



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/24

# Kapitel 1: Einführung

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani  
Skript © 2019 Christian Böhm, LMU



# Das Team

## Vorlesung



Prof. Dr. Thomas Seidl

## Übungsbetrieb



Collin Leiber

[leiber@dbs\\_ifi.lmu.de](mailto:leiber@dbs_ifi.lmu.de)

Walid Durani  
[durani@dbs\\_ifi.lmu.de](mailto:durani@dbs_ifi.lmu.de)

## Tutoren/Korrektoren:

Fabian Porsche,  
Patrick Lackner,  
Lea Pietsch,  
Silvan Hecht,  
Jinke Li,  
Karl Deck,  
Ksenija Beser,  
Remo Kötter,  
Magdalena Mansfeld



# Vorlesungs-Websiten

- Moodle
  - Ausgabe und Abgabe der Übungsblätter
  - Anmeldung Klausur
  - Vorlesungsinformationen
  - Vorlesungsfolien



# Literaturliste

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

- A. Kemper, A. Eickler:  
**Datenbanksysteme**  
DeGruyter, 10. Auflage (2015). 49,95 €
- R. Elmasri, S. B. Navathe:  
**Grundlagen von Datenbanksystemen**  
Pearson Studium, 3. Auflage (2004). 39,95 €
- R. Elmasri, S. B. Navathe:  
**Fundamentals of Database Systems**  
Addison-Wesleys (2011).
- 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).





# Wovon handelt die Vorlesung?

- Bisher (Einführungsvorlesung):  
Nur Betrachtung des Arbeitsspeichers.  
Objekte werden im Arbeitsspeicher erzeugt und nach dem Programmlauf wieder entfernt
- Warum ist diese Sichtweise nicht ausreichend?
  - Anwendungen müssen Daten **persistent** speichern
  - Arbeitsspeicher ist häufig nicht **groß** genug, um z.B. alle Kundendaten einer Bank oder Patientendaten einer Klinik zu speichern
- Aus Vorlesung TGI/Betriebssysteme: Virtueller Speicher
  - Aus unserer Sicht nicht ausreichend; Notwendigkeit, die Speicherung „von der Persistenz her zu denken“



# Persistente Datenspeicherung

- Daten können auf dem sog. *Externspeicher* (Festplatte) persistent gespeichert werden
- Arbeitsspeicher:
  - rein elektronisch (Transistoren und Kondensatoren)
  - flüchtig
  - schnell: 10 ns/Zugriff \*
  - wahlfreier Zugriff
  - teuer:  
59,90 € für 16 GByte\*  
(3,70€/Gbyte)
- Externspeicher:
  - Speicherung auf magnetisierbaren Platten (rotierend)
  - nicht flüchtig
  - langsam: 5 ms/Zugriff \*
  - blockweiser Zugriff
  - wesentlich billiger:  
102,-- € für 3000 GByte\*  
(3 ct/Gbyte)

\*Oktober 2013



# Beispiele

## Hardware

- » Arbeitsspeicher
- » Bürotechnik
- » Cooling
- » Drucker & Scanner
- » Eingabegeräte
- » Festplatten

- » IDE
- » SATA
- 1,8 Zoll
- 2,5 Zoll
- 3,5 Zoll

- » SCSI
- » SAS
- » USB

- » FireWire

eSATA

Thunderbolt

Solid State Drives

Multimedia

- » Zubehör

- » Gehäuse

- » Grafikkarten

- » Kabel

- » Laufwerke

- » Mainboards

- » Monitore

- » Netzwerktechnik

- » PC-Audio

Hardware > Festplatten > SATA > HDS723020BLA642 2 TB

## Hitachi HDS723020BLA642 2 TB

(SATA 600, Deskstar 7K3000, 24/7)

HITACHI



- » Kapazität: 2000 GB
- » ms/Cache/U: -/64/7200
- » Preis pro GB: € 0,04\*

★★★★★ 40 Bewertungen lesen | bewerten

€ 87,90\*

Auf Lager

In den Warenkorb ▶

Abb. kann vom Original abweichen

| Details | Bewertungen | Preisentwicklung | Zubehör (64) | Video |
|---------|-------------|------------------|--------------|-------|
|---------|-------------|------------------|--------------|-------|

Die DeskStar 7K3000 ist die erste Festplattenserie von Hitachi, die eine enorme Kapazität von bis zu 2 TB mit einer hohen Performance dank 7.200 U/min, 64 MB Cache und den schnellen SATA/6Gb/s-Interface verbindet. Die HDS723020BLA642 ist das 2 TB fassende Modell der Serie und ist mit einer Sektorgroße von 512 Byte auch zu älteren Systemen kompatibel. Die HDS723020BLA642 ist für den 24/7-Dauereinsatz geeignet und stellt damit eine gute Wahl für Datenbanken und Server-Systeme dar.

### Neueste User-Bewertungen



Leise, flott, gutes...

von Coolblue1978

am 12.10.2011



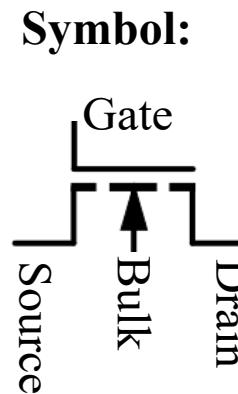
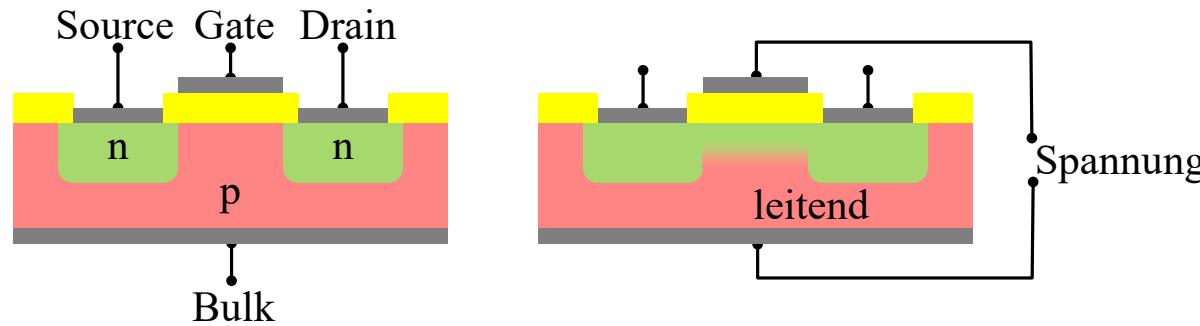
Mahlzeit, habe mir diese FP...

von Computer-chaos



# Technologie des Hauptspeichers

- (Feldeffekt-) Transistor:  
Elektronischer Schalter bzw. Verstärker  
(mit einer schwachen Spannung kann eine höhere  
Spannung gesteuert werden)

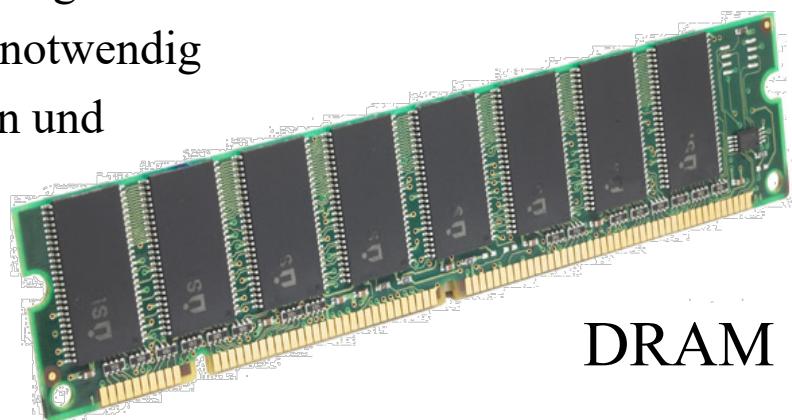
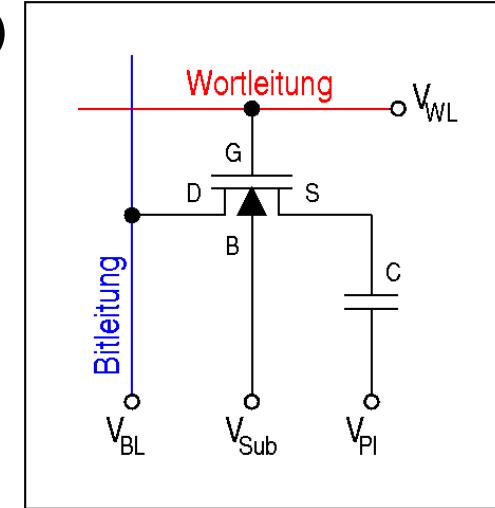


