeuge ein Relationenschema $R_A = (A \cup B)$
   - Ordne dem Schema R_A die FDs $F_A = \{(A' \rightarrow B') \in F_c \mid A' \cup B' \subseteq R_A\}$ zu.
3. Rekonstruktion eines Schlüsselkandidaten:
   Falls eines der in Schritt 2. erzeugten Schemata R_A einen Schlüsselkandidaten von R enthält, sind wir fertig. Ansonsten wähle einen Schlüsselkandidaten $\kappa \in R$ aus und erzeuge das zusätzliche Schema $R_A = \kappa$ mit $F_A = \emptyset$.
4. Elimination überflüssiger Relationen:
   - Eliminiere diejenigen Schemata $R_{A'}$ die in einem anderen Schema $R_{A'}$ enthalten sind: $R_A \subseteq R_{A'}$

Synthesealgorithmus für 3NF

Beispiel:

Einkauf (Anbieter, Ware, WGruppe, Kunde, KOrt, KLand, Kaufdatum)

Schritte des Synthesealgorithmus:

1. Kanonische Überdeckung F_c der funktionalen Abhängigkeiten:
   Kunde, WGruppe → Anbieter
   Anbieter → WGruppe
   Ware → WGruppe
   Kunde → KOrt
   KOrt → KLand
2. Erzeugen der neuen Relationenschemata und ihrer FDs:

Bezugsquelle (Kunde, WGruppe, Anbieter)	{Kunde,WGruppe → Anbieter, Anbieter → WGruppe}
Lieferant (Anbieter, WGruppe)	{Anbieter → WGruppe}
Produkt (Ware, WGruppe)	{Ware → WGruppe}
Adresse (Kunde, KOrt)	{Kunde → KOrt}
Land (KOrt, KLand)	{KOrt → KLand}

3. Da keine dieser Relationen einen Schlüsselkandidaten der ursprünglichen Relation enthält, muß noch eine eigene Relation mit dem ursprünglichen Schlüssel angelegt werden:
   Einkauf (Ware, Kunde, Kaufdatum)
4. Da die Relation *Lieferant* in *Bezugsquelle* enthalten ist, können wir *Lieferant* wieder streichen.

Boyce-Codd-Normalform

- Welche Abhängigkeiten können in der dritten Normalform noch auftreten?

  Abhängigkeiten unter Attributen, die prim sind,
  aber noch nicht vollständig einen Schlüssel bilden

- Beispiel:
  Autoverzeichnis (Hersteller, HerstellerNr, ModellNr)
  - es gilt 1:1-Beziehung zw. Hersteller und HerstellerNr:
    Hersteller → HerstellerNr
    HerstellerNr → Hersteller
  - Schlüsselkandidaten sind deshalb:
    {Hersteller, ModellNr}
    {HerstellerNr, ModellNr}
- Schema in 3. NF, da alle Attribute prim sind.

Boyce-Codd-Normalform

- Trotzdem können auch hier Anomalien auftreten

- Definition:
  Ein Schema R ist in Boyce-Codd-Normalform, wenn für alle nichttrivialen Abhängigkeiten $X \rightarrow A$ gilt: X enthält einen Schlüsselkandidaten von R

- Die Zerlegung ist teilweise schwieriger.
- Man muß auf die Verlustlosigkeit achten.
- Verlustlose Zerlegung nicht immer möglich.

Mehrwertige Abhängigkeiten

- Mehrwertige Abhängigkeiten entstehen, wenn mehrere unabhängige 1:*n*-Beziehungen in einer Relation stehen (was nach Kapitel 6 eigentlich nicht sein darf):
- Mitarbeiter (Name, Projekte, Verwandte)

Name,	Projekte,	Verwandte)
Huber,	{P1, P2, P3}	{Heinz, Hans, Hubert}
Müller,	{P2, P3}	{Manfred}

- In erster Normalform müsste man mindestens 3 Tupel für Huber und 2 Tupel für Müller speichern:
- Mitarbeiter (Name, Projekte, Verwandte)

Name,	Projekte,	Verwandte)
Huber,	P1,	Heinz,
Huber,	P2,	Hans,
Huber,	P3,	Hubert,
Müller,	P2,	Manfred
Müller,	P3,	NULL

Mehrwertige Abhängigkeiten

- Um die Anfrage zu ermöglichen, wer sind die Verwandten von Mitarbeitern in Projekt P2 müssen pro Mitarbeiter sogar sämtliche Kombinationstupel gespeichert werden:

Mitarbeiter (Name, Projekte, Verwandte)

Name	Projekte	Verwandte
Huber,	P1,	Heinz,
Huber,	P1,	Hans,
Huber,	P1,	Hubert,
Huber,	P2,	Heinz,
Huber,	P2,	Hans,
Huber,	P2,	Hubert,
Huber,	P3,	Heinz,
Huber,	P3,	Hans,
Huber,	P3,	Hubert,
Müller,	P2,	Manfred,
Müller,	P3,	Manfred.

- Wir nennen dies eine Mehrwertige Abhängigkeit zwischen Name und Projekte (auch zwischen Name und Verwandte)

Mehrwertige Abhängigkeiten (MVD)

Geg: $\alpha,\beta,\gamma \subseteq R$, mit $R= \alpha \cup \beta \cup \gamma$

β ist ***mehrwertig abhängig*** von α ($\beta \rightarrow\rightarrow \alpha$), wenn für jede gültige Ausprägung von R gilt: Für jedes Paar aus Tupeln t_1, t_2 mit $t_1.\alpha = t_2.\alpha$, aber (natürlich) $t_1.\beta\gamma <> t_2.\beta\gamma$ existieren 2 weitere Tupel t_3 und t_4 mit folgenden Eigenschaften:

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

Jede FD ist auch eine MVD !!!

Beispiel MVD

	R		
	α $\overbrace{A_1 \ldots A_i}$	β $\overbrace{A_{1+1} \ldots A_j}$	γ $\overbrace{A_{j+1} \ldots A_n}$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$
t_4	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$

Weiteres Beispiel

Relation: Modelle

ModellNr	Farbe	Leistung
E36	blau	170 PS
E36	schwarz	170 PS
E36	blau	198 PS
E36	schwarz	198 PS
E34	schwarz	170 PS

$\{ModellNr\} \rightarrow\rightarrow \{Farbe\}$ und $\{ModellNr\} \rightarrow\rightarrow \{Leistung\}$

Farben

ModellNr	Farbe
E36	blau
E36	schwarz
E34	Schwarz

Leistung

ModellNr	Leistung
E36	170 PS
E36	198 PS
E34	170 PS

$$Modelle = \Pi_{ModellNr,Sprache}(Farben) \bowtie \Pi_{ModellNr,Leistung}(Leistung)$$

Verlustlose Zerlegung MVD

Ein Relationenschema R mit einer Menge D von zugeordneten funktionalen mehrwertigen Abhängigkeiten kann genau dann verlustlos in die beiden Schemata R_1 und R_2 zerlegt werden wenn gilt:

- $R = R_1 \cup R_2$
- mindestens eine von zwei MVDs gilt:
  1. $R_1 \cap R_2 \rightarrow\rightarrow R_1$ oder
  2. $R_1 \cap R_2 \rightarrow\rightarrow R_2$

Triviale MVD und 4. Normalform

Eine MVD $\alpha \rightarrow\rightarrow \beta$ bezogen auf $R \supseteq \alpha \cup \beta$ ist *trivial*, wenn jede mögliche Ausprägung *r* von R diese MVD erfüllt. Man kann zeigen, daß $\alpha \rightarrow\rightarrow \beta$ trivial ist, genau dann wenn:

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

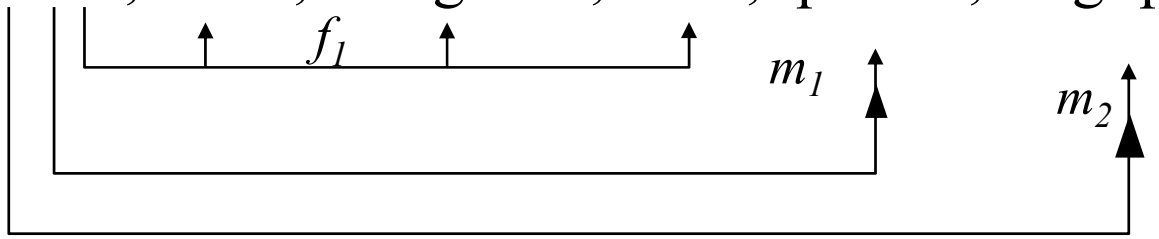
Eine Relation R mit zugeordneter Menge D von funktionalen und mehrwertigen Abhängigkeiten in ***4NF***, wenn für jede MVD $\alpha \rightarrow\rightarrow \beta \in D^+$ eine der folgenden Bedingungen gilt:

1. Die MVD ist trivial oder
2. α ist ein Superschlüssel von R.

Beispiel

Assistenten:

{[PersNr,Name,Fachgebiet, Boss,Sprache,ProgSprache]}



- Assistenten: {[PersNr,Name,Fachgebiet, Boss]}
- Sprachen:{[PersNr, Sprache]}
- ProgSprach:{[PersNr,ProgSprache}

Schlussbemerkungen

- Ein gut durchdachtes E/R-Diagramm liefert bereits weitgehend normalisierte Tabellen
- Normalisierung ist in gewisser Weise eine Alternative zum E/R-Diagramm
- Extrem-Ansatz: Universal Relation Assumption:
  - Modelliere alles zunächst in einer Tabelle
  - Ermittle die funktionalen Abhängigkeiten
  - Zerlege das Relationenschema entsprechend (der letzte Schritt kann auch automatisiert werden: Synthesealgorithmus für die 3. Normalform)

Schlussbemerkungen

- Normalisierung kann schädlich für die Performanz sein, weil Joins sehr teuer auszuwerten sind
- Nicht *jede* FD berücksichtigen:
  - Abhängigkeiten zw. Wohnort, Vorwahl, Postleitzahl
  - Man kann SQL-Integritätsbedingungen formulieren, um Anomalien zu vermeiden (Trigger, siehe später)
- Aber es gibt auch Konzepte, Relationen so abzuspeichern, dass Join auf bestimmten Attributen unterstützt wird
  - ORACLE-Cluster

Zusammenfassung

Implikation

- 1. Normalform:
  Alle Attribute atomar
- 2. Normalform:
  Keine funktionale Abhängigkeit eines Nicht-Schlüssel-Attributs von Teil eines Schlüssels
- 3. Normalform:
  Zusätzlich keine nichttriviale funktionale Abhängigkeit eines Nicht-Schlüssel-Attributs von Nicht-Schlüssel-Attributen
- Boyce-Codd-Normalform:
  Zusätzlich keine nichttriviale funktionale Abhängigkeit unter den Schlüssel-Attributen
- 4. Normalform:
  keine Redundanz durch MVDs.

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

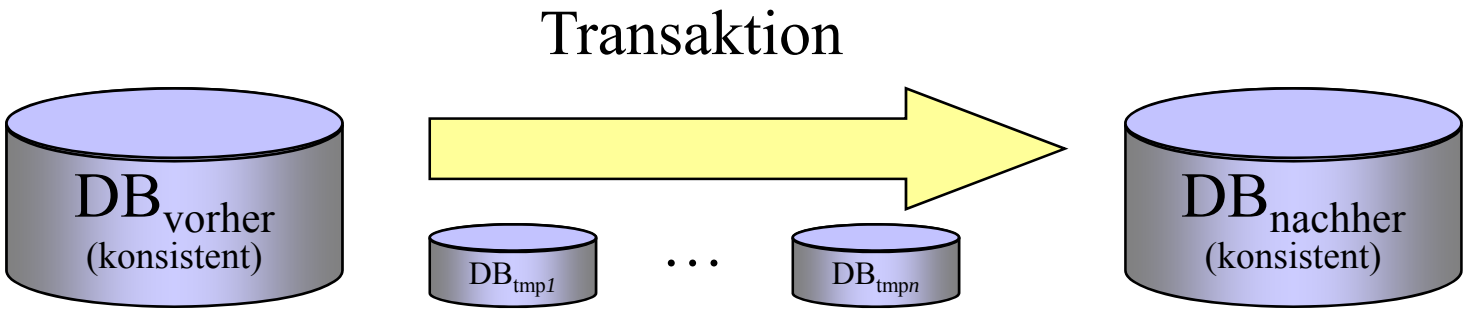
Kapitel 8: Transaktionen

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Transaktionskonzept

- Transaktion: Folge von Befehlen (*read*, *write*), die die DB von einen konsistenten Zustand in einen anderen konsistenten Zustand überführt
- Transaktionen: Einheiten integritätserhaltender Zustandsänderungen einer Datenbank
- Hauptaufgaben der Transaktions-Verwaltung
  - Synchronisation (Koordination mehrerer Benutzerprozesse)
  - Recovery (Behebung von Fehlersituationen)



Transaktionskonzept

Beispiel Bankwesen:

Überweisung von Huber an Meier in Höhe von 200 €

- Mgl. Bearbeitungsplan:
  (1) Erniedrige Stand von Huber um 200 €
  (2) Erhöhe Stand von Meier um200 €
- Möglicher Ablauf

Konto	Kunde	Stand
	Meier	1.000 €
	Huber	1.500 €

(1) →

Konto	Kunde	Stand
	Meier	1.000 €
	Huber	1.300 €

(2) → **System-absturz**

Inkonsistenter DB-Zustand darf nicht entstehen bzw. darf nicht dauerhaft bestehen bleiben!

Eigenschaften von Transaktionen

- **ACID-Prinzip**
  - **Atomicity** (Atomarität)
    Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zum Tragen.
  - **Consistency** (Konsistenz, Integritätserhaltung)
    Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt.
  - **Isolation** (Isoliertheit, 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.
- Weitere Forderung: TA muss in endlicher Zeit bearbeitet werden können

Steuerung von Transaktionen

- **begin of transaction (BOT)**
  - markiert den Anfang einer Transaktion
  - Transaktionen werden implizit begonnen, es gibt kein `begin work` o.ä.
- **end of transaction (EOT)**
  - markiert das Ende einer Transaktion
  - alle Änderungen seit dem letzten BOT werden festgeschrieben
  - SQL: `commit oder commit work`
- **abort**
  - markiert den Abbruch einer Transaktion
  - die Datenbasis wird in den Zustand vor BOT zurückgeführt
  - SQL: `rollback oder rollback work`
- **Beispiel**

```
UPDATE Konto SET Stand = Stand-200 WHERE Kunde = ‘Huber’;
UPDATE Konto SET Stand = Stand+200 WHERE Kunde = ‘Meier’;
COMMIT;
```

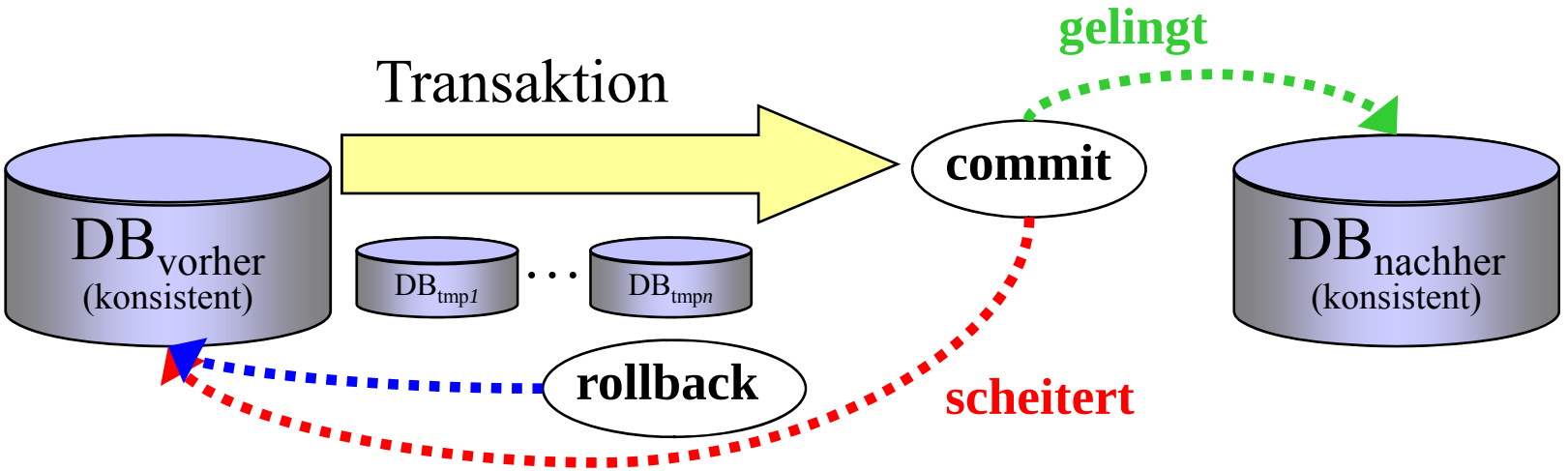
Steuerung von Transaktionen

Unterstützung langer Transaktionen durch

- **define savepoint**
  - markiert einen zusätzlichen Sicherungspunkt, auf den sich die noch aktive Transaktion zurücksetzen lässt
  - Änderungen dürfen noch nicht festgeschrieben werden, da die Transaktion noch scheitern bzw. zurückgesetzt werden kann
  - SQL: `savepoint <identifier>`
- **backup transaction**
  - setzt die Datenbasis auf einen definierten Sicherungspunkt zurück
  - SQL: `rollback to <identifier>`