- Der Bereich zwischen Source und Drain ist normal nicht leitend
- Durch Anlegen einer Spannung zwischen Bulk und Gate wandern Elektronen in diesen Bereich und dieser wird leitend („geschlossener Schalter“)



# Haupt- bzw. Arbeitsspeicher

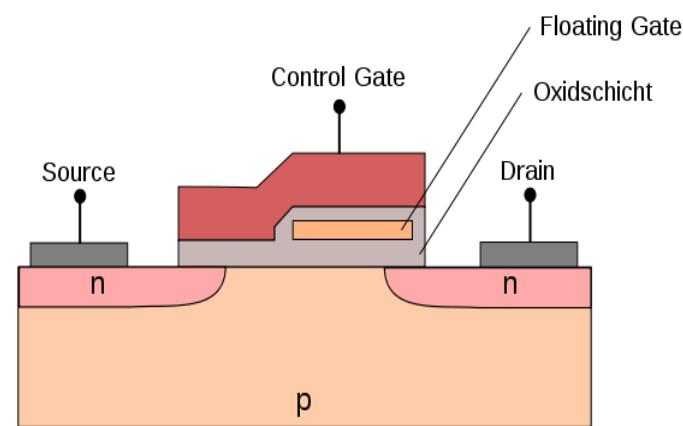
- DRAM (Dynamic Random Access Memory)
- Nur zwei Bauteile pro Bit:
  - 1 Feldeffekttransistor zur Steuerung
  - 1 Kondensator („Akku“ zur Speicherung von Ladung)
- Arbeitsweise:
  - Kondensator kann nur sehr wenig Ladung aufnehmen und diese nur kurzzeitig speichern
  - Wird eine Spannung auf die Wortleitung gegeben, dann kann der Speicherinhalt auf der Bitleitung ausgelesen werden.
  - Verstärker mit Zwischenspeicher notwendig
  - Inhalt muss regelmäßig ausgelesen und wieder zurückgespeichert werden (Refresh)





# Flash-Memory

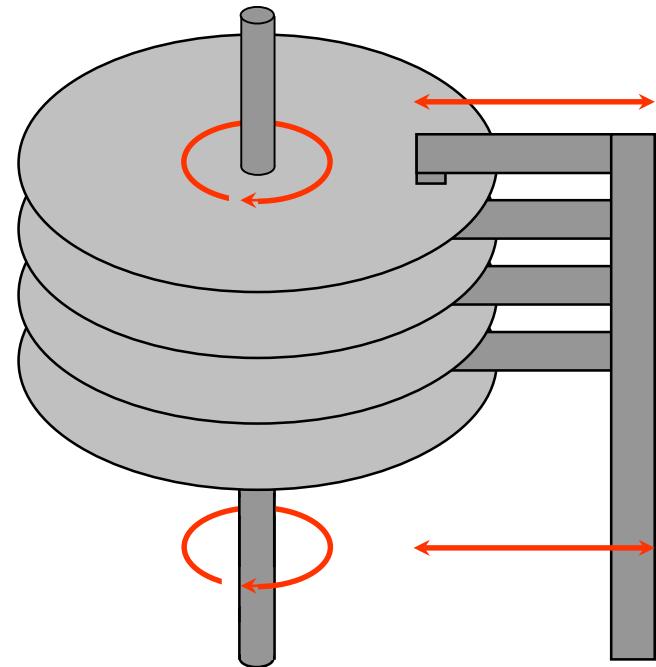
- Flash-Memory
  - Eingesetzt in Memory Sticks, Speicherkarten und sog. Solid State Disks (SSD)
  - Nicht-flüchtige Halbleiterspeicher
- Arbeitsweise
  - Gate von einer Isolationsschicht umgeben („floating gate“)
  - Durch Anlegen einer hohen Spannung können Elektronen eingebracht oder wieder herausgenommen werden („Tunneleffekt“)
- Eigenschaften:
  - Begrenzte Zahl von Schreibzyklen (100.000)
  - Daher nicht verwendbar als Arbeitsspeicher
  - + Sehr geringer Platzbedarf (geringer als DRAM)
  - + Deutlich schneller und robuster als Festplatten





# 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.





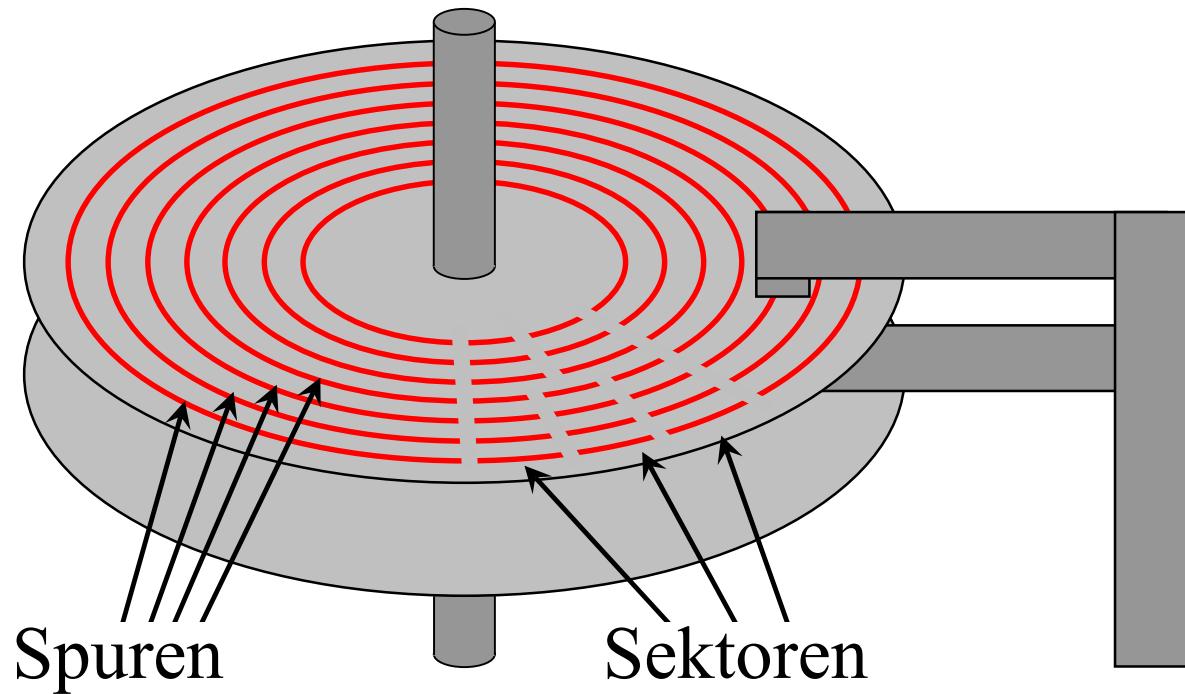
# Demonstration Festplatte

Datenbanksysteme  
Kapitel 1: Einführung





# Einteilung der Plattenoberflächen



- (interne) Adressierung einer Information:  
[Platten-Nr | Oberfl.-Nr | Spur-Nr | Sektor-Nr | Byte-Nr]



# 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



# Demonstration Schreiben/Lesen

Datenbanksysteme  
Kapitel 1: Einführung





# Zugriffszeit auf einen Sektor

Zugriffszeit für Schreib-/Lese-Auftrag zusammengesetzt aus:

- **Suchzeit** (seek time) zur Positionierung des Kamms:  
typisch 3 ms
- **Latenzzeit** (latency time): Wartezeit wegen Rotation
  - maximal eine Umdrehung, also bei 7200 UPM: 8.3 ms
  - Im Durchschnitt die Hälfte, 4.1 ms  
(Annahme, dass Zeit zwischen zwei Aufträgen zufällig ist -- Poisson-Verteilung)
- **Transferzeit** (transfer time)  
Abhängig von der Länge des Sektors, bzw. es ist auch die Übertragung mehrerer Sektoren in einem Auftrag möglich
  - typische Transferrate: 200 Mbyte/s
  - 1 Sektor à 512 Bytes: 2,5 µs

$$\begin{aligned} 7200 \text{ Umdr./min} &= \\ &= 120 \text{ Umdr./sec}; \\ 1/120 &= 0.0083 \end{aligned}$$



# 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 durchnummieriert.
  - 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"); ← Datei öffnen  
        int c = f1.read(); ← ein Byte lesen  
        long new_position = ....;  
        f1.seek (new_position); ← auf neue Position  
        f1.write (c); ← ein Byte schreiben  
        f1.close (); ← Datei schließen  
    } catch (IOException e) { ← Fehlerbehandlung  
        System.out.println ("Fehler: " + e);  
    }  
}
```

Annotations in red:

- A red arrow points from the text "Datei-Handle" to the line "RandomAccessFile f1 = new".
- A red arrow points from the text "Datei öffnen" to the line "RandomAccessFile ("test.file", "rw");".
- A red arrow points from the text "ein Byte lesen" to the line "int c = f1.read();".
- A red arrow points from the text "auf neue Position" to the line "f1.seek (new\_position);".
- A red arrow points from the text "ein Byte schreiben" to the line "f1.write (c);".
- A red arrow points from the text "Datei schließen" to the line "f1.close ();".
- A red arrow points from the text "Fehlerbehandlung" to the line "catch (IOException e) {".



# Beispiel: Dateizugriff in Java

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

| Name (10 Zeichen) | Vorname (8 Z.) | Jahr (4 Z.) |
|-------------------|----------------|-------------|
| F r a n k l i n   | A r e t h a    | 1 9 4 2     |

- 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



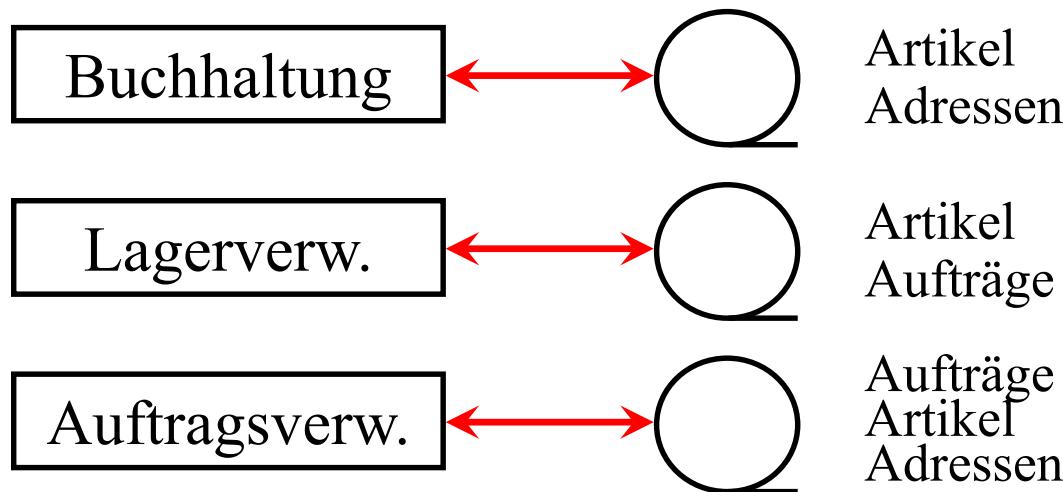
# Mangelnde „Datenunabhängigkeit“

- 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



# Redundanzproblem bei Dateien

- Daten werden mehrfach gespeichert

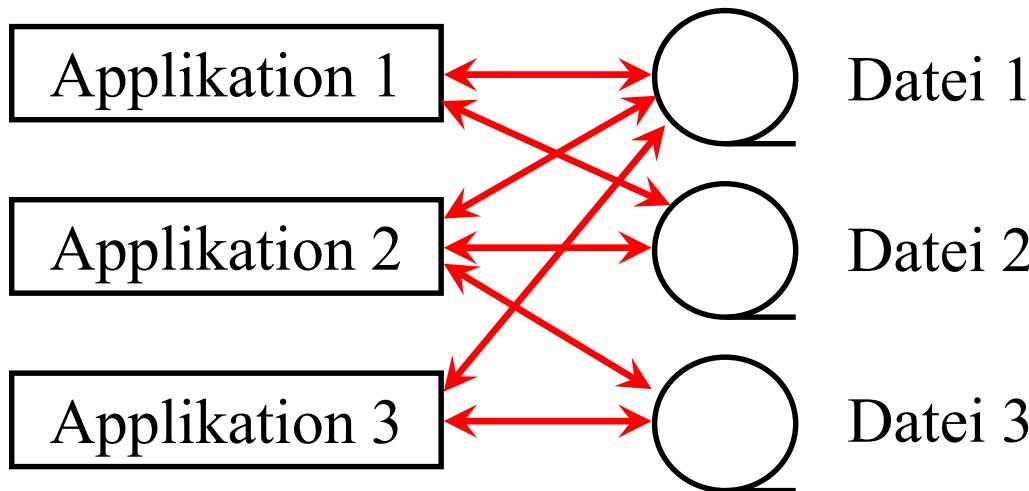


- Konsequenz: u.a. *Änderungs-Anomalien*  
Bei Änderung einer Adresse müssen viele Dateien nach den Einträgen durchsucht werden  
(hierzu später mehr)



# Schnittstellenproblematik

- Alternative Implementierung



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



# Weitere Probleme von Dateien

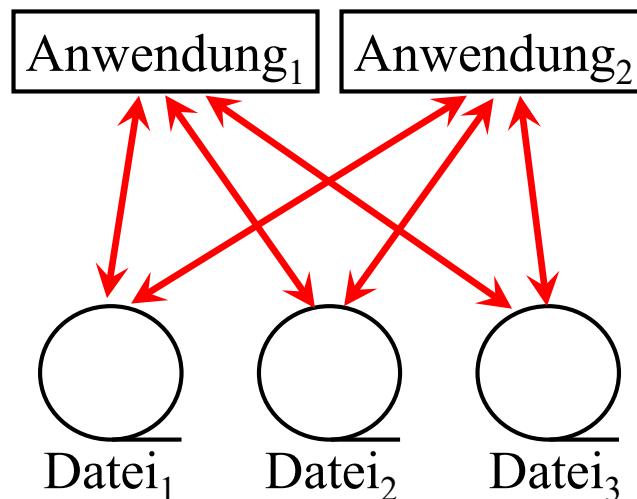
- In großen Informationssystemen arbeiten viele Benutzer gleichzeitig mit den Daten: Dateisysteme bieten zu wenige Möglichkeiten, um diese Zugriffe zu synchronisieren
- Dateisysteme schützen nicht in ausreichendem Maß vor Datenverlust im Fall von Systemabstürzen und Defekten
- Dateisysteme bieten nur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



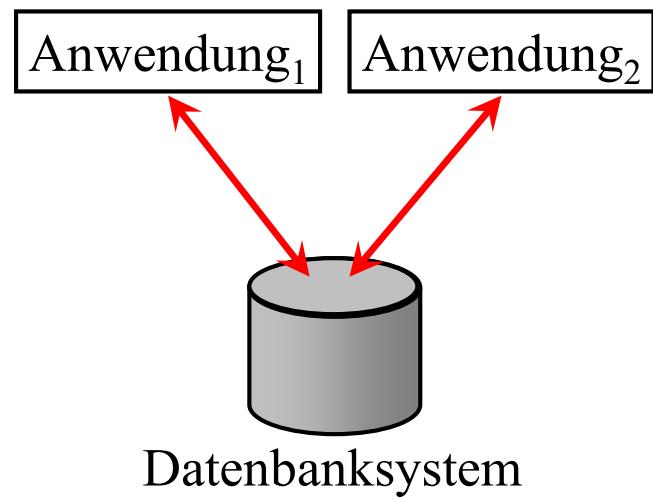
# Von Dateien zu Datenbanken

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

Mit Dateisystem:



Mit Datenbanksystem:



- Einheitliche Speicherung **aller** Daten, die z.B. in einem Betrieb anfallen



# Komponenten eines DBS

Man unterscheidet zwischen...

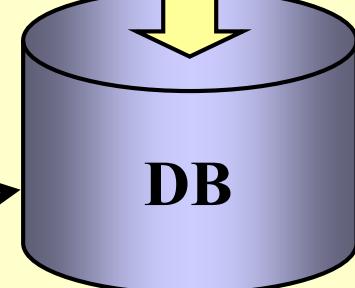
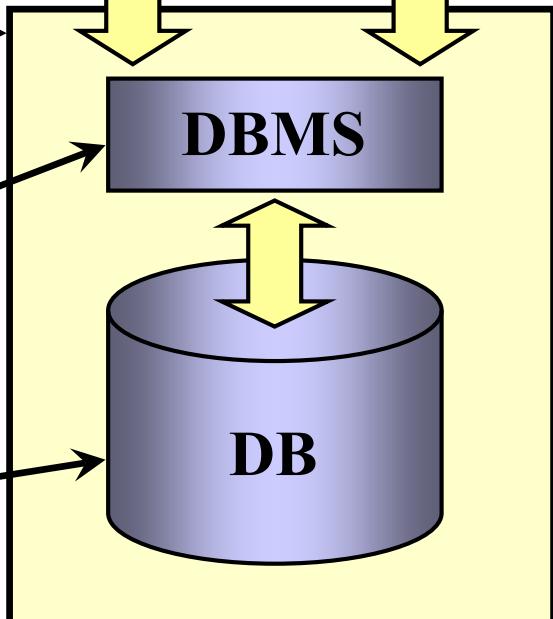
## DB-Anwendungen

(kommunizieren mit DBMS)



## DBS: Datenbanksystem

(DB + DBMS)



## DBMS: Datenbank-Management-System

(Software zur Verwaltung)



## DB: Datenbank

(eigentliche Datensammlung)





# Aufgaben eines DBS

Primäre Aufgabe eines DBS ist die ...

- Beschreibung
- Speicherung und Pflege
- und Wiedergewinnung

umfangreicher Datenmengen, die von verschiedenen Anwendungsprogrammen 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 zwischen:

- **Datenbankschema**

- beschreibt *möglichen* Inhalt der DB
- Struktur- und Typinformation der Daten (Metadaten)
- Art der Beschreibung vorgegeben durch Datenmodell
- Änderungen möglich, aber selten (Schema-Evolution)

- **Ausprägung** der Datenbank bzw. **Datenbank-Zustand**

- *Tatsächlicher, eigentlicher* Inhalt der DB
- Objektinformation, Attributwerte
- Struktur vorgegeben durch Datenbankschema
- Änderungen häufig (Flugbuchung: 10000 TA/min)



# Inhalte von Datenbanken

Einfaches Beispiel:

- Schema:

| Name (10 Zeichen) | Vorname (8 Z.) | Jahr (4 Z.) |
|-------------------|----------------|-------------|
|                   |                |             |

- 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 |   |   |   |

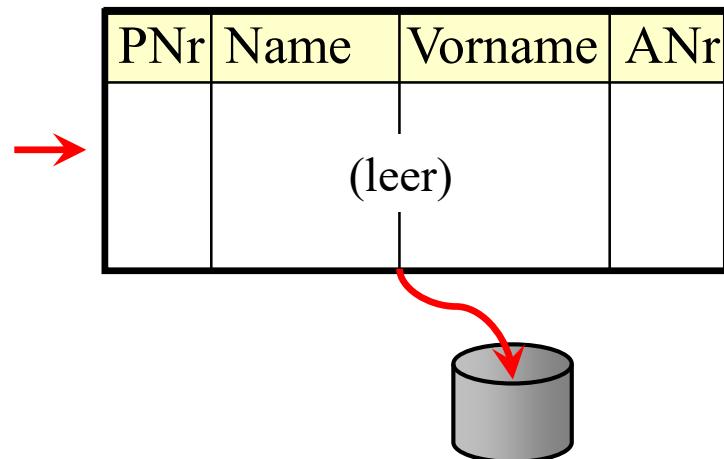
- Bei Schema-Änderung müssen Migrations-Prozeduren programmiert werden, um bestehende Dateien auf das neue Format umzustellen



# 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-Tabelle S. 33)