Ende von Transaktionen

- COMMIT **gelingt**
  → der neue Zustand wird dauerhaft gespeichert.
- COMMIT **scheitert**
  → der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann z.B. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.
- ROLLBACK
  → Benutzer widerruft Änderungen



Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Klassifikation von Fehlern

- **Transaktionsfehler**
  Lokaler Fehler einer noch nicht festgeschriebenen Transaktion, z.B. durch
  - Fehler im Anwendungsprogramm
  - Expliziter Abbruch der Transaktion durch den Benutzer (`ROLLBACK`)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - Konflikte mit nebenläufigen Transaktionen (Deadlock)

Klassifikation von Fehlern

- **Systemfehler**
  Fehler mit Hauptspeicherverlust, d.h. permanente Speicher sind *nicht* betroffen, z.B. durch
  - Stromausfall
  - Ausfall der CPU
  - Absturz des Betriebssystems, …
- **Medienfehler**
  Fehler mit Hintergrundspeicherverlust, d.h. Verlust von permanenten Daten, z.B. durch
  - Plattencrash
  - Brand, Wasserschaden, …
  - Fehler in Systemprogrammen, die zu einem Datenverlust führen

Recovery-Techniken

- **Rücksetzen** (bei Transaktionsfehler)
  - *Lokales UNDO*: der ursprüngliche DB-Zustand wie zu BOT wird wiederhergestellt, d.h. Rücksetzen aller Aktionen, die diese Transaktion ausgeführt hat
  - Transaktionsfehler treten relativ häufig auf
    → Behebung innerhalb von Millisekunden notwendig

Recovery-Techniken

- **Warmstart** (bei Systemfehler)
  - *Globales UNDO*: Rücksetzen aller noch nicht abgeschlossenen Transaktionen, die bereits in die DB eingebracht wurden
  - *Globales REDO*: Nachführen aller bereits abgeschlossenen Transaktionen, die noch nicht in die DB eingebracht wurden
  - Dazu sind Zusatzinformationen aus einer Log-Datei notwendig, in der die laufenden Aktionen, Beginn und Ende von Transaktionen protokolliert werden
  - Systemfehler treten i.d.R. im Intervall von Tagen auf → Recoverydauer einige Minuten

Recovery-Techniken

- **Kaltstart** (bei Medienfehler)
  - Aufsetzen auf einem früheren, gesicherten DB-Zustand (Archivkopie)
  - *Globales REDO:* Nachführen aller Transaktionen, die nach dem Erzeugen der Sicherheitskopie abgeschlossenen wurden
  - Medienfehler treten eher selten auf (mehrere Jahre) → Recoverydauer einige Stunden / Tage
  - Wichtig: regelmäßige Sicherungskopien der DB notwendig

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Datenintegrität

- **Beispiel**

  Kunde(KName, KAdr, Konto)
  Auftrag(KName, Ware, Menge)
  Lieferant(LName, LAdr, Ware, Preis)

- **Mögliche Integritätsbedingungen**
  - Kein Kundenname darf mehrmals in der Relation „Kunde“ vorkommen.
  - Jeder Kundenname in „Auftrag“ muss auch in „Kunde“ vorkommen.
  - Kein Kontostand darf unter -100 € sinken.
  - Das Konto des Kunden Huber darf überhaupt nicht überzogen werden.
  - Es dürfen nur solche Waren bestellt werden, für die es mindestens einen Lieferanten gibt.
  - Der Brotpreis darf nicht erhöht werden.

Statische vs. dynamische Integrität

- **Statische Integritätsbedingungen**
  - Einschränkung der möglichen Datenbankzustände
  - z.B. „Kein Kontostand darf unter -100 € sinken.“
  - In SQL durch Constraint-Anweisungen implementiert (`UNIQUE, DEFAULT, CHECK, ….`)
  - Im Bsp.:

```
create table Kunde(
   kname varchar(40) primary key,
   kadr varchar(100),
   konto float check konto > -100,
   );
```

Statische vs. dynamische Integrität

- **Dynamische Integritätsbedingungen**
  - Einschränkung der möglichen Zustandsübergänge
  - z.B. „Der Brotpreis darf nicht erhöht werden.“
  - In Oracle durch sog. Trigger implementiert
    - PL/SQL-Programm, das einer Tabelle zugeordnet ist und durch ein best. Ereignis ausgelöst wird
    - Testet die mögliche Verletzung einer Integritätsbedingung und veranlasst daraufhin eine bestimmte Aktion
    - Mögliche Ereignisse: insert, update oder delete
    - **Befehls-Trigger** (statement-trigger): werden einmal pro auslösendem Befehl ausgeführt
    - **Datensatz-Trigger** (row-trigger): werden einmal pro geändertem / eingefügtem / gelöschtem Datensatz ausgeführt
    - Mögliche Zeitpunkte: vor (BEFORE) oder nach (AFTER) dem auslösenden Befehl oder alternativ dazu (INSTEAD OF)

Statische vs. dynamische Integrität

- Datensatz-Trigger haben Zugriff auf Instanzen *vor* und *nach* dem Ereignis mit
  - `:new.` bzw. `:old.` (PL/SQL-Block)
  - `new.` bzw. `old.` (Trigger-Restriktion)
- **Im Bsp.:**

```
create trigger keineErhoehung
before update of Preis on Lieferant
for each row
when (old.Ware = „Brot“)
begin
    if (:old.Preis < :new.Preis) then
        :new.Preis = :old.Preis;
    endif;
end
```

Modellinhärente Integrität

Durch das Datenmodell vorgegebene Integritätsbedingungen

- **Typintegrität**
  Beschränkung der zulässigen Werte eines Attributs durch dessen Wertebereich
- **Schlüsselintegrität**
  DB darf keine zwei Tupel mit gleichem Primärschlüssel enthalten, z.B. „Kein Kundenname darf mehrmals in der Relation Kunde vorkommen.“.
- **Referentielle Integrität** (Fremdschlüsselintegrität)
  Wenn Relation R einen Schlüssel von Relation S enthält, dann muss für jedes Tupel in R auch ein entsprechendes Tupel in S vorkommen, z.B. „Jeder Kundenname in Auftrag muss auch in Kunde vorkommen.“.

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Synchronisation (Concurrency Control)

- **Serielle Ausführung** von Transaktionen ist unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist (niedriger Durchsatz, hohe Wartezeiten)
- **Mehrbenutzerbetrieb** führt i.a. zu einer besseren Auslastung des Systems (z.B. Wartezeiten bei E/A-Vorgängen können zur Bearbeitung anderer Transaktionen genutzt werden)
- **Aufgabe der Synchronisation**
  Gewährleistung des logischen Einbenutzerbetriebs, d.h. innerhalb einer TA ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen

Anomalien bei unkontrolliertem Mehrbenutzerbetrieb

- Verloren gegangene Änderungen (*Lost Updates*)
- Zugriff auf „schmutzige“ Daten (*Dirty Read / Dirty Write*)
- Nicht-reproduzierbares Lesen (*Non-Repeatable Read*)
- Phantomproblem
- **Beispiel:** Flugdatenbank

Passagiere	FlugNr	Name	Platz	Gepäck
	LH745	Müller	3A	8
	LH745	Meier	6D	12
	LH745	Huber	5C	14
	BA932	Schmidt	9F	9
	BA932	Huber	5C	14

Lost Updates

- Änderungen einer Transaktion können durch Änderungen anderer Transaktionen überschrieben werden und dadurch verloren gehen
- Bsp.: Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus
  - T1:
```
UPDATE Passagiere SET Gepäck = Gepäck+3
WHERE FlugNr = LH745 AND Name = „Meier“;
```
  - T2:
```
UPDATE Passagiere SET Gepäck = Gepäck+5
WHERE FlugNr = LH745 AND Name = „Meier“;
```
- Mgl. Ablauf:

T1	T2
`read(Passagiere.Gepäck, x1);`	`read(Passagiere.Gepäck, x2);`
	`x2 := x2 + 5;`
	`write(Passagiere.Gepäck, x2);`
`x1 := x1+3;`	
`write(Passagiere.Gepäck, x1);`	

- In der DB ist nur die Änderung von T1 wirksam, die Änderung von T2 ist verloren gegangen → Verstoß gegen *Durability*

Dirty Read / Dirty Write

- Zugriff auf „schmutzige“ Daten, d.h. auf Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden
- Bsp.:
  - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
  - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen
- Mgl. Ablauf:

T1	T2

`UPDATE Passagiere`
`SET Gepäck = Gepäck+3;`
 	
`ROLLBACK;`
	
	`UPDATE Passagiere`
	`SET Gepäck = Gepäck+5;`
	`COMMIT;`

- Durch den Abbruch von T1 werden die geänderten Werte ungültig. T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*)
- Verstoß gegen ACID: Dieser Ablauf verursacht einen inkonsistenten DB-Zustand (*Consistency*) bzw. T2 muss zurückgesetzt werden (*Durability*).

Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Bsp.:
  - T1 liest das Gepäckgewicht der Passagiere auf Flug BA932 zwei mal
  - T2 bucht den Platz 3F auf dem Flug BA932 für Passagier Meier mit 5kg Gepäck
- Mgl. Ablauf

T1	T2

`SELECT Gepäck FROM Passagiere`
`WHERE FlugNr = „BA932“;`
 	
`SELECT Gepäck FROM Passagiere`
`WHERE FlugNr = „BA932“;`
	`INSERT INTO Passagiere`
	`VALUES (BA932, Meier, 3F, 5);`
	`COMMIT;`

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl die T1 den DB-Zustand nicht geändert hat → Verstoß gegen *Isolation*

Phantomproblem

- Ausprägung des nicht-reproduzierbaren Lesen, bei der Aggregatfunktionen beteiligt sind
- Bsp.:
  - T1 druckt die Passagierliste sowie die Anzahl der Passagiere für den Flug LH745
  - T2 bucht den Platz 7D auf dem Flug LH745 für Phantomas
- Mgl. Ablauf

T1	T2

`SELECT * FROM Passagiere`
`WHERE FlugNr = „LH745“;`
 	
`SELECT COUNT(*) FROM Passagiere`
`WHERE FlugNr = „ LH745“;`
	`INSERT INTO Passagiere`
	`VALUES (LH745, Phantomas, 7D, 2);`
	`COMMIT;`

- Für Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere berücksichtigt ist

Serialisierbarkeit von Transaktionen

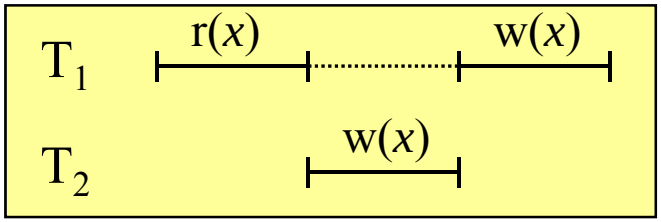
- Die nebenläufige Bearbeitung von Transaktionen geschieht für den Benutzer transparent, d.h. als ob die Transaktionen (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden
- **Begriffe**
  - Ein *Schedule* für eine Menge $\{T_1, \ldots, T_n\}$ von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der T_is 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.
  - Ein Schedule S von $\{T_1, \ldots, T_n\}$ ist *serialisierbar*, wenn er dieselbe Wirkung hat wie ein beliebiger serieller Schedule von $\{T_1, \ldots, T_n\}$.

Nur serialisierbare Schedules dürfen zugelassen werden!

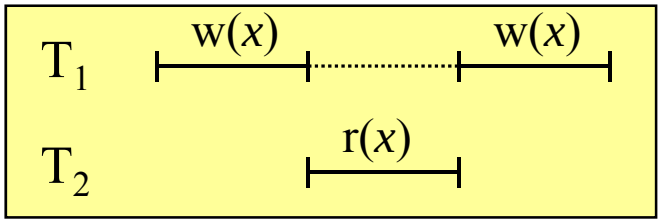
Serialisierbarkeit von Transaktionen

Beispiele für nicht-serialisierbare Schedules

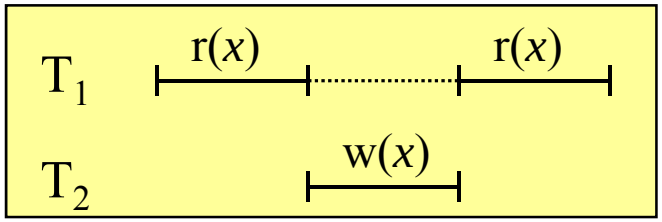
- Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



- Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$



- Non-repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$



Kriterium für Serialisierbarkeit

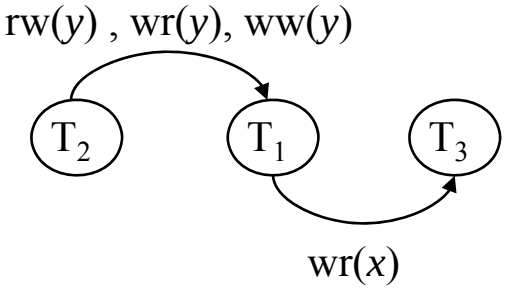
Mit Hilfe von Serialisierungsgraphen kann man prüfen, ob ein Schedule $\{T_1, \ldots, T_n\}$ serialisierbar ist

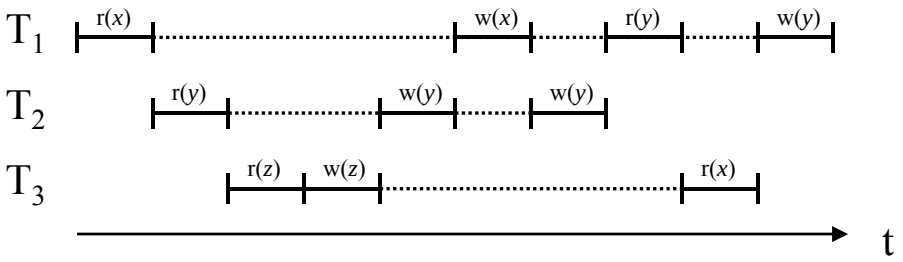
- Die beteiligten Transaktionen $\{T_1, \ldots, T_n\}$ sind die *Knoten* des Graphen
- Die *Kanten* beschreiben die Abhängigkeiten der Transaktionen:
  Eine Kante $T_i \rightarrow T_j$ wird eingetragen, falls im Schedule
  - $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr(x)$
  - $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw(x)$
  - $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww(x)$
- Ein Schedule ist serialisierbar, falls der Serialisierungs-graph *zyklenfrei* ist
- Einen zugehörigen seriellen Schedule erhält man durch topologisches Sortieren des Graphen