```
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  |

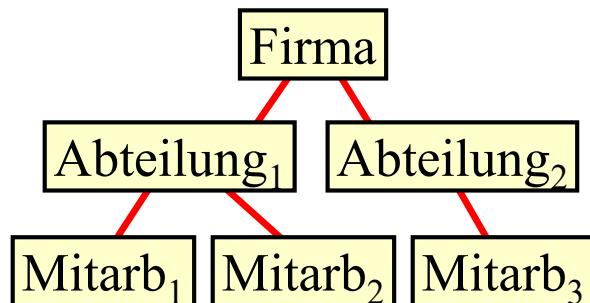
Ergebnis einer Anfrage ist immer eine (neue) Tabelle



# 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

# Abteilungen



# 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
  - („NoSQL-Datenbanken“)



# Relationales Modell



# 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

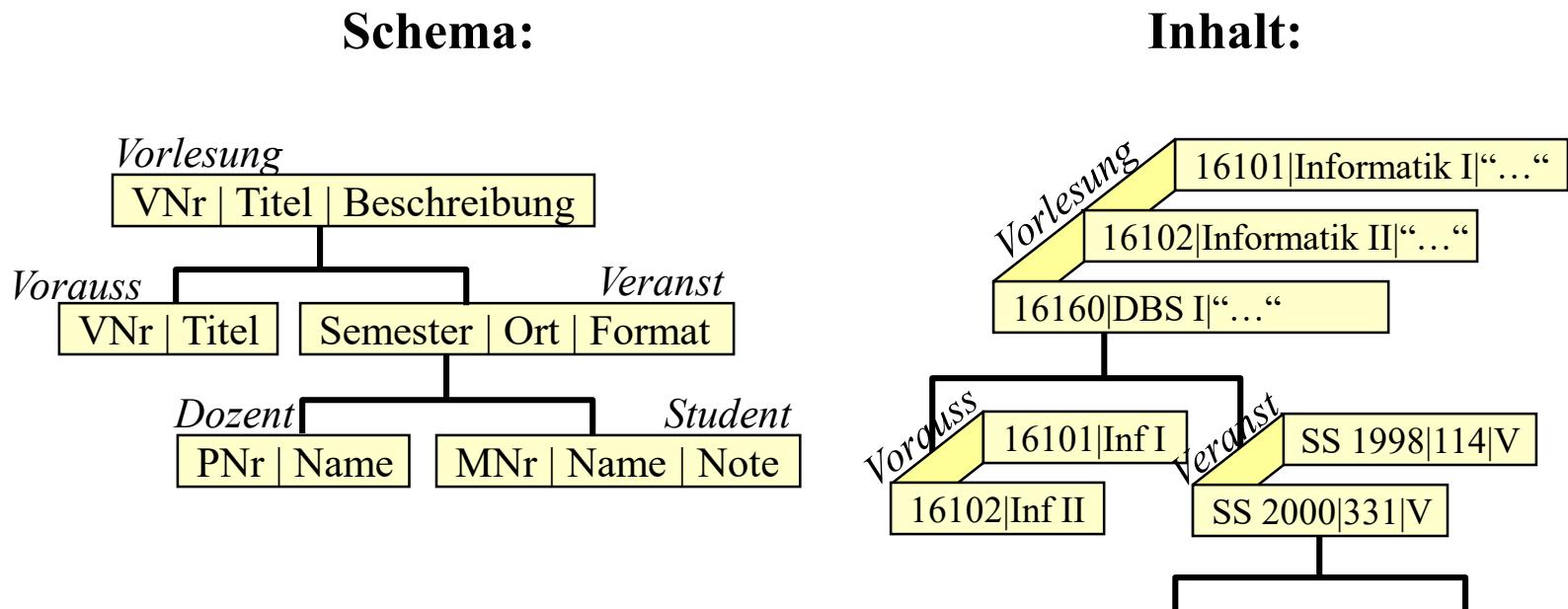
| ANr | Abteilungsname |
|-----|----------------|
| 01  | Buchhaltung    |
| 02  | Produktion     |
| 03  | Marketing      |