Kriterium für Serialisierbarkeit

Beispiele

- $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$

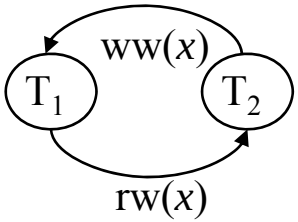




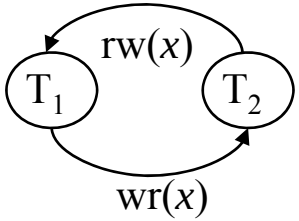
Serialisierungsreihenfolge: (T_2, T_1, T_3)

- Nicht-serialisierbare Schedules

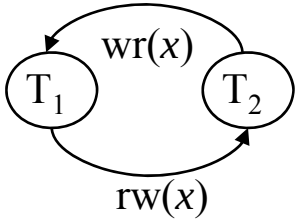
$S=(r_1(x), w_2(x), w_1(x))$
Lost Update



$S=(w_1(x), r_2(x), w_1(x))$
Dirty Read



$S=(r_1(x), w_2(x), r_1(x))$
Non-Repeatable Read



Techniken zur Synchronisation

- Verwaltungsaufwand für Serialisierungsgraphen zu hoch. Deshalb: Andere Verfahren, die die Serialisierbarkeit gewährleisten
- **Pessimistische Ablaufsteuerung** (Locking)
  - Konflikte werden vermieden, indem Transaktionen durch Sperren blockiert werden
  - Nachteil: ggf. lange Wartezeiten
  - Vorteil: I.d.R. nur wenig Rücksetzungen aufgrund von Synchronisationsproblemen nötig
  - Standardverfahren
- **Optimistische Ablaufsteuerung** (Zeitstempelverfahren)
  - Transaktionen werden im Konfliktfall zurückgesetzt
  - Transaktionen arbeiten bis zum COMMIT ungehindert. Anschließend erfolgt Prüfung anhand von Zeitstempeln, ob ein Konflikt aufgetreten ist
  - Nur geeignet, falls Konflikte zwischen Schreibern eher selten auftreten

Techniken zur Synchronisation

- Nur-lesende Transaktionen können sich gegenseitig nicht beeinflussen, da Synchronisationsprobleme nur im Zusammenhang mit Schreiboperationen auftreten.
- Es gibt deshalb die Möglichkeit, Transaktionen als nur-lesend zu markieren, wodurch die Synchronisation vereinfacht und ein höherer Parallelitätsgrad ermöglicht wird:
  - `SET TRANSACTION READ-ONLY`:
    kein INSERT, UPDATE, DELETE
  - `SET TRANSACTION READ-WRITE`:
    alle Zugriffe möglich

Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

Kapitel 9: Physische Datenorganisation

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

Wiederholung (1)

Permanente Datenspeicherung: Daten können auf dem sog. Externspeicher (auch Festplatte genannt) permanent gespeichert werden

- Arbeitsspeicher:
  - rein elektronisch (Transistoren und Kondensatoren)
  - flüchtig
  - schnell: 10 ns/Zugriff *
  - wahlfreier Zugriff
  - teuer: 300-400 € für 1 GByte*

Externspeicher:

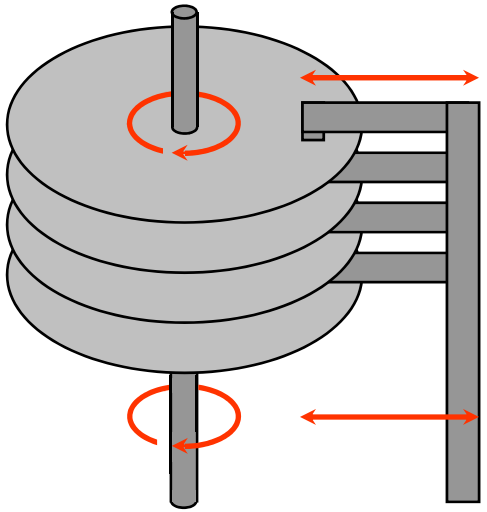
- Speicherung auf magnetisierbaren Platten (rotierend)
- nicht flüchtig
- langsam: 5 ms/Zugriff *
- blockweiser Zugriff
- wesentlich billiger: 200-400 € für 100 GByte*

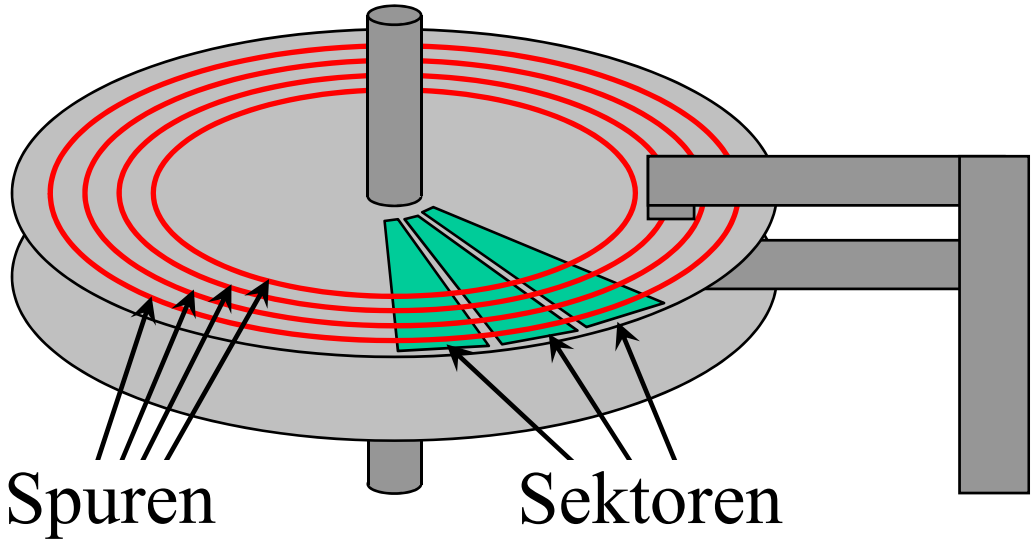
*April 2002

Wiederholung (2)

Aufbau einer Festplatte

- Mehrere magnetisierbare Platten rotieren um eine gemeinsame Achse
- Ein Kamm mit je zwei Schreib-/Leseköpfen pro Platte (unten/oben) bewegt sich in radialer Richtung.

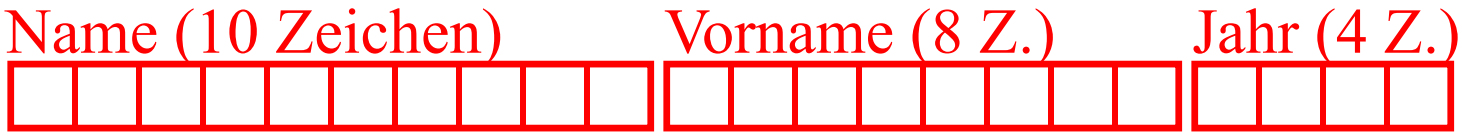




Wiederholung (3)

Intensionale Ebene vs. Extensionale Ebene

- Datenbankschema:



- Ausprägung der Datenbank:

Franklin	Aretha	1942
Ritchie	Lionel	1949

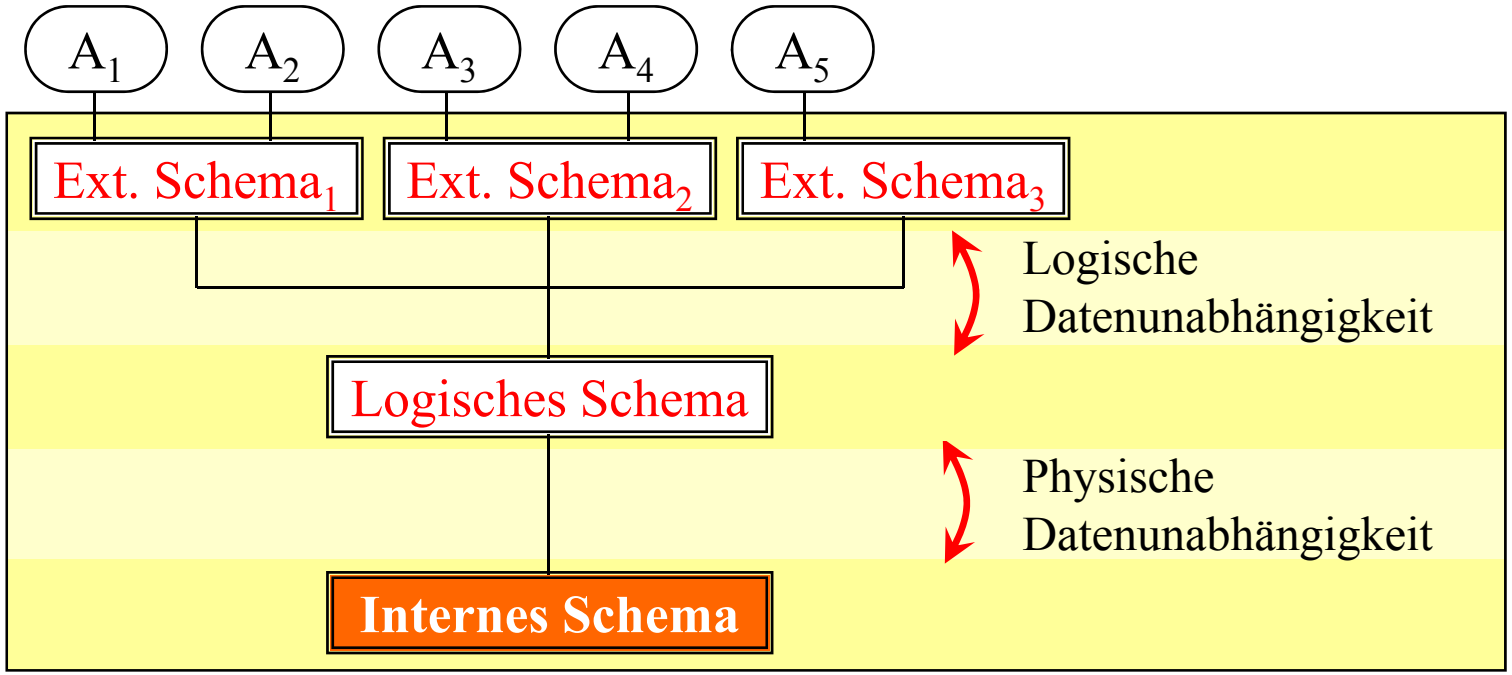
- Nicht nur DB-Zustand, sondern auch DB-Schema wird in DB gespeichert.
- Vorteil: Sicherstellung der Korrektheit der DB

Wiederholung (4)

Drei-Ebenen-Architektur zur Realisierung von

- **physischer**
- **und logischer**

Datenunabhängigkeit (nach ANSI/SPARC)



Wiederholung (5)

- Das interne Schema beschreibt die systemspezifische Realisierung der DB-Objekte (physische Speicherung), z.B.
  - Aufbau der gespeicherten Datensätze
  - Indexstrukturen wie z.B. Suchbäume
- Das interne Schema bestimmt maßgeblich das Leistungsverhalten des gesamten DBS
- Die Anwendungen sind von Änderungen des internen Schemas nicht betroffen (physische Datenunabhängigkeit)

Indexstrukturen (1)

- Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystems geeignete Datenstrukturen und Speicherungsverfahren (**Indexstrukturen**) ein.

- **Aufgaben**
  - **Zuordnung eines Suchschlüssels** zu denjenigen physischen Datensätzen, die diese Wertekombination besitzen,

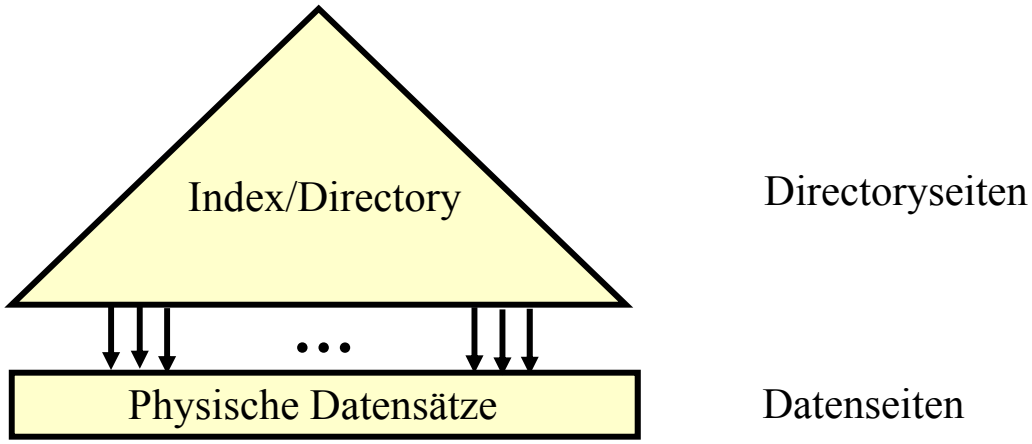
    d.h. Zuordnung zu der oder den Seiten der Datei, in denen diese Datensätze gespeichert sind.

    (VW, Golf, schwarz, M-ÜN 40) → (logische) Seite 37
  - **Organisation der Seiten** unter dynamischen Bedingungen.

    Überlauf einer Seite → Aufteilen der Seite auf zwei Seiten

Indexstrukturen (2)

- **Aufbau**
  Strukturinformation zur Zuordnung von Suchschlüsseln und zur Organisation der Datei.
  - **Directoryseiten:**
    Seiten in denen das Directory gespeichert wird.
  - **Datenseiten**
    Seiten mit den eigentlichen physischen Datensätzen.



Anforderungen an Indexstrukturen (1)

- **Effizientes Suchen**
  - Häufigste Operation in einem DBS: Suchanfragen.
  - Insbesondere Suchoperationen müssen mit wenig Seitenzugriffen auskommen.

    Beispiel: unsortierte sequentielle Datei
    - Einfügen und Löschen von Datensätzen werden effizient durchgeführt.
    - Suchanfragen müssen ggf. die gesamte Datei durchsuchen.
  - Eine Anfrage sollte daher mit Hilfe der Indexstruktur möglichst schnell zu der Seite oder den Seiten geführt werden, auf denen sich die gesuchten Datensätze befinden.

Anforderungen an Indexstrukturen (2)

- **Dynamisches Einfügen, Löschen und Verändern von Datensätzen**
  - Der Datenbestand einer Datenbank verändert sich im Laufe der Zeit.
  - Verfahren, die zum Einfügen oder Löschen von Datensätzen eine Reorganisation der gesamten Datei erfordern, sind nicht akzeptabel.

    Beispiel: sortierte sequentielle Datei
    - Das Einfügen eines Datensatzes erfordert im schlechtesten Fall, dass alle Datensätze um eine Position verschoben werden müssen.
    - Folge: auf alle Seiten der Datei muss zugegriffen werden.
  - Das Einfügen, Löschen und Verändern von Datensätzen darf daher nur *lokale Änderungen* bewirken.

Anforderungen an Indexstrukturen (3)

- **Ordnungserhaltung**
  - Datensätze, die in ihrer Sortierordnung direkt aufeinander folgen, werden oft gemeinsam angefragt.
  - In der Ordnung aufeinander folgende Datensätze sollten in der gleichen Seite oder in benachbarten Seiten gespeichert werden.
- **Hohe Speicherplatzausnutzung**
  - Dateien können sehr groß werden.
  - Eine möglichst hohe Speicherplatzausnutzung ist wichtig:
    - Möglichst geringer Speicherplatzverbrauch.
    - Im Durchschnitt befinden sich mehr Datensätze in einer Seite, wodurch auch die Effizienz des Suchens steigt und die Ordnungserhaltung an Bedeutung gewinnt.

Klassen von Indexstrukturen

- **Datenorganisierende Strukturen**
  Organisiere die Menge der tatsächlich auftretenden Daten (Suchbaumverfahren)
- **Raumorganisierende Strukturen**
  Organisiere den Raum, in den die Daten eingebettet sind (dynamische Hash-Verfahren)

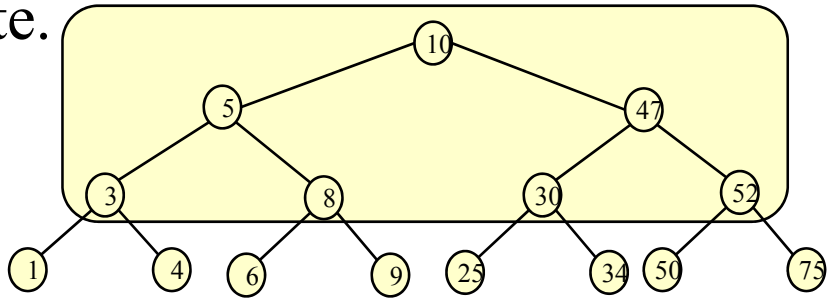
Anwendungsgebiete:

- **Primärschlüsselsuche** (B-Baum und lineares Hashing)
- **Sekundärschlüsselsuche** (invertierte Listen)

B-Baum (1)

Idee:

- Daten auf der Festplatte sind in Blöcken organisiert (z.B. 4 Kb Blöcke)
- Bei Organisation der Schlüssel mit einem binärem Suchbaum entsteht pro Knoten, der erreicht wird, ein Seitenzugriff auf der Platte.
  => sehr teuer



- Fasse mehrere Knoten zu einem zusammen, so dass ein Knoten im Baum einer Seite auf der Platte entspricht.

B-Baum (2)

Definition: B-Baum der Ordnung *m*
(Bayer und McCreight (1972))

(1) Jeder Knoten enthält höchstens $2m$ Schlüssel.

(2) Jeder Knoten außer der Wurzel enthält mindestens m Schlüssel.

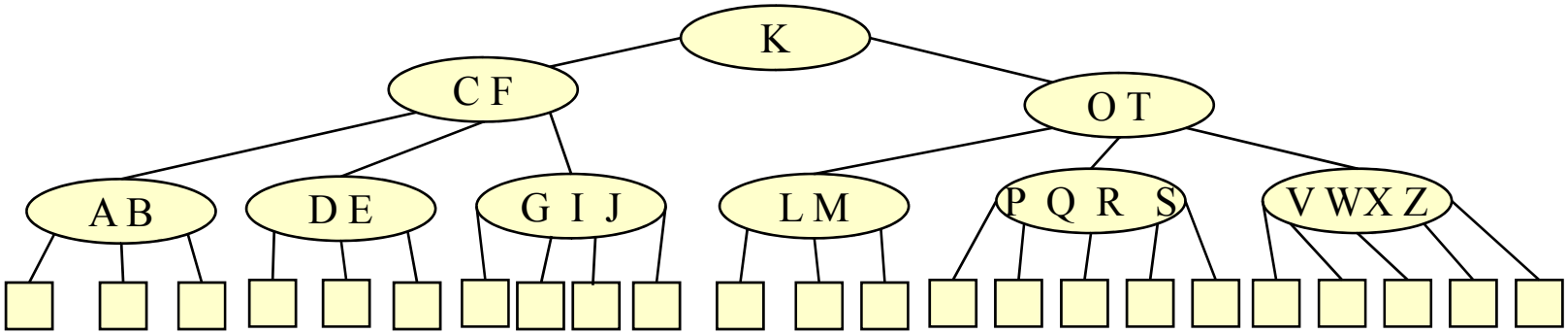
(3) Die Wurzel enthält mindestens einen Schlüssel.

(4) Ein Knoten mit k Schlüsseln hat genau $k+1$ Söhne.

(5) Alle Blätter befinden sich auf demselben Level.

B-Baum (3)

Beispiel: B-Baum der Ordnung 2



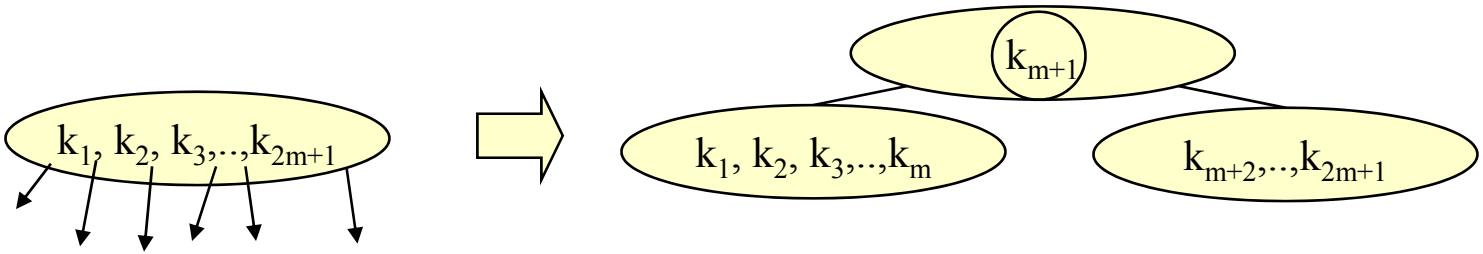
- max Höhe: $h \leq \left\lfloor \log_{m+1}\left(\frac{n+1}{2}\right)\right\rfloor + 1$

- Ordnung in realen B-Bämen: 600-900 Schlüssel pro Seite
- Effiziente Suche in den Knoten ? => binäre Suche

Einfügen in B-Baum

Einfügen eines Schlüssels ***k***:

- Suche Knoten ***B*** in den ***k*** eingeordnet werden würde. (Blattknoten bei erfolgloser Suche)
- 1. Fall: ***B*** enthält ≤ 2***m*** Schlüssel
  => füge ***k*** in ***B*** ein
- 2. Fall: ***B*** enthält 2***m*** Schlüssel
  => Overflow Behandlung
  - Split des Blattknotens



- Split kann sich über mehrere Ebene fortsetzen bis zur Wurzel

Entfernen aus B-Baum

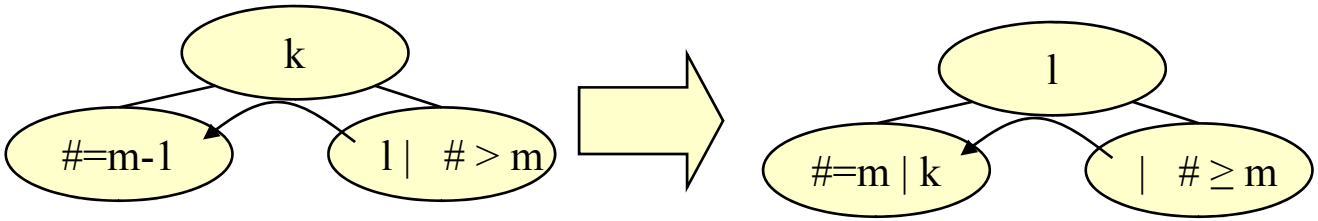
Lösche Schlüssel ***k*** aus Baum:

- Suche Schlüssel
- Falls Schlüssel in inneren Knoten, vertausche Schlüssel mit dem größten Schlüssel im linkem Teilbaum (=> Rückführung auf Fall mit Schlüssel in Blattknoten)
- Falls Schlüssel im Blattknoten ***B***:
  - 1. Fall: ***B*** hat noch mehr als ***m*** Schlüssel, => lösche Schlüssel
  - 2. Fall: ***B*** hat genau ***m*** Schlüssel

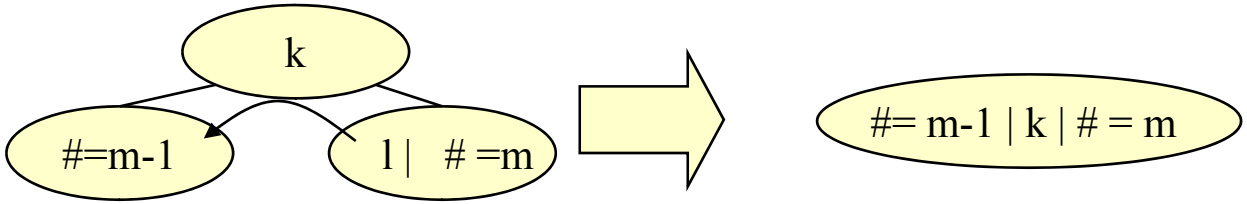
    => Underflow

Underflow-Behandlung im B-Baum

- Betrachte Bruderknoten (immer den rechten falls vorhanden)
- 1.Fall: Bruder hat mehr als m Knoten => ausgleichen mit Bruder



- 2. Fall: Bruder hat genau m Knoten => Verschmelzen der Brüder



- Verschmelzen kann sich bis zur Wurzel hin fortsetzen.

B+-Baum (1)

- Häufig tritt in Datenbankanwendungen neben der Primärschlüsselsuche auch sequentielle Verarbeitung auf.
- **Beispiele für sequentielle Verarbeitung:**
  - *Sortiertes Auslesen aller Datensätze*, die von einer Indexstruktur organisiert werden.
  - *Unterstützung von Bereichsanfragen* der Form:
  - "Nenne mir alle Studenten, deren Nachname im Bereich [Be ... Brz] liegt."
- →Die Indexstruktur sollte die *sequentielle Verarbeitung* unterstützen, d.h. die Verarbeitung der Datensätze in aufsteigender Reihenfolge ihrer Primärschlüssel.

B+-Baum (2)

Grundidee:

- Trennung der Indexstruktur in *Directory* und *Datei*.
- *Sequentielle Verkettung* der Daten in der Datei.

B+-Datei:

- Die Blätter des B+-Baumes heißen ***Datenknoten*** oder ***Datenseiten***.
- Die Datenknoten enthalten alle Datensätze.
- Alle Datenknoten sind entsprechend der Ordnung auf den Primärschlüsseln *verkettet*.

B+-Directory:

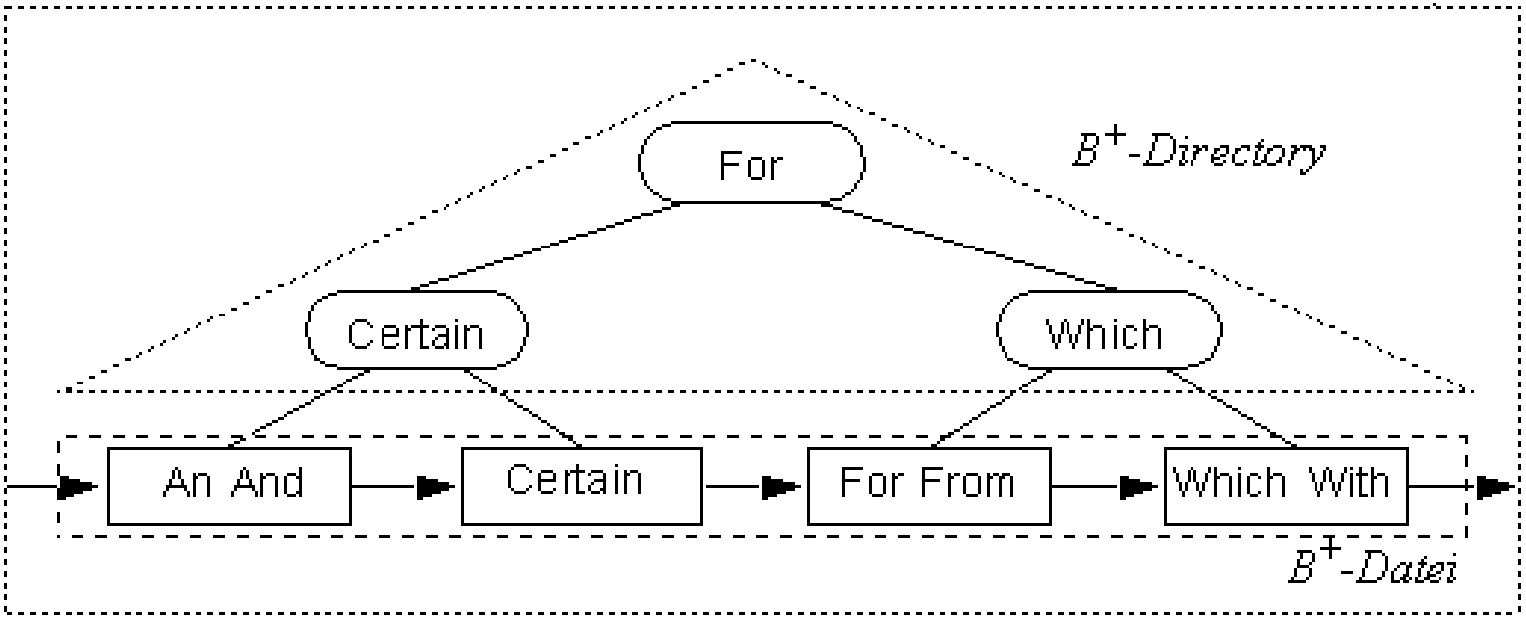
- Die inneren Knoten des B+-Baumes heißen ***Directoryknoten*** oder ***Directoryseiten***.
- Directoryknoten enthalten nur noch ***Separatoren*** s.
- Für jeden Separator s(u) eines Knotens u gelten folgende ***Separatoreneigenschaften***:
  - $s(u) > s(v)$ für alle Directoryknoten v im linken Teilbaum von s(u).
  - $s(u) < s(w)$ für alle Directoryknoten w im rechtenTeilbaum von s(u).
  - $s(u) > k(v')$ für alle Primärschlüssel k(v') und alle Datenknoten v' im linken Teilbaum von s(u).
  - $s(u) \leq k(w')$ für alle Primärschlüssel k(w') und alle Datenknoten w' im rechten Teilbaum von s(u).

B+-Baum (3)

Beispiel:

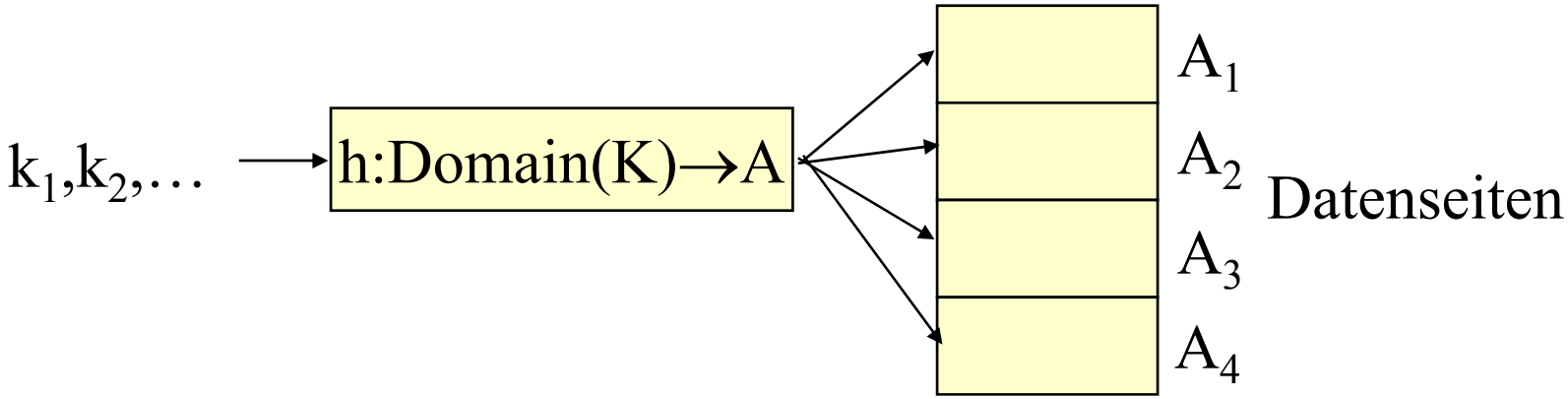
B+-Baum für die Zeichenketten:

An, And, Certain, For, From, Which, With



Hash-Verfahren

- Raumorganisierendes Verfahren
- **Idee**: Verwende Funktion, die aus den Schlüsseln ***K*** die Seitenadresse ***A*** berechnet. (Hashfunktion)
- **Vorteil**: Im besten Fall konstante Zugriffszeit auf Daten.
- **Probleme:**
  - Gleichmäßige Verteilung der Schlüssel über A
  - |Domain(K)|>>|A| => Kollision