- Ausführliche Behandlung im nächsten Kapitel



# Hierarchisches Datenmodell

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





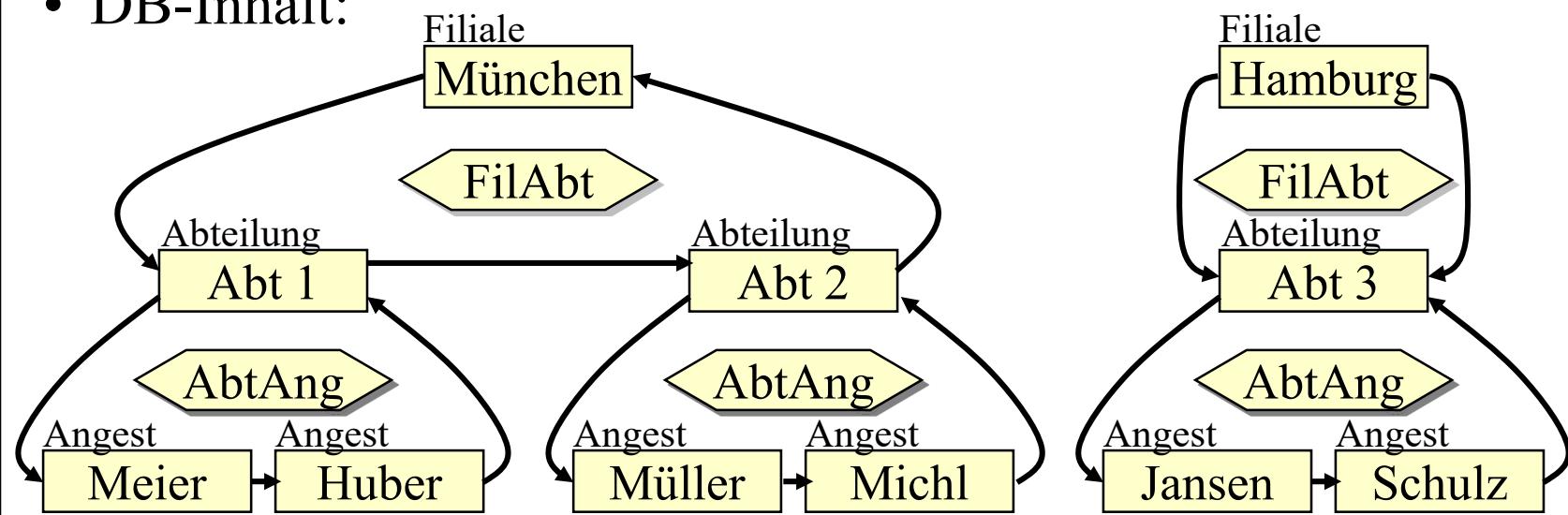
# Netzwerk-Datenmodell

- 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

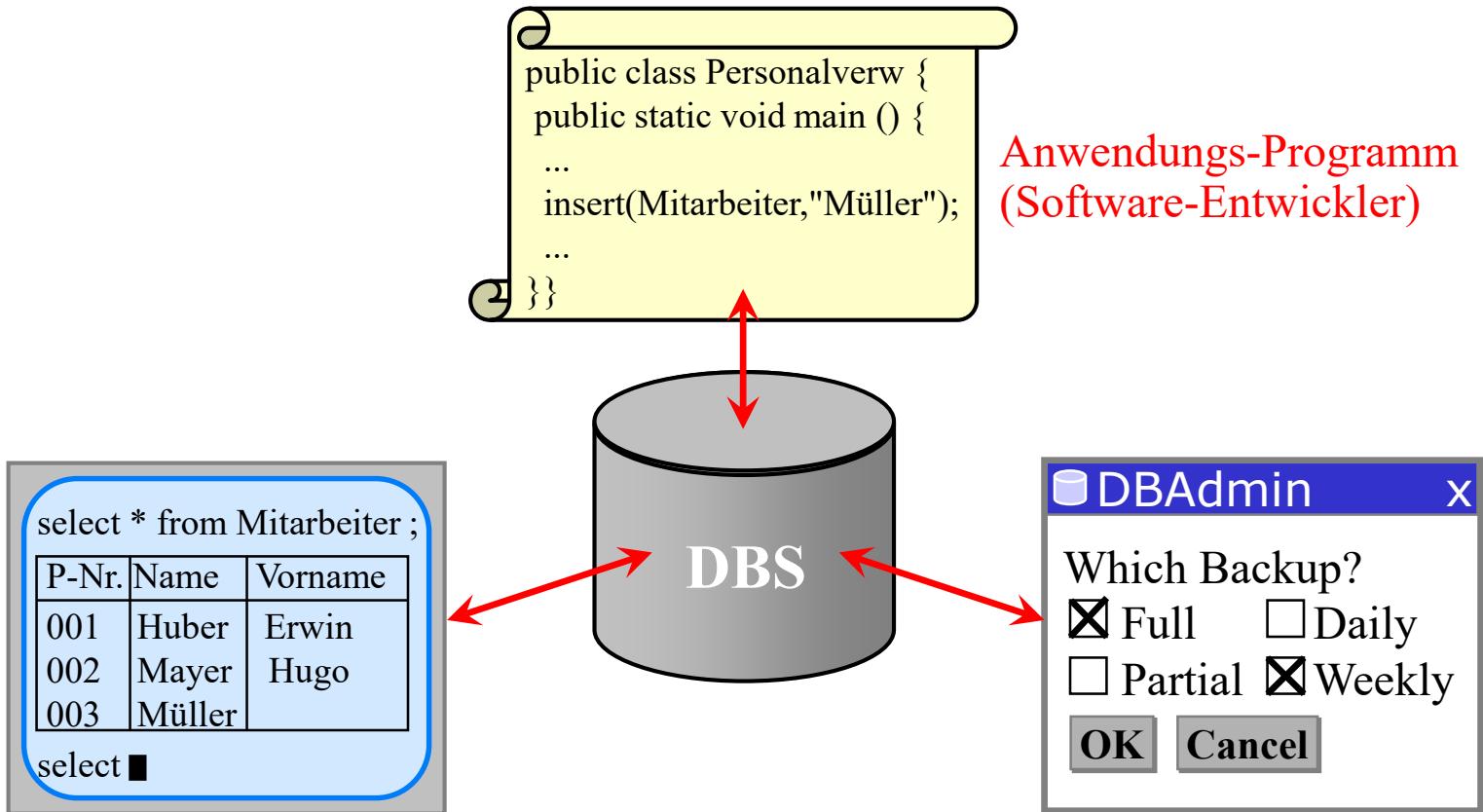


# NoSQL Datenbanken

- Sammelbegriff für viele neuere Entwicklungen, u.a.
  - Dokumentorientierte Speichersysteme (MongoDB)
  - Graph-Datenbanken
  - Key-Value-Datenbanken
- Unterstützen meist laufzeitkritische Anwendungen
- Oft eingeschränkte Konsistenz-Überwachung