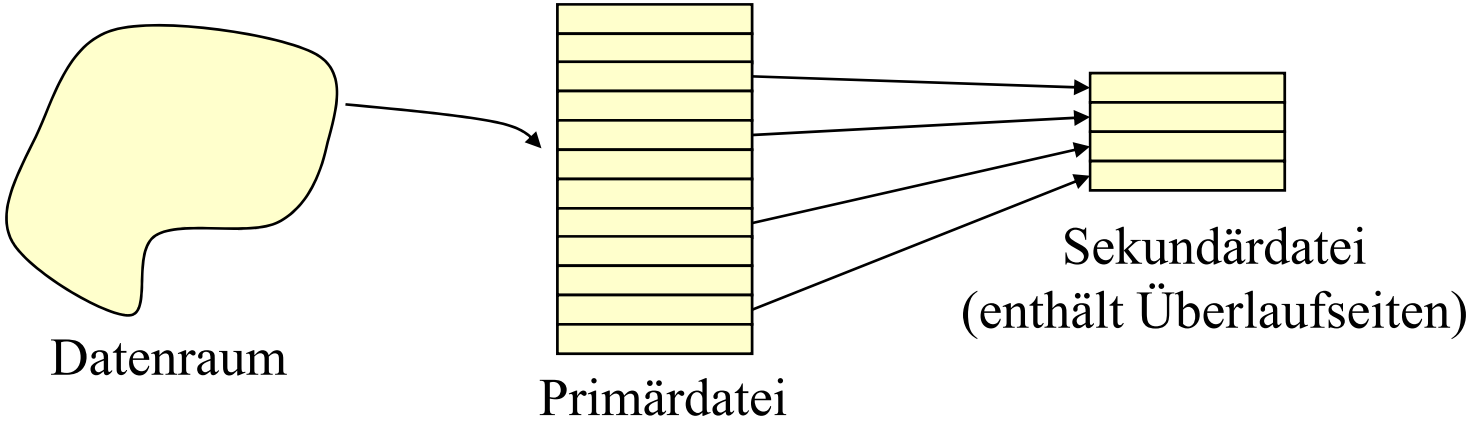
Hash-Verfahren für Sekundärspeicher

Für Sekundärspeicher sind weitere Anforderungen von Bedeutung:

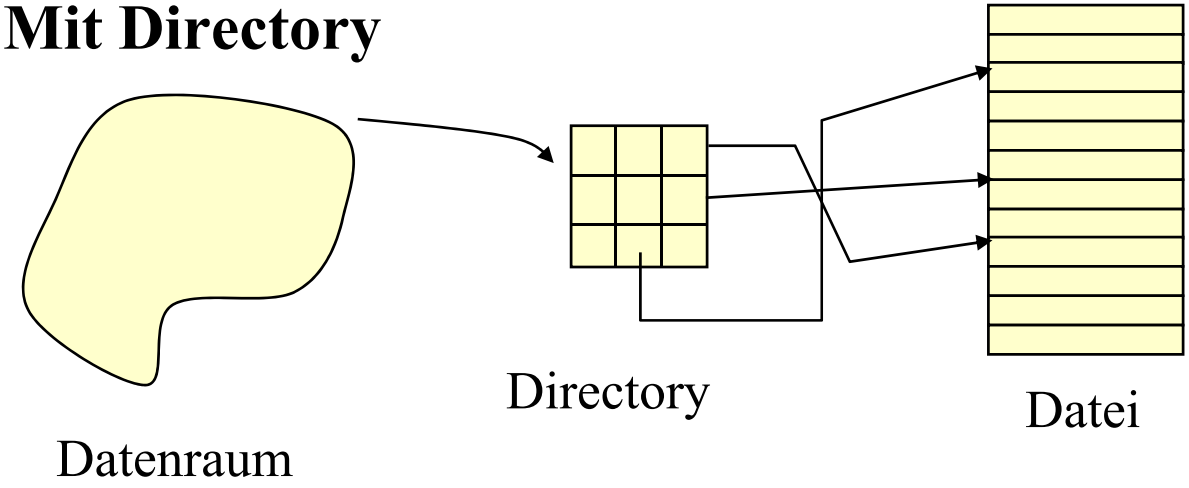
- hohe Speicherplatzausnutzung
  (Datenseiten sollten über 50 % gefüllt sein)
- Gutes dynamisches Verhalten:
  schnelles Einfügen, Löschen von Schlüsseln und Datenseiten
- Gleichbleibend effiziente Suche

Klassifizierung von Hash-Verfahren

- **Ohne Directory**



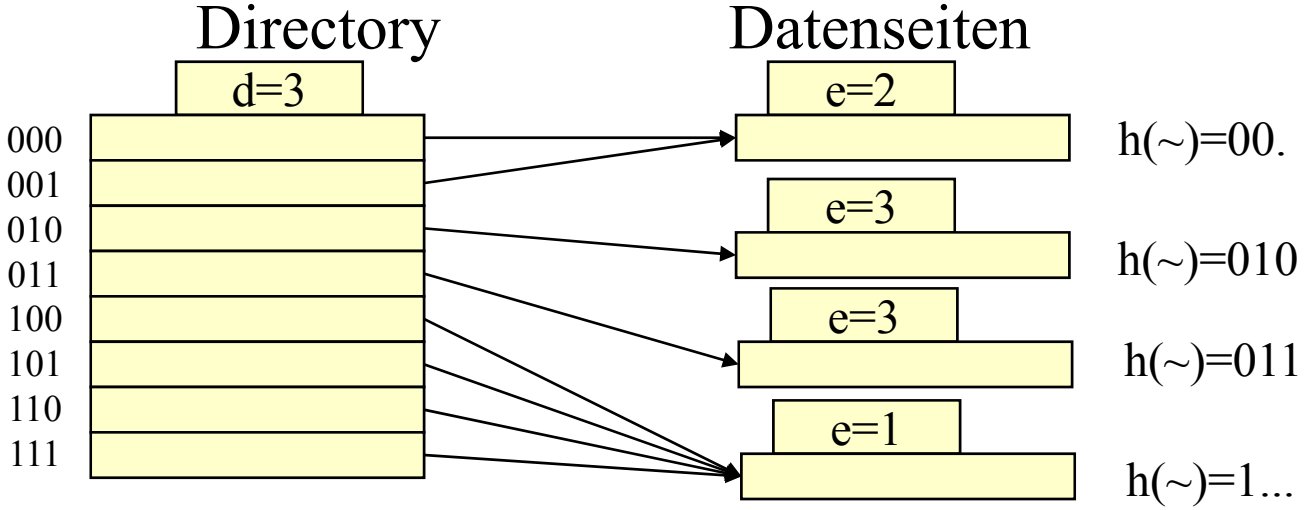
- **Mit Directory**



Hash-Verfahren mit Directory

Erweiterbares Hashing

- Hashfunktion: h(k) liefert Bitfolge $(b_1,b_2,\ldots,b_d,\ldots)$
- Directory besteht aus eindimensionalen Array D $[0..2^d-1]$ aus Seitenadressen. d heißt Tiefe des Directory.
- Verschiedene Einträge können auf die gleiche Seite zeigen



Einfügen Erweiterbares Hashing (1)

Gegeben: Datensatz mit Schlüssel k

1. Schritt: Bestimme die ersten Bits des Pseudoschlüssels $h(k) = (b_1,b_2,..,b_d,..)$
2. Schritt:
   Der Directoryeintrag $D[b_1,b_2,..,b_d]$ liefert Seitennummer.
   Datensatz wird in berechnete Seite eingefügt.

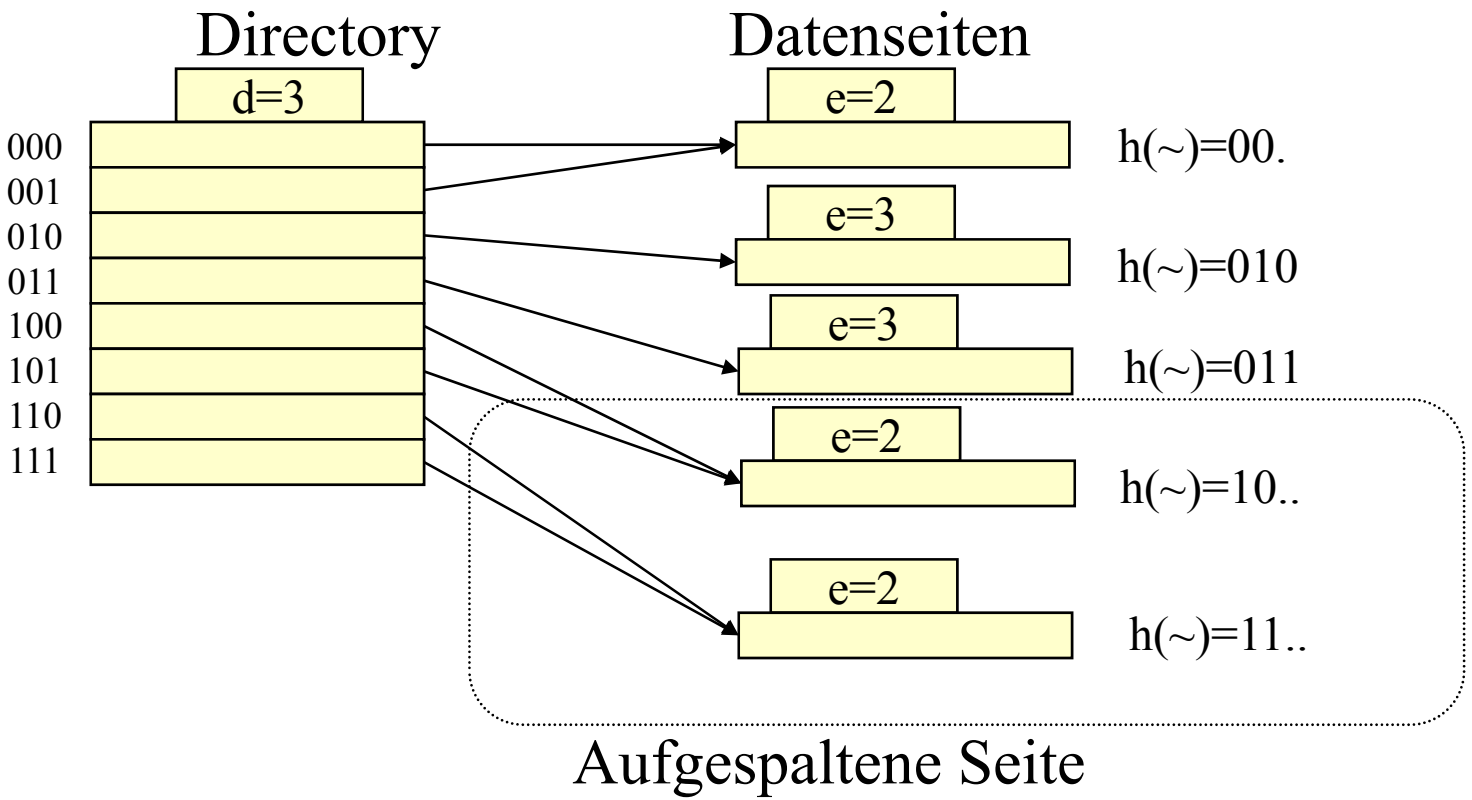
Falls Seite danach max. gefüllt:

1. Aufspalten der Datenseite.
2. Verdoppeln des Directory.

Einfügen Erweiterbares Hashing (2)

Aufspalten einer Datenseite

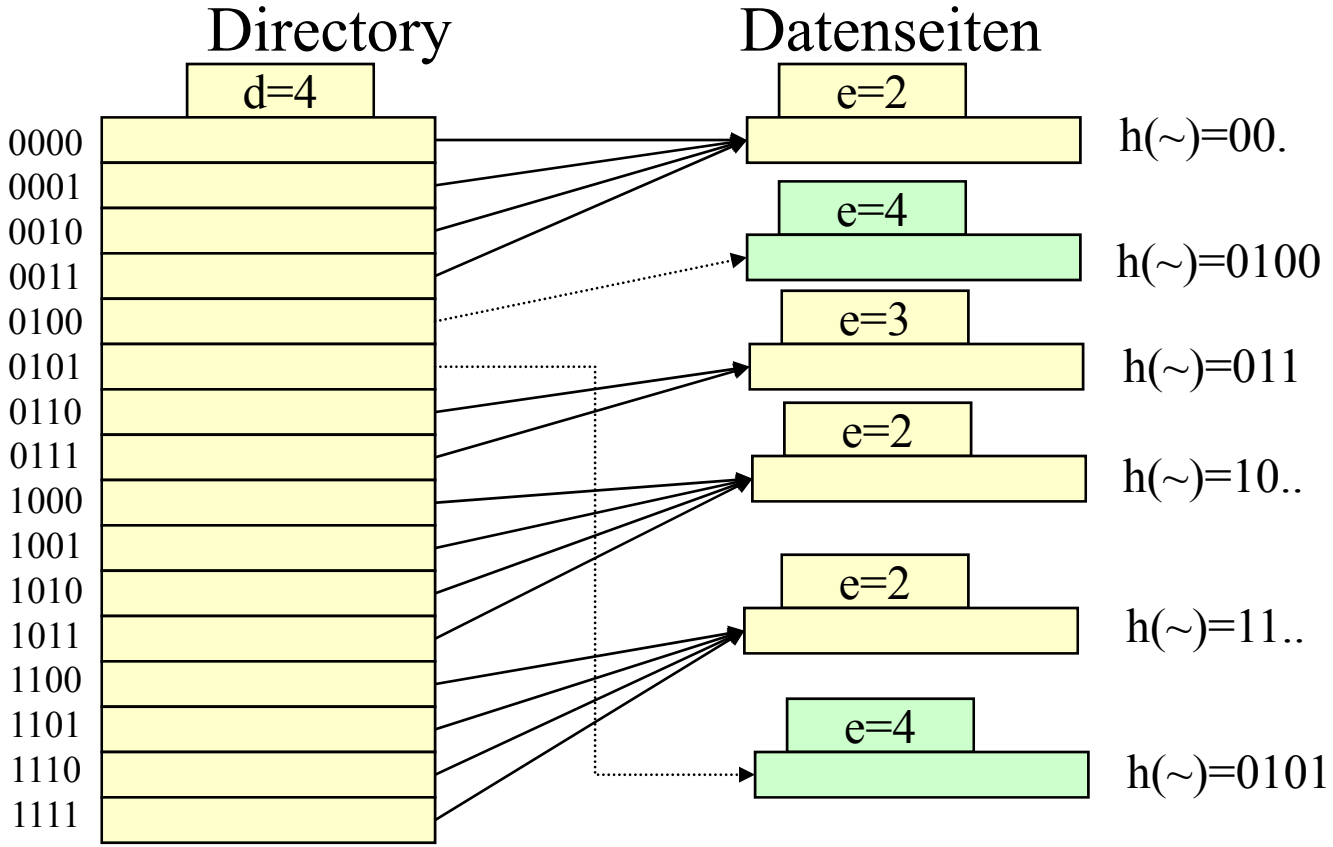
Aufspalten wenn Füllungsgrad einer Seite zu hoch(>90%).



Einfügen Erweiterbares Hashing (3)

Verdopplung des Directory

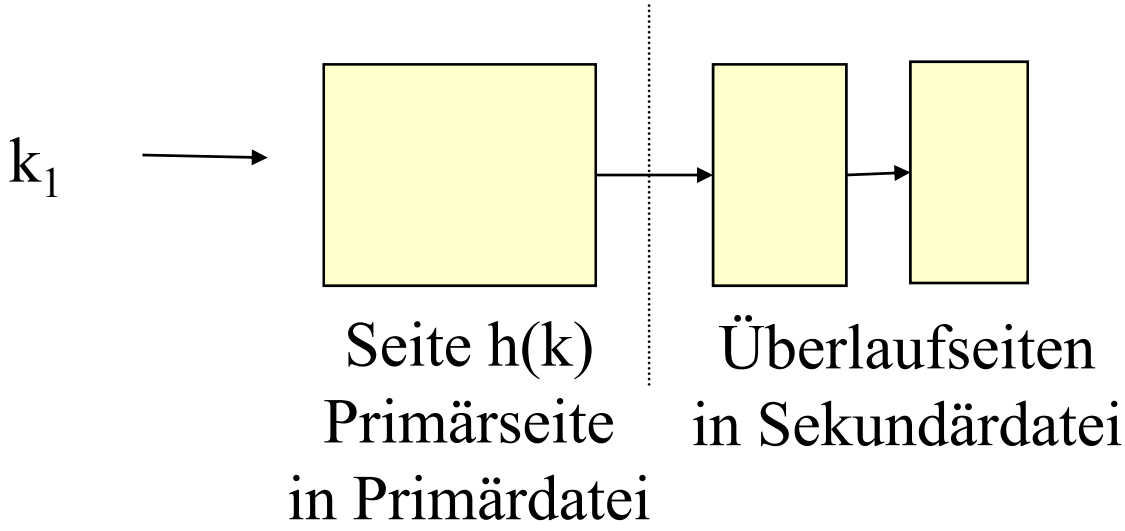
Datenseite läuft über und d = e.



Hashing ohne Directory

Lineares Hashing

- Hash-Funktion h:K→A liefert direkt eine Seitenadresse
- Problem: Was ist wenn Datenseite voll ist ?
- Lösung: Überlaufseiten werden angehängt. Aber bei zu vielen Überlaufseiten degeneriert Suchzeit.



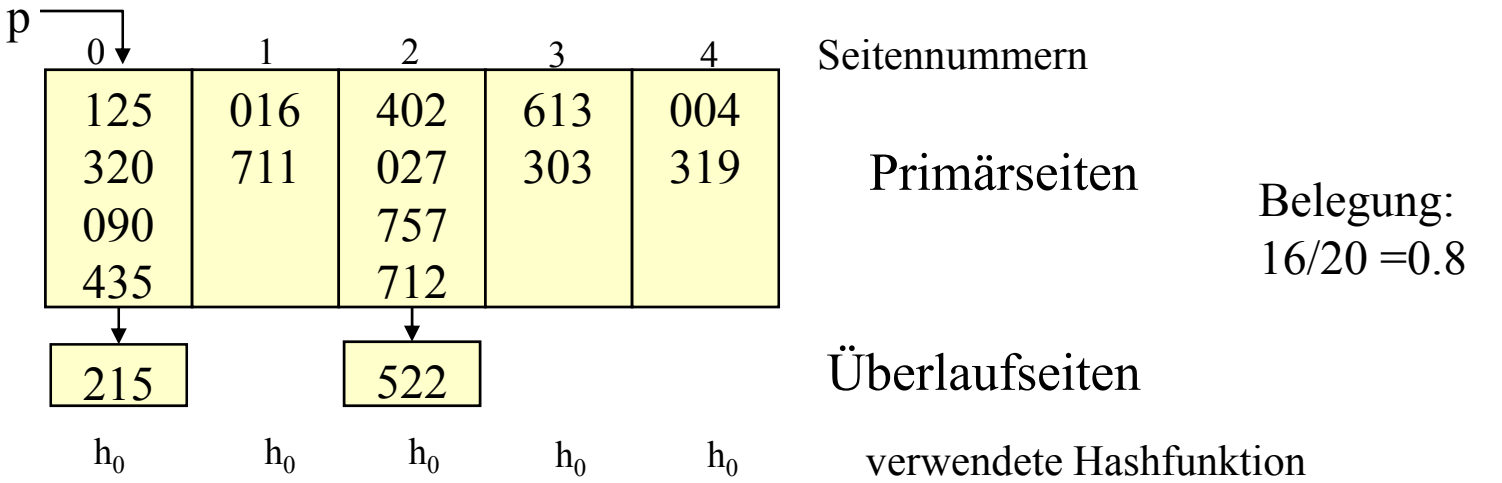
Lineares Hashing (1)

- dynamisches Wachstum der Primärdatei
- Folge von Hash-Funktionen: h_0, h_1, h_2, …
- Erweitern der Primärdatei um jeweils eine Seite
- feste Splitreihenfolge
- Expansionzeiger zeigt an welche Seite gesplittet wird
- Kontrollfunktion: Wann wird gesplittet ?
  Belegungsfaktor übersteigt Schwellwert:
  z.B.

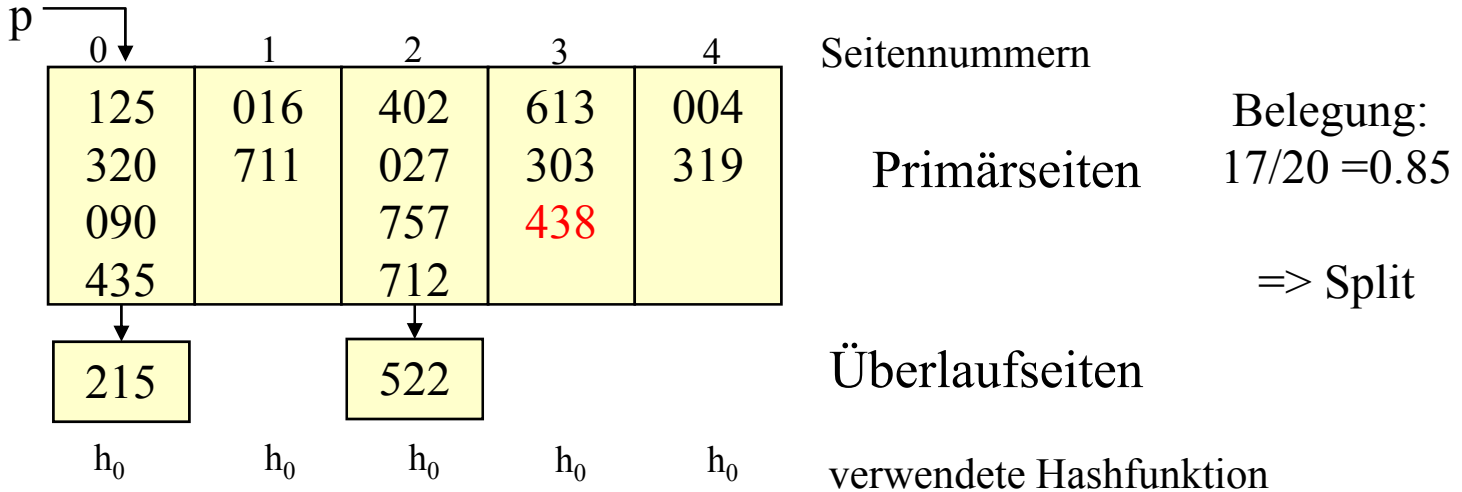
$$80\% < \frac{\#\,abgespeicherte\;\;Datensätze}{\#\,mögl.\;\;Datensätze\;\;in\;\;Primärdatei}$$

Lineares Hashing (2)

- Hashfunktionen: $h_0(k)=k \; mod(5)$, $h_1(k)=k \; mod(10)$, …

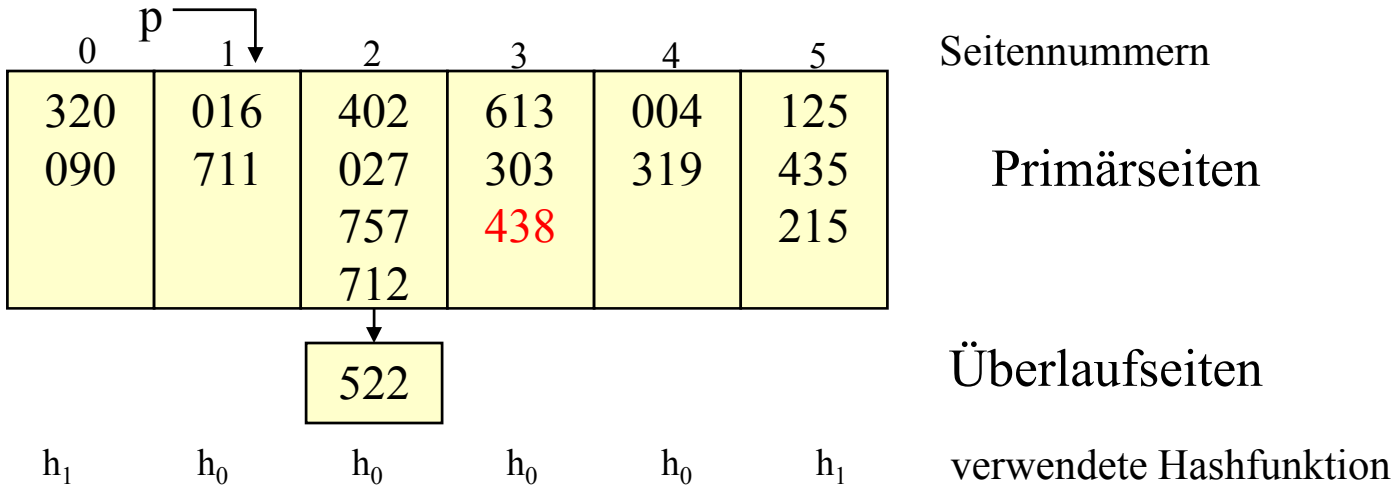


Einfügen von Schlüssel 438:



Lineares Hashing (3)

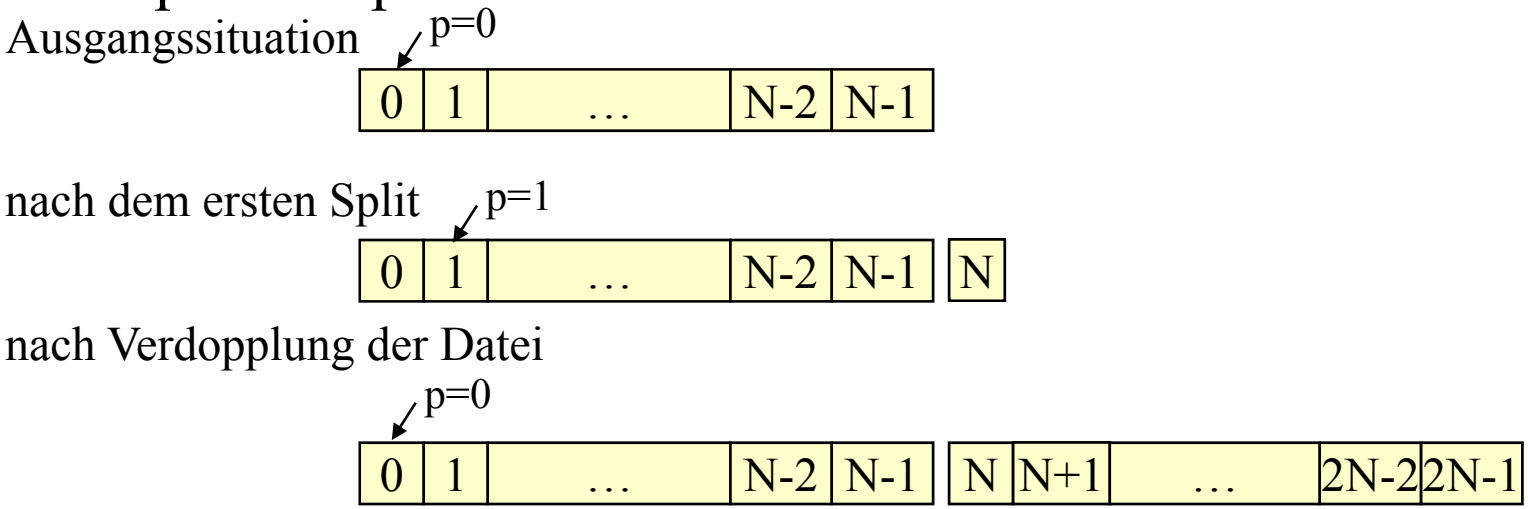
Expansion der Seite 0 auf die Seiten 0 und 5:



- Umspeichern aller Datensätze mit $h_1(k) = 5$ in neue Seite
- Datensätze mit $h_1(k) = 0$ bleiben

Lineares Hashing (4)

Prinzip der Expansion:



- Split in fester Ordnung (nicht: Split der vollen Seiten)
- trotzdem wenig Überlaufseiten
- gute Leistung für gleich verteilte Daten
- Adreßraum wächst linear

Lineares Hashing (5)

Anforderungen an die Hashfunktionen $\{h_i\}$, $i>0$:

1.) Bereichsbedingung:

$h_{L:}\ \text{domain}(k) \rightarrow \{0,1,\ldots,(2^L * N)-1\},\ L \geq 0$

2.) Splitbedingung:

$h_{L+1}(k)=h_L(k)$ oder

$h_{L+1}(k)=h_L(k) + 2^L * N,\ L \geq 0$

L gibt an wie oft sich Datei schon vollständig verdoppelt hat.

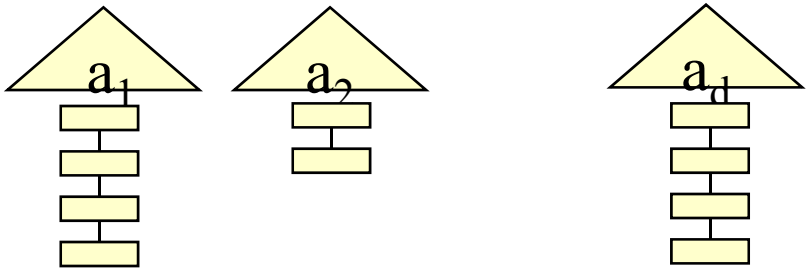
Beispiel:

$h(k)= k \bmod (2^L * N)$

Anfragen auf mehreren Attributen (1)

Invertierte Listen (häufigste Lösung)

- jedes relevante Attribute wird mit eindimensionalem Index verwaltet.
- Suche nach mehreren Attributen $a_1,a_2,..,a_d$
- Erstellen von Ergebnislisten mit Datensätzen d bei denen $d.a_1$ der Anfragebedingung genügt.



- Bestimmen des Ergebnis über mengentheoretischen Verknüpfung (z.B. Schnitt) der einzelnen Ergebnislisten.

Anfragen auf mehreren Attributen (2)

Eigenschaften Invertierter Listen:

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Suchzeit wächst mit Anzahl der Attribute
- genügend Effizienz bei kleinen Listen
- Sekundärindizes für nicht Primärschlüssel beeinflussen die physikalische Speicherung nicht.
- zusätzliche Sekundärindizes können das Leistungsverhalten bei DB-Updates stark negativ beeinflussen.

Index-Generierung in SQL

- Generierung eines Index:

  CREATE INDEX *index-name* ON *table* (a_1, a_2, ..., a_n);

  Ein *Composite Index besteht aus mehr als einer Spalte. Die Tupel sind dann nach den* Attributwerten (lexikographisch) geordnet:

  Für den Vergleich der einzelnen Attribute gilt die jeweils übliche Ordnung:

  $t_1 < t_2$ *gdw.*

  $t_1.a_1 < t_2.a_1$ *oder* ($t_1.a_1 = t_2.a_1$ *und* $t_1.a_2 < t_2.a_2$) *oder* ...

  numerischer Vergleich für numerische Typen, lexikographischer Vergleich bei **CHAR,** Datums-Vergleich bei **DATE** usw**.**

- Löschen eines Index:

  DROP INDEX *index-name;*

- Verändern eines Index:

  ALTER INDEX *index-name ...;*

  (betrifft u.a. Speicherungs-Parameter und Rebuild)

Durch Index unterstützte Anfragen

- **Exact match query:**

  SELECT * FROM *t* WHERE a_1 = ... AND ... AND a_n =...