# Verwendung eines DBS



Interaktive Oberfläche  
für Ad-Hoc-Anfragen

Benutzung vorgege-  
bener Anwendungen

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



# 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:

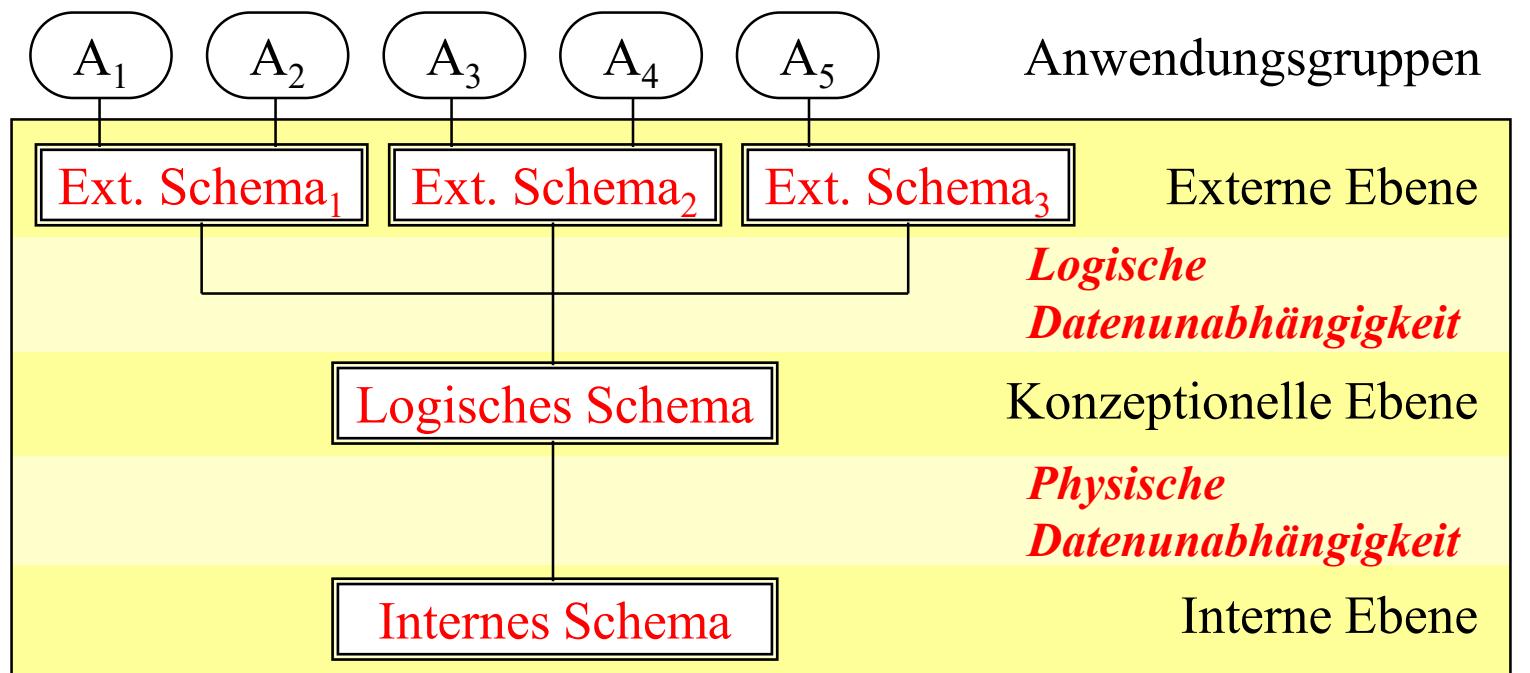
```
public static void main () {  
    System.out.println ("Hallöchen") ;  
    #sql {SELECT * FROM Mitarbeiter  
          WHERE Name = 'Müller'}  
    ...  
}
```

- 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)



# Überblick über die Vorlesung

| Kapitel | Inhalt  |
|---------|---|
| 1       | Einführung                                    |
| 2       | Das Relationale Datenmodell                   |
| 3       | Die Relationale Algebra, (SQL, Teil 1)        |
| 4       | Tupel- und Bereichskalkül (SQL, Teil 2)       |
| 5       | Sortieren, Gruppieren und Views (SQL, Teil 3) |
| 6       | Datenbank-Modellierung mit dem E/R-Modell     |
| 7       | Normalformen                                  |
| 8       | Transaktionen                                 |
| 9       | Index-Strukturen                              |
| 10      | Relationale Anfragebearbeitung                |
| 11      | Anwendungsentwicklung                         |



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 2: Das Relationale Modell

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani  
Skript © 2019 Christian Böhm



# Charakteristika

- Einführungskapitel:  
Viele Informationen darstellbar als Tabelle
- Die Tabelle (Relation) ist das ausschließliche Strukturierungsmittel des relationalen Datenmodells
- Edgar F. Codd, 1970.  
*A relational model of data for large shared data banks.* Comm. of the ACM 13.06.1970
- Grundlage vieler kommerzieller und freier DBS:

**ORACLE**



DB2 / Informix



**MySQL**



PostgreSQL



# Domain

- Ein Wertebereich (oder Typ)
- Logisch zusammengehörige Menge von Werten
- Beispiele:
  - $D_1 = \text{Integer}$
  - $D_2 = \text{String}$
  - $D_3 = \text{Date}$
  - $D_4 = \{\text{rot, gelb, grün, blau}\}$
  - $D_5 = \{1, 2, 3\}$
- Kann *endliche* oder *unendliche* Kardinalität  $|...|$  haben:
  - $|D_4| = 4; |D_5| = 3;$
  - $|D_1| = \text{unendlich};$  ebenso  $|D_2|$  und  $|D_3|.$



# Kartesisches Produkt

- Bedeutung kartesisches Produkt (Kreuzprodukt) von  $k$  Mengen?  
**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 ( $k = 3$ ):  
 $D_1 = D_2 = D_3 = \mathcal{N}$   
 $D_1 \times D_2 \times D_3 = \{(1,1,1), (1,1,2), (1,1,3), \dots, (1,2,1), \dots\}$



# Relation in der Mathematik

- Mathematische Definition:

Relation  $R$  ist Teilmenge des kartesischen Produktes von  $k$  Domains  $D_1, D_2, \dots, D_k$

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

- Beispiel ( $k = 2$ ):

$$D_1 = \{1, 2, 3\}, D_2 = \{a, b\}$$

$$R_1 = \{\} \text{ (leere Menge)}$$

$$R_2 = \{(1, a), (2, b)\}$$

$$R_3 = \{(1, a), (2, a), (3, a)\}$$

$$R_4 = D_1 \times D_2 = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$



# Relation in der Mathematik

- Weiteres Beispiel:

$$D_1 = D_2 = \mathcal{N}$$

$$\text{Relation } R_1 = \{(1,1), (1,2), (1,3), \dots, (2,2), (2,3), \dots, \\ (3,3), (3,4), \dots, (4,4), (4,5), (4,6), \dots\}$$

Wie heißt diese mathematische Relation?



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

- Es gibt endliche und unendliche Relationen  
(wenn mindestens eine Domain unendlich ist).
- In Datenbanksystemen: Nur endliche Relationen  
Unendlich: Nicht darstellbar .
- Die Anzahl der Tupel einer Relation heißt *Kardinalität* [...]



# Relation in der Mathematik

- Die einzelnen Domains lassen sich als **Spalten einer Tabelle** verstehen und werden als **Attribute** bezeichnet
- Für  $R \subseteq D_1 \times \dots \times D_k$  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:  
 $\{(a,0), (b,1)\} \neq \{(0,a), (1,b)\}$



# Relationen-Schema

Alternative Definition in DBS:  
Relation ist Ausprägung eines **Relationen-Schemas**.

- Geordnetes Relationenschema:
  - $k$ -Tupel aus Domains (Attribute)
  - Attribute werden anhand ihrer **Position** im Tupel referenziert
  - Attribute können zusätzlich einen Attributnamen haben
- 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

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



# 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:  $R = (\text{Name}: \text{String}, \text{Einwohner}: \text{Integer}, \text{Land}: \text{String})$

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

- Als Relationenschema mit Domänenabbildung:

Schema:  $R = \{\text{Name}, \text{Einwohner}, \text{Land}\}$

mit  $\text{dom}(\text{Name}) = \text{String}$ ,  $\text{dom}(\text{Einwohner}) = \text{Integer}$ , ...