- **Partial match query:**

  SELECT * FROM *t* WHERE a_1 = ... AND ... AND a_i = ...

  für $i < n$, *d.h. wenn die exakt spezifizierten Attribute ein Präfix der indizierten Attribute sind.*

  Eine Spezifikation von a_{i+1} *kann i.a. nicht genutzt werden, wenn* a_i *nicht spezifiziert ist.*

- **Range query:**

  SELECT * FROM *t* WHERE a_1 = ... AND ... AND a_i = ... AND a_{i+1} <= ...

  auch z.B. für '**>**' **oder 'BETWEEN'**

- **Pointset query:**

  SELECT * FROM *t* WHERE a_1 = ... AND ... AND a_i = ... AND a_{i+1} IN *(7,17,77)*

  auch z.B. (***a_{i+1}*=... *OR* a_{i+1} =... *OR ...)***

Durch Index unterstützte Anfragen

- **Pattern matching query:**

  SELECT * FROM *t* WHERE a_1=... AND ... AND a_i=... AND a_{i+1} LIKE *'c_1c_2... c_k%'

  Problem: Anfragen wie ***wort*** **LIKE *'%system'*** werden nicht unterstützt.
  Man kann aber z.B. eine Relation aufbauen, in der alle Wörter revers gespeichert werden und dann effizient nach ***revers_wort*** **LIKE *'metsys%'*** suchen lassen.

Zusammenfassung

- Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystems geeignete Datenstrukturen und Speicherungsverfahren (**Indexstrukturen**) ein.

- Primärindizes verwalten Primärschlüssel
- Sekundärindizes unterstützen zusätzliche Suchattribute oder Kombinationen von diesen.

- Ein Beispiel für eine solche Indexstruktur sind B+-Baum und dynamische Hash-Verfahren.

- Anfragen auf mehreren Attributen werden meist mit invertierten Listen realisiert.

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2008/2009

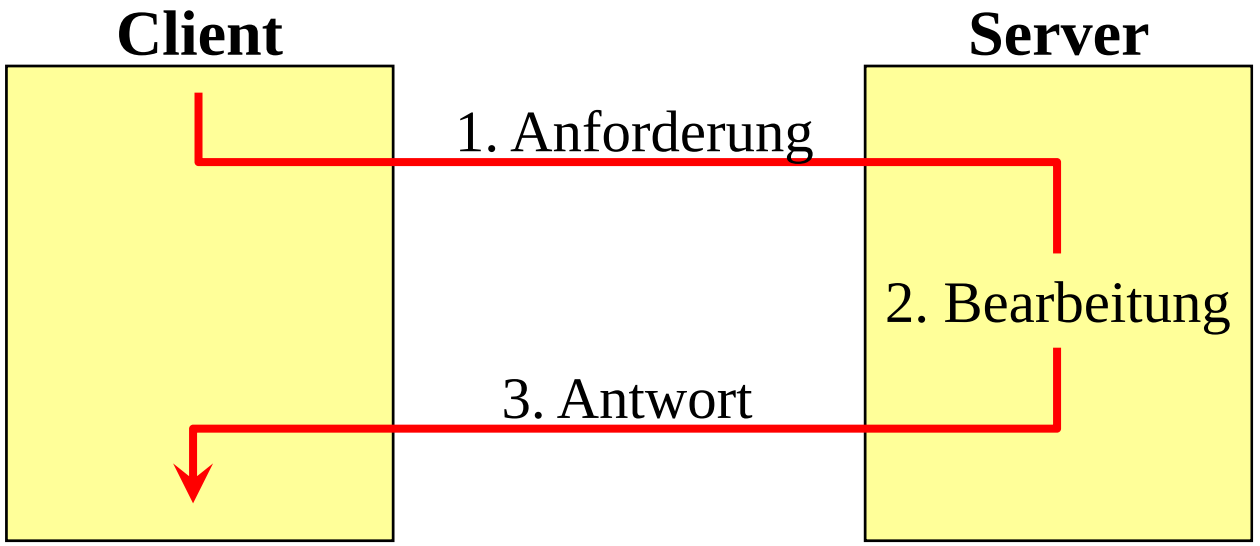
Kapitel 10: Anwendungsentwicklung

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Annahita Oswald, Bianca Wackersreuther

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBS

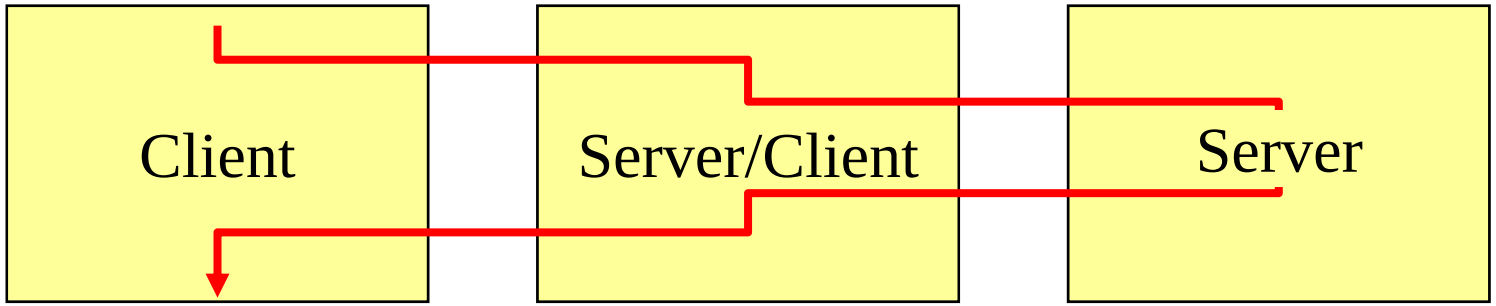
Client-Server-Architektur

- Datenbankanwendungen unterstützen gleichzeitige Arbeit von vielen Benutzern
- Dienstnehmer (Client) nimmt Dienste eines Dienstleisters (Server) zur Erfüllung seiner Aufgaben in Anspruch:



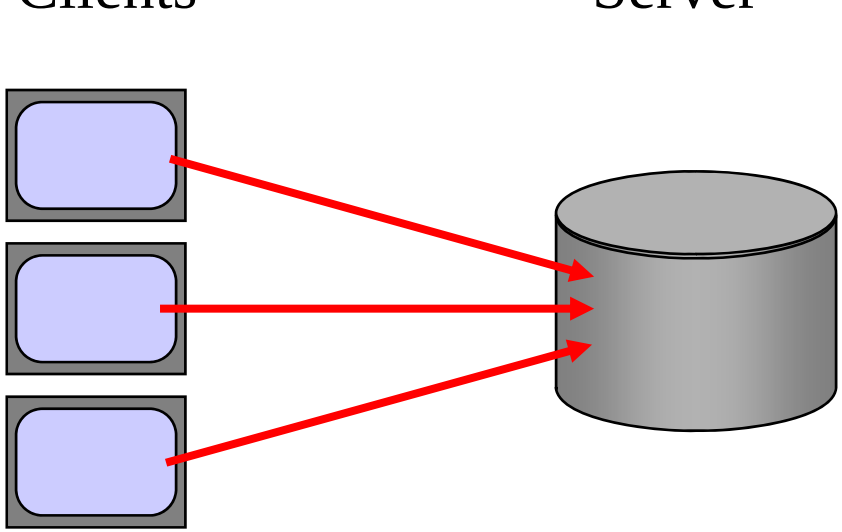
Client-Server-Architektur

- Für jeden Server bekannt, welche Dienste er erbringen kann
- Protokoll:
  Regelt Interaktionen (d.h. Aufbau der Anforderung und der Antwort)
- Asymmetrische Beziehung zw. Client und Server
  - Möglich, daß ein Client seinerseits wieder Server für einen anderen Dienst ist:



DB-Server

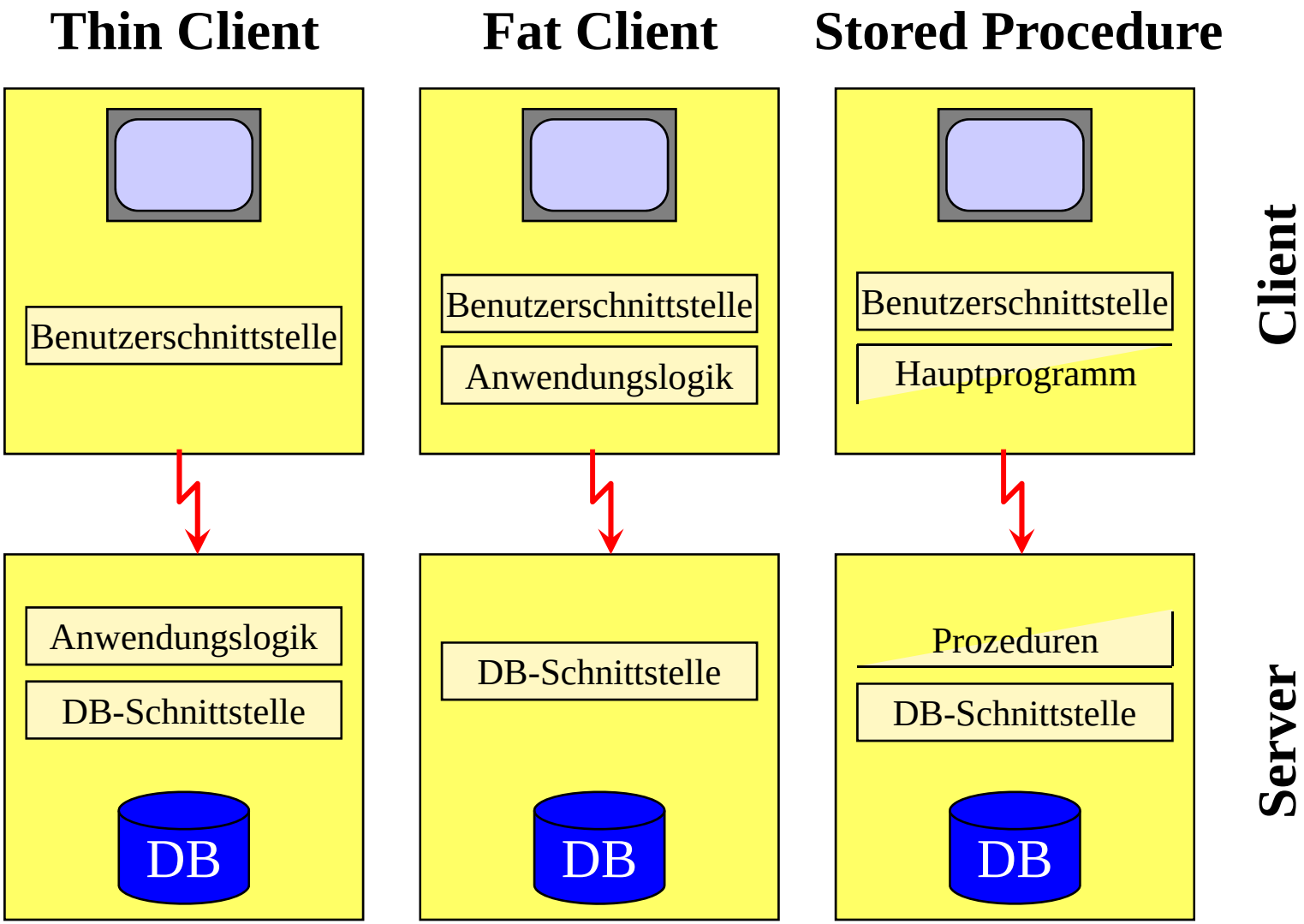
- Datenbankanwendungen:
  - Mehrere Benutzer arbeiten auf gemeinsamen Daten
  - Zentrale Kontrollinstanz ist erforderlich
  - PCs sind Frontends zur Darstellung und benutzerfreundlichen Manipulation der Daten



Aufteilung der Funktionalität

- Funktionsgruppen von Datenbankanwendungen:
  - Präsentation und Benutzerinteraktion
  - Anwendungslogik (im eigentlichen Sinn)
  - Datenmanagement (Anfragebearbeitung)
- Trennung an jeder beliebigen Stelle:
  - Präsentation und Interaktion meist vollst. im Client
  - Datenmanagement meist vollständig im Server
  - Anwendungslogik:
    - meist im Client
    - im Server: typ. Großrechner-Anwendungen
    - Aufteilung der Logik: „gespeicherte Prozeduren“

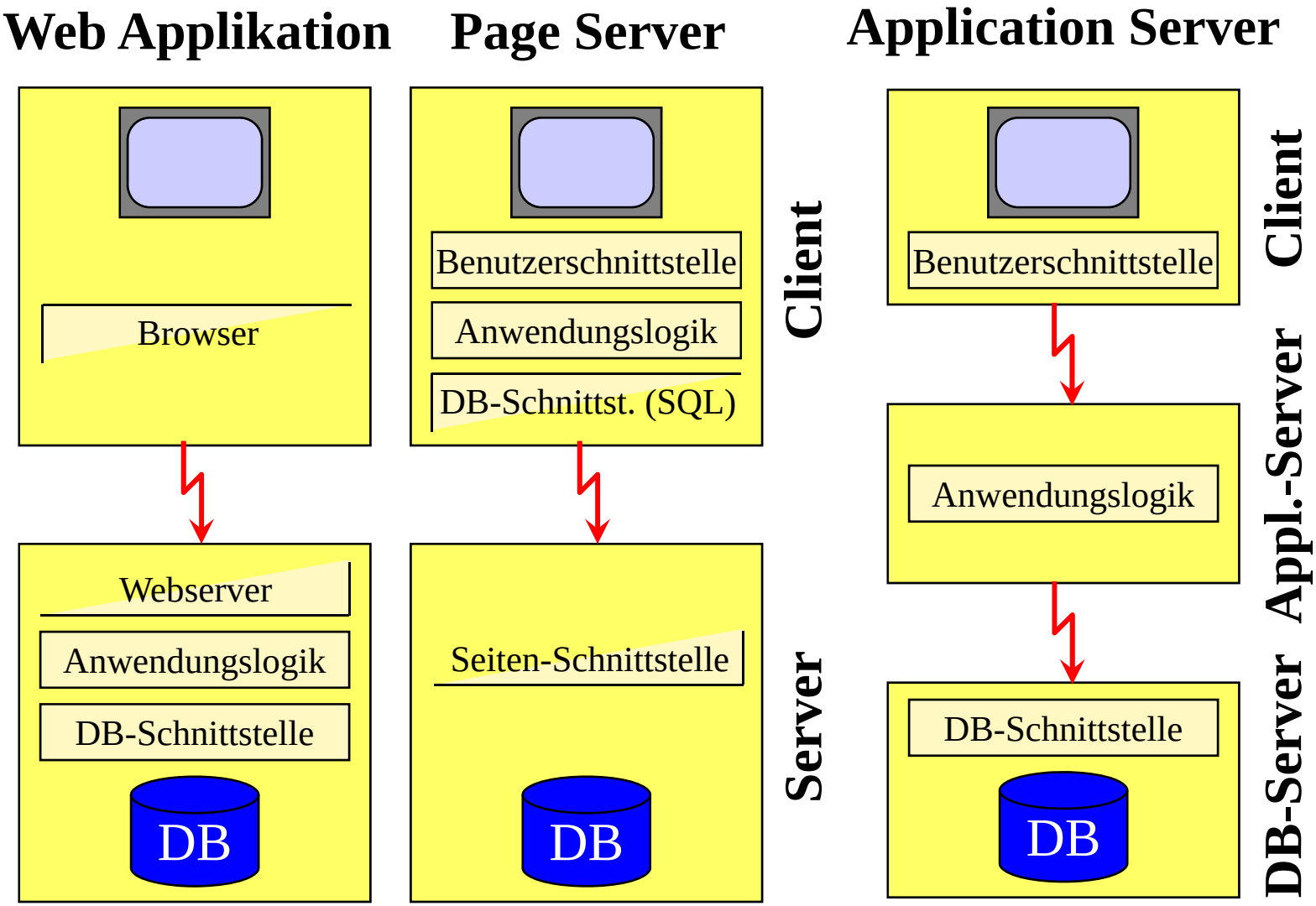
Aufteilung der Funktionalität



Aufteilung der Funktionalität

- Trennung zwischen Client und Server auch innerhalb einer Funktionsgruppe möglich:
  - Web-Anwendung:
    Server erbringt teilweise Präsentationsfunktionalität
    Client hat nur Standard-Darstellungsfunktionalität
  - Seiten-Server:
    Teil der Anfragefunktionalität (SQL) liegt im Client
    Server hat nur die Funktionalität, Datenblöcke gem. Adresse zu speichern und zu laden
- 3-stufige Hierarchie:
  - Applikationslogik bildet eigenes Client-Server-Modell
  - Applikations-Server ist Datenbank-Client

Aufteilung der Funktionalität



Protokoll

- Client und Server können
  - auf dem selben Rechner laufen
  - auf verschiedenen Rechnern laufen, die über ein Netzwerk miteinander kommunizieren
- Die Kommunikation zwischen Client und Server wird in jedem Fall durch ein **Protokoll** geregelt:
  - Bei Web-Anwendungen: **http**
  - A.-Logik/DB-Schnittstelle: Spez. Protokoll des DBMS
    - Anforderung: im wesentlichen SQL-Anfrage
    - Antwort: Tabelle bzw. Liste von Tupeln
  - Bei Applikationsservern: Standard-Protokolle wie RMI, CORBA

Vor-/Nachteile der Varianten

- Kriterien:
  - Netzbelastung
  - Belastung des Server-Rechners (gemeinsame Ressource für alle)
  - Belastung der Clients-Rechner (oft schlecht ausgestattete Bürorechner)
  - Flexibilität
  - Entwicklungsaufwand

SQL ↔ Java

- SQL: keine Schleifen, etc.
- Anwendungslogik:
  Konventionelle Programmiersprache nötig

- SQL wird mit herkömmlicher Programmiersprache (Hostsprache) gekoppelt

Kopplung SQL/Host-Sprache

- Idee:
  - Java-Programm (z.B.) enthält SQL-Anfragen
  - Diese werden an DB-Server gesandt
  - Server schickt als Antwort Ergebnistabelle
  - Java-Programm verarbeitet diese Tabelle:
    - Darstellung der Tupel am Bildschirm
    - ggf. weitere Anforderungen an die DB
- Probleme:
  - Wie können die SQL-Anfragen in das Programm integriert werden (verschiedene Sprachen)
  - Unterschiedliche Datenstruktur-Konzepte
    - Java kann nicht Tabellen verarbeiten

Integration der Anfragen

Grundsätzlich zwei verschiedene Möglichkeiten:

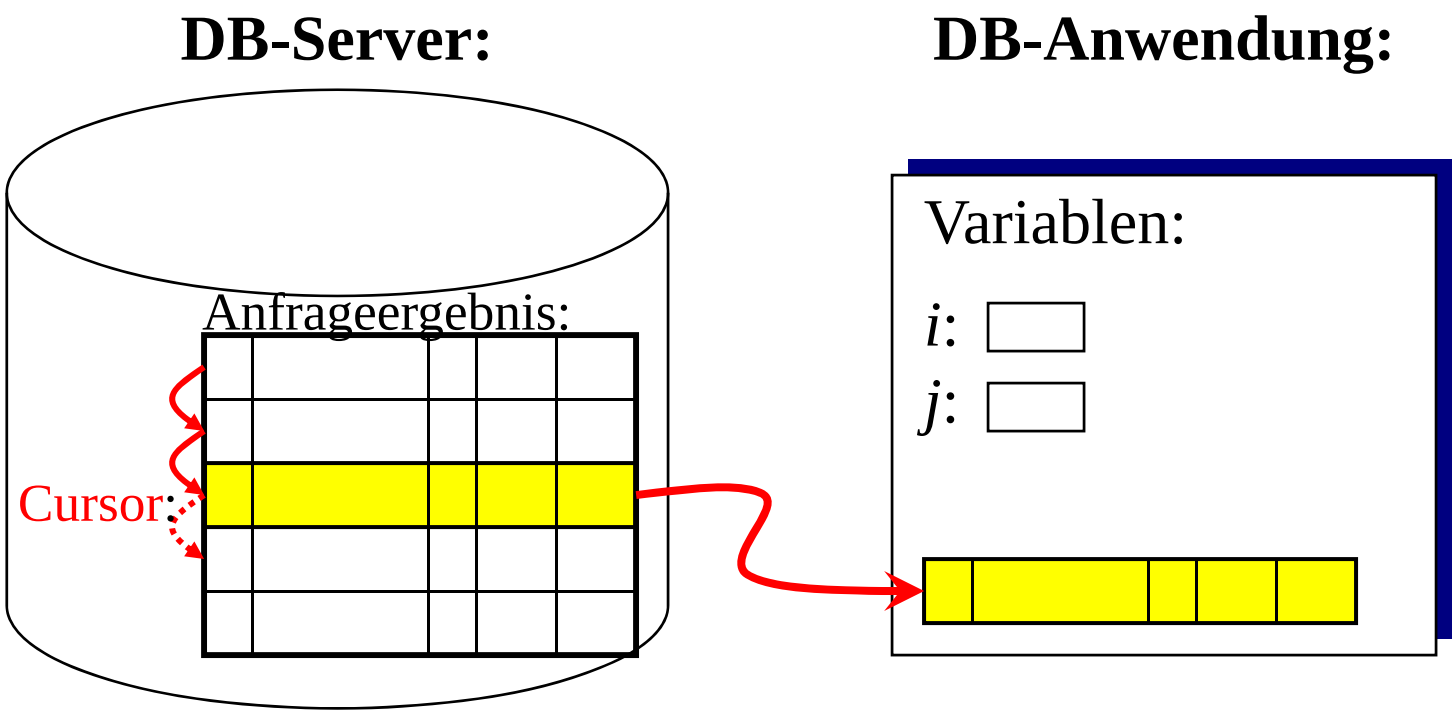
- Call-Level-Schnittstellen:
  Es gibt eine Bibliothek mit speziellen Prozeduren, die die Anfrage als Parameter enthalten:
  - CLI, ODBC, OCI, JDBC:
    Statement stmt = con.createStatement () ;
    ResultSet rs = stmt.executeQuery (“select * from ...“) ;
- Einbettung (Embedded SQL):
  Ein spezielles Schlüsselwort oder Zeichen kennzeichnet SQL-Anfrage (Vorübersetzung):
  - Embedded SQL-C (und versch. andere), JSQL, SQLJ:
    #sql {insert into Mitarbeiter values (25, “Meier“, 1) } ;

Das Cursor-Konzept

- In beiden Fällen nicht möglich/sinnvoll, sofort die gesamte Ergebnis-Tabelle an das Hostprogramm zu übergeben:
  - Nicht möglich wegen sog. Impedance Mismatch:
    - SQL beruht auf der Datenstruktur „Tabelle“
    - Java beruht auf der Datenstruktur „Datensatz“ (bzw. Tupel, Klasse)
  - Nicht sinnvoll, weil Ergebnis sehr groß werden kann:
    - Client-Rechner oft schwach ausgestattet
    - Übertragung des gesamten Ergebnisses kostet Zeit, in der der Benutzer warten muß (erste Ergebnisse)
    - Anwendungsprogramm braucht oft nur einen Teil der Ergebnistupel

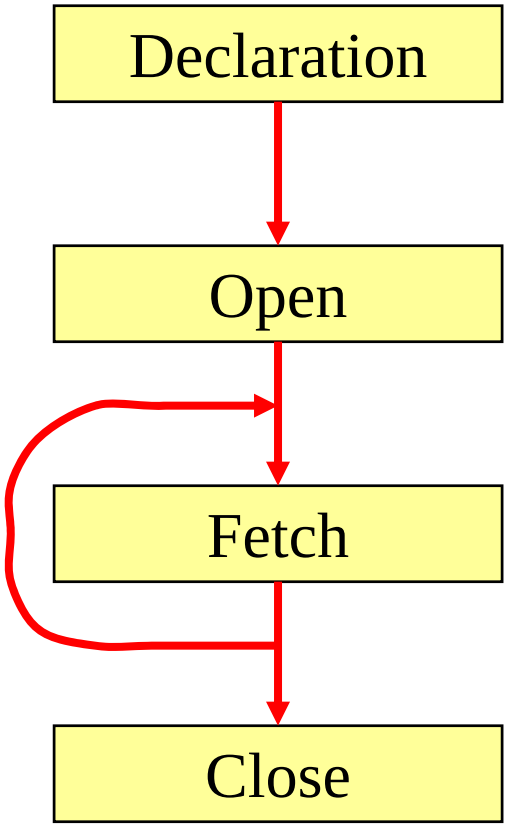
Das Cursor-Konzept

- Deshalb wird die Ergebnistabelle grundsätzlich tupelweise an das Hostprogramm übergeben.
- Cursor: Position des aktuellen Tupels



Programmierschritte beim Cursor

- Grundsätzlich sind 4 Schritte erforderlich:



Verknüpfung des Cursors mit einer bestimmten SQL-Anfrage

Auftrag an den DB-Server, die Anfrage zu bearbeiten

Verarbeitung der Ergebnistupel durch das Anwendungsprogramm

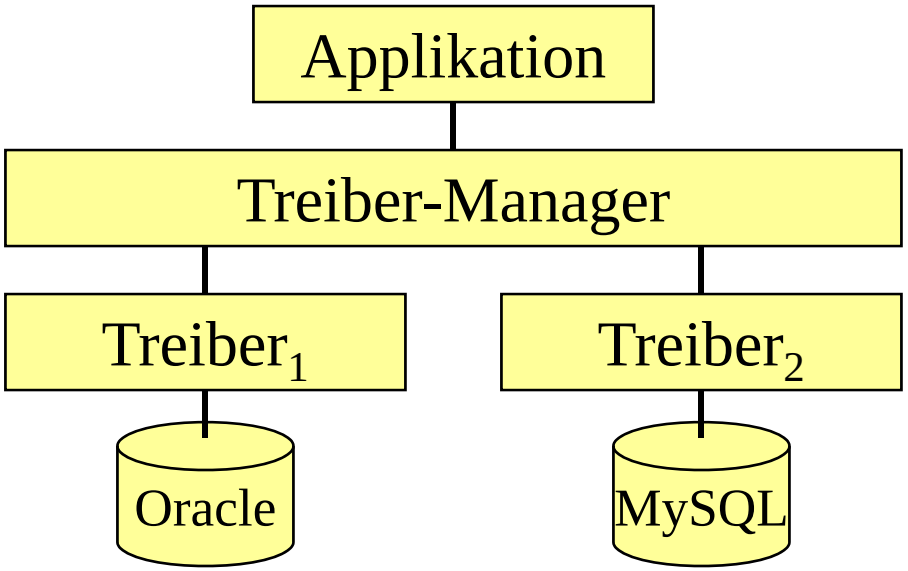
Freigabe von Ressourcen (z.B. Ergebnistabelle in der DB)

Weitere Aufgaben

- Über das reine Absetzen von Anfragen hinaus sind weitere Interaktionen mit dem DBS nötig:
  - Verwalten der Verbindung:
    - Verbindungsaufbau zu einer oder mehreren DBs
    - Verbindungsabbau
    - Authentifizierung mit Benutzername/Passwort
  - Management von Transaktionen
    - commit
    - rollback
  - Fehlerbehandlung
  - Aufruf von gespeicherten Prozeduren sowie deren Verwaltung

Java Database Connectivity JDBC

- Call Level Interface (Bibliotheksfunktionen)
- Eng verwandt mit Microsofts ODBC
  - ähnliches Konzept für verschiedene Wirtssprachen
- Idee:
  Hersteller-Unabhängigkeit durch Treiber



Verbindungsaufbau

- Der prinzipielle Ablauf einer DB-Anwendung umfasst die folgenden Einzelschritte:
  - Laden des geeigneten Treibers
  - Aufbau einer Verbindung zur Datenbank
  - Senden einer SQL-Anweisung
  - Verarbeiten der Anfrageergebnisse
- Diesen Schritten sind jeweils Java-Klassen zugeordnet:
  - `java.sql.DriverManager`
  - `java.sql.Connection`
  - `java.sql.Statement`
  - `java.sql.ResultSet`

Treiber und Verbindungsaufbau

- Es muß also java.sql.* importiert werden.
- Laden des Datenbank-Treibers für Oracle:
  Class.forName ("oracle.jdbc.driver.OracleDriver") ;
- Einrichten einer Verbindung durch Treiber-Mgr:

  Connection con = DriverManager.getConnection (
  "jdbc:oracle:oci8:@dbis01", "scott", "tiger") ;

  richtet folgende Verbindung ein:
  - Datenbankname: dbis01
  - Benutzername: scott
  - Passwort: tiger
- Connection-Objekt (hier: *con*) ist Ausgangspunkt für alle weiteren Aktionen

Änderungs-Anweisungen

- Ausgangspunkt: das Connection-Objekt
- Zunächst muß ein Statement-Objekt erzeugt werden (dient für Verwaltungszwecke)
- insert, delete und update mittels der Methode executeUpdate:

```
Statement stmt = con.createStatement () ;
int rows = stmt.executeUpdate
        (“update Produkt set Bestand=5 where ProdID=3“) ;
```

Transaktionssteuerung:

- Nach Verbindungsherstellung Auto-Commit-Modus (jede Änderung wird ohne explizites Commit sofort permanent)
- später Näheres

Retrieval-Anweisungen

- Ausgangspunkt: Wieder Connection-Objekt
- Für Verwaltungszwecke: Statement-Objekt
- Ein Cursor wird deklariert *und* geöffnet
  durch die Methode executeQuery (query):

```
Statement stmt = con.createStatement () ;
ResultSet rs = stmt.executeQuery ("select * from Mitarbeiter") ;
```

- Das Rückgabeobjekt (hier *rs*) vom Typ ResultSet ist aber noch nicht die Ergebnistabelle selbst sondern eine ID für den entsprechenden Cursor
- Um 1. (2., 3,...) Datensatz zu laden, Methode next:

```
while (rs.next ( ) ) {
        /* Datensatz verarbeiten */
}
```

false, wenn kein Datensatz (mehr) vorhanden ist

Schließen in JDBC implizit durch Garbage Collection

Retrieval-Anweisungen

- Zugriff auf die einzelnen Attribute durch Spalten-Indizes (beginnend bei 1) über die Funktionen:
  - getString (*i*)
  - getDouble (*i*)
  - getInt (*i*) usw.

  die einen Wert des entsprechenden Typs liefern.
- Beispiel:

```
Statement stmt = con.createStatement ( ) ;
ResultSet rs = stmt.executeQuery (“select * from Mitarbeiter“) ;
while (rs.next ( ) ) {
        int pnr = rs.getInt (1) ;
        String name = rs.getString (2) ;
        double gehalt = rs.getDouble (5) ;
        System.out.println (pnr + “ “ + name + “ “ + gehalt) ;
}
```

Fehlerbehandlung

- Datenbankanweisungen können fehlschlagen, z.B.
  - Connect mit einer nicht existenten Datenbank
  - Select auf eine nicht existente Tabelle usw.
- In Java generell mit folgendem Konstrukt:

```
try {
        // Aufruf von JDBC-Anweisungen,
        // die möglicherweise Fehler generieren
} catch (SQLException exc) {
        // Fehlerbehandlung, z.B.
        System.out.println (exc.getMessage ( ) ) ;
}
```

Vorübersetzte SQL-Anfragen

- Bisher:
  Jede Anfrage wird als String an DBMS übergeben
  - Vorteil: Flexibel
  - Nachteil: Jedes mal Übersetzungsaufwand
- Werden oft ähnliche Anfragen gestellt, empfiehlt sich, auf wiederholte Übersetzung zu verzichten
  - Parametrisierbarkeit ist wichtig

```
PreparedStatement stmt = con.prepareStatement
        ( " insert into Mitarbeiter values (?, ?, ?, NULL, 0.0) " );
```

- Die einzelnen Platzhalter müssen vor dem Ausführen mit SetInt, SetString, ... besetzt werden
  - Spaltenindizes (beginnend bei 1)

Vorübersetzte SQL-Anfragen

- Beispiel:

```
PreparedStatement stmt = con.prepareStatement
        ( " insert into Mitarbeiter values (?, ?, ?, NULL, 0.0) " );
// erstes Insert:
stmt.setInt (1, 25) ;
stmt.SetString (2, "Wagner") ;
stmt.SetString (3, "Richard") ;
stmt.executeUpdate ( ) ;
// zweites Insert:
stmt.SetInt (1, 26) ;
...
stmt.executeUpdate ( ) ;
```