Ausprägung:  $r = \{t_1, t_2, t_3\}$

mit  $t_1(\text{Name}) = \text{München}$ ,  $t_1(\text{Einwohner}) = 1.211.617, \dots$



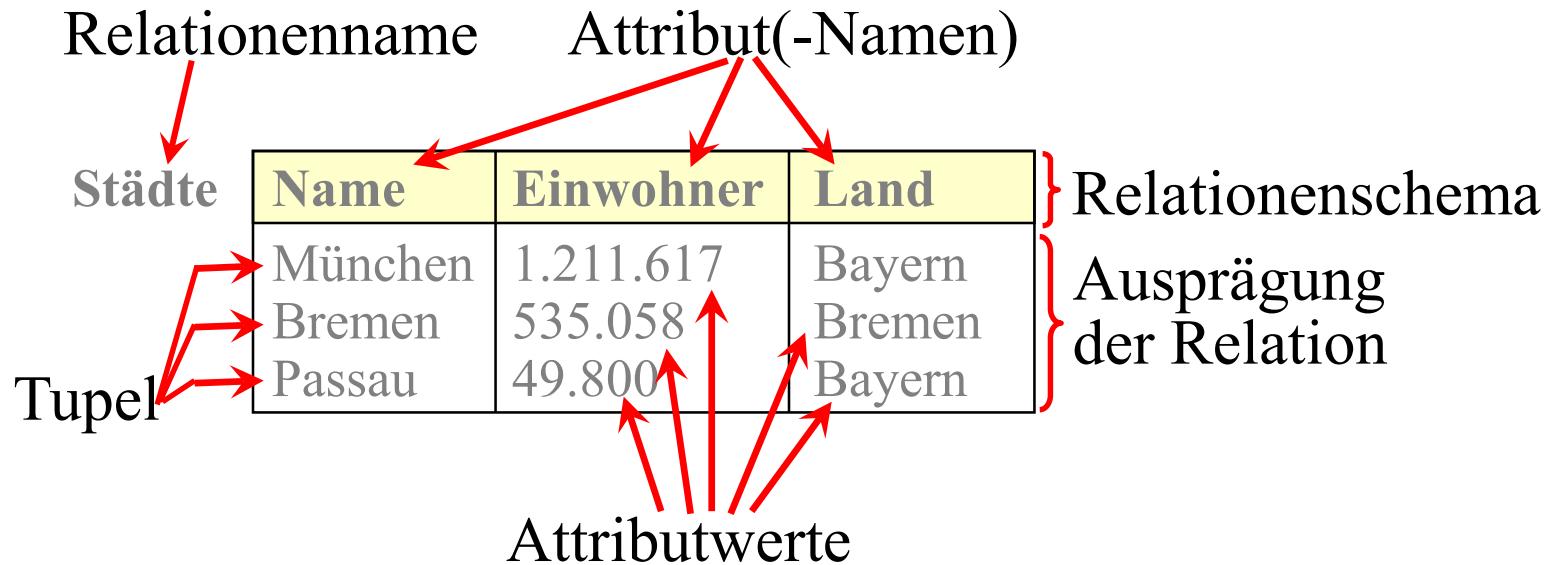
# Diskussion

- Vorteil von geordnetem Relationenschema:
  - Prägnanter aufzuschreiben.  
Wichtig z.B. beim Einfügen neuer Tupel:  
 $t_3 = (\text{Passau}, 49.800, \text{Bayern})$   
vergleiche:  $t_3(\text{Name}) = \text{Passau}$ ;  $t_3(\text{Einwohner}) = \dots$
- Nachteil von geordnetem Relationenschema:
  - Einschränkungen bei logischer Datenunabhängigkeit:  
Applikationen sensibel bzgl. Einfügung neuer Attribute (nur am Ende!)
- Definitionen prinzipiell gleichwertig
- 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  $R = \{A_1, \dots, A_k\}$  ?
  - Reihenfolge der Spalten ist **irrelevant**  
(das ist gerade das besondere am ungeordneten RS)
  - Duplikate **treten nicht auf, weil alle Attribut-Namen verschieden sein müssen**



# 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

| <b><u>PNr</u></b> | Name   | Vorname | Abteilung |
|-------------------|--------|---------|-----------|
| 001               | Huber  | Erwin   |           |
| 002               | Mayer  | Hugo    |           |
| 003               | Müller | Anton   |           |

Abteilungen

| <b><u>ANr</u></b> | Abteilungsname |
|-------------------|----------------|
| 01                | Buchhaltung    |
| 02                | Produktion     |
| 03                | Marketing      |

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

Mitarbeiter

| <b><u>PNr</u></b> | Name   | Vorname | Abteilung |
|-------------------|--------|---------|-----------|
| 001               | Huber  | Erwin   | 01        |
| 002               | Mayer  | Hugo    | 01        |
| 003               | Müller | Anton   | 02        |

Abteilungen

| <b><u>ANr</u></b> | Abteilungsname |
|-------------------|----------------|
| 01                | Buchhaltung    |
| 02                | Produktion     |
| 03                | Marketing      |



# Zusammengesetzter Schlüssel

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

| Lehrveranstaltung | <u>VNr</u> | <u>Titel</u>                   | <u>Semester</u> |
|-------------------|------------|--------------------------------|-----------------|
|                   | 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.**  
**→ hierzu mehr in Kapitel 6+7**



# Schlüssel: Formale Definition

Definition:

- Eine Teilmenge  $S$  der Attribute eines Relationenschemas  $R$  ( $S \subseteq R$ ) heißt **Schlüssel**, wenn gilt:

## 1) Eindeutigkeit

Keine Ausprägung von  $R$  kann zwei verschiedene Tupel enthalten, die sich in **allen** Attributen von  $S$  gleichen.

## 2) Minimalität

Es existiert keine **echte** Teilmenge  $T \subsetneq S$ , die bereits die Bedingung der Eindeutigkeit erfüllt.

Anm.: Der Teilmengenbegriff umfasst die Menge selbst, also jede Menge ist Teilmenge von sich selbst. Eine Teilmenge einer Menge  $S$ , die ungleich  $S$  ist, heißt *echte* Teilmenge. In Symbolen:  $T \subsetneq S \Leftrightarrow T \subseteq S \wedge T \neq S$



# Schlüssel: Formale Definition

Manche Lehrbücher definieren in noch formalerer Notation:

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:

$\forall$  Attributmengen  $T$ , die (1) erfüllen, gilt:

$$T \subseteq S \Rightarrow T = S.$$

Hierbei bezeichne  $t[S]$  ein Tupel  $t$  eingeschränkt auf die Attribute aus  $S$  (alle anderen Attribute gestrichen).

Wir schreiben später auch  $\pi_S(t)$  für  $t[S]$  (*Projektion*, s. Kap. 3)



# Superschlüssel / Minimale Menge

- Eine Menge  $S \subseteq R$  heißt Superschlüssel (oder Oberschlüssel, engl. Superkey), wenn sie die Eindeutigkeitseigenschaft erfüllt
- Der Begriff des Superschlüssels impliziert keine Aussage über die Minimalität
- In der Mathematik wird allgemein eine Menge  $M$  als **minimale Menge bezüglich einer Eigenschaft  $B$**  bezeichnet, wenn es keine echte Teilmenge von  $M$  gibt, die ebenfalls  $B$  erfüllt.
- Damit können wir auch definieren:  
**Ein Schlüssel ist ein minimaler Superschlüssel**  
(minimale Menge  $S \subseteq R$  mit Eindeutigkeits-Eigenschaft)



# Schlüssel: Beispiele

- Gegeben sei die folgende Relation:

| 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 |
| ...         | ... | ... | ...                            | ...        |

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



# 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 |
|             | ... | ... | ...                            | ...        |

- $\{\text{LNr}\}$  ist **Schlüssel !!!**  
Eindeutigkeit: Alle  $t_i[\text{LNr}]$  sind paarweise verschieden,  
d.h.  $t_1[\text{LNr}] \neq t_2[\text{LNr}], t_1[\text{LNr}] \neq t_3[\text{LNr}], t_2[\text{LNr}] \neq t_3[\text{LNr}]$   
Minimalität: Trivial, weil 1 Attribut kürzeste Möglichkeit
- $\{\text{LNr}, \text{VNr}\}$  ist **kein Schlüssel (aber Superschlüssel)**  
Eindeutigkeit: Alle  $t_i[\text{LNr}, \text{VNr}]$  paarweise verschieden.  
Nicht minimal, da **echte** Teilmenge  $\{\text{LNr}\} \subset \{\text{LNr}, \text{VNr}\}$  ( $\neq$ ) die Eindeutigkeit bereits gewährleistet, s.o.

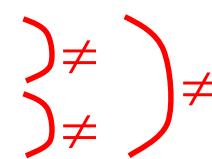


# Schlüssel: Beispiele

| 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:

- $t_1 [VNr, Semester] = (012, \text{WS 2001/02})$
  - $t_2 [VNr, Semester] = (012, \text{WS 2002/03})$
  - $t_3 [VNr, Semester] = (013, \text{WS 2001/02})$
- 

**Minimalität:**

Weder  $\{VNr\}$  noch  $\{\text{Semester}\}$  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 Eindeutigkeit nichts beitragen)
- Manchmal gibt es mehrere verschiedene Schlüssel
  - {LNr}
  - {VNr, Semester} → **Schlüsselkandidat** (SQL: **unique**)
- Später ist wichtig, *alle* Schlüsselkandidaten zu ermitteln.
- Man wählt einen dieser Kandidaten aus als sogenannter  
**Primärschlüssel** (SQL: **primary key**)
- Attribut(e) das auf einen Schlüssel einer anderen Relation  
verweist, heißt **Fremdschlüssel** (SQL: **foreign key**)



# 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
- Wirkung: Definition eines Relationenschemas mit einer leeren Relation als Ausprägung.



# Basis-Typen in SQL

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

- **integer** oder auch **integer4**, **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 (nicht bei zusammengesetzten Schlüsseln)
  - **unique**:  
Das Attribut ist Schlüsselkandidat
  - **references  $t_1(a_1)$** :  
Ein Verweis auf Attribut  $a_1$  von Tabelle  $t_1$
  - **default  $w_1$** : Wert  $w_1$  ist Default, wenn unbesetzt.
  - **check  $f$** :  
Die Formel  $f$  wird bei jeder Einfügung überprüft, z.B.: **check  $A \leq 100$**

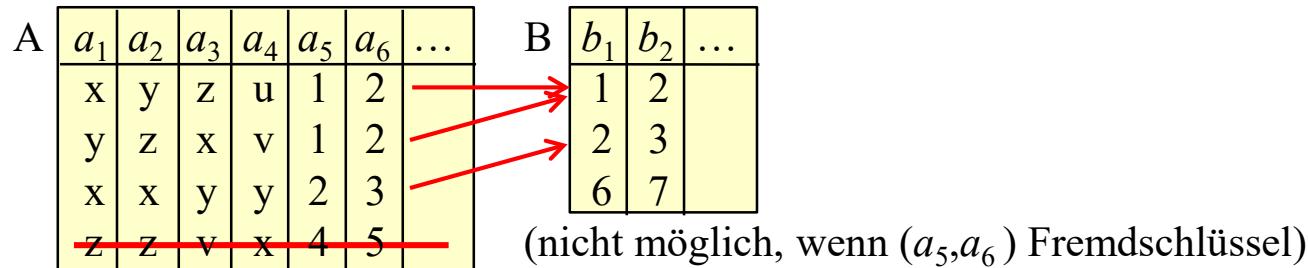


# Integritätsbedingungen

- Zusätze, die keinem einzelnen Attribut zugeordnet sind, stehen mit Komma abgetrennt in extra Zeilen
  - **primary key** ( $A_1, A_2, \dots$ ):  
Zusammengesetzter Primärschlüssel
  - **unique** ( $A_1, A_2, \dots$ ):  
Zusammengesetzter Schlüsselkandidat
  - **foreign key** ( $A_1, A_2, \dots$ ) **references**  $t_1$  ( $B_1, B_2, \dots$ )  
Verweis auf zusammengesetzten Schlüssel in Rel.  $T_1$   
Anmerkung: Fehlt die Angabe ( $B_1, B_2, \dots$ ) hinter  $t_1$  so wird automatisch ( $A_1, A_2, \dots$ ) eingesetzt.
  - **check**  $f$
- Anmerkung: SQL ist case-insensitiv:  
Im Ggs. zu Java hat die Groß-/Kleinschreibung weder bei Schlüsselworten noch bei Bezeichnern Bedeutung



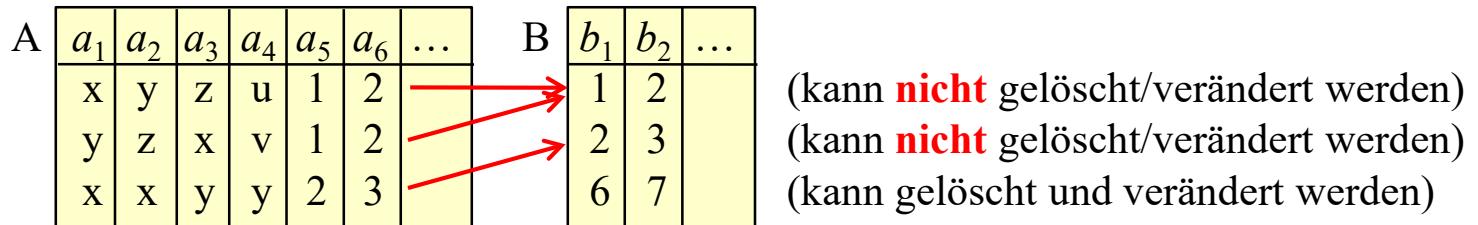
# Schlüssel-Definitionen



- **primary key ( $a_1, a_2$ )**,  
definiert den Primärschlüssel.
- **unique ( $a_3, a_4$ )**,  
definiert einen weiteren Schlüsselkandidaten.
- **foreign key ( $a_5, a_6$ ) references B ( $b_1, b_2$ )**  
definiert einen Fremdschlüssel.
  - Tupel in A ohne gültigen Partner in B nicht erlaubt
  - Ohne weiteren Zusatz nicht möglich, Tupel in B, auf die durch in Tupel in A verwiesen wird, zu löschen oder die Werte von  $b_1, b_2$  zu verändern.



# Schlüssel-Definitionen



Löschen eines Tupels in B mit Referenzen nicht möglich.

Es gibt aber verschiedene Zusätze:

- **foreign key ( $a_5,a_6$ ) references B ( $b_1,b_2$ )  
on delete cascade**

Löschen eines Tupels in B führt auch zum Löschen der entsprechenden Tupel in A

- **on update cascade**

Verändern eines Tupels in B führt zum Verändern in A

- **on delete set null**

„hängende Verweise“ werden ggf. auf **null** gesetzt.



# 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 ( $a_1 d_1 c_1, a_2 d_2 c_2, \dots$ )**
    - Zusätzliche Attribute oder Integritätsbedingungen werden (rechts) an die Tabelle angehängt
    - Bei allen vorhandenen Tupeln Null-Werte
  - **alter table  $n_1$  drop ( $a_1, a_2, \dots$ )**
  - **alter table  $n_1$  modify ( $a_1 d_1 c_1, a_2 d_2 c_2, \dots$ )**



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 3: Die Relationale Algebra

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani  
Skript © 2019 Christian Böhm



# 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
    - Boolesche 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 (z.B. *Gehalt > 1000*)
    - 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:  $R = \pi_{A,B,\dots}(S)$
- 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 (Name *und* Domäne)
- Die Ergebnis-Relation  $R$  bekommt dieses Schema
- Vereinigung: 
$$S \cup T = \{t \mid t \in S \vee t \in T\}$$
- Differenz: 
$$S - T = \{t \mid t \in S \wedge t \notin T\}$$
- Was wissen wir über die *Kardinalität* des Ergebnisses (Anzahl der Tupel)?
$$|S \cup T| \leq |S| + |T|$$
$$|S - T| \geq |S| - |T|$$
$$|S - T| \leq |S|$$
- **Achtung:** Zwei Tupel gelten nur dann als *gleich*, wenn *alle* ihre Attributwerte gleich sind.



# Beispiel

**Mitarbeiter:**

| Name    | Vorname  |
|---------|----------|
| Huber   | Egon     |
| Maier   | Wolfgang |
| Schmidt | Helmut   |

**Studierende:**

| Name    | Vorname |
|---------|---------|
| Müller  | Heinz   |
| Schmidt | Helmut  |

Alle Personen, die Mitarbeiter oder Studierende sind:

**Mitarbeiter  $\cup$  Studierende:**

| Name               | Vorname           |
|--------------------|-------------------|
| Huber              | Egon              |
| Maier              | Wolfgang          |
| Schmidt            | Helmut            |
| Müller             | Heinz             |
| <del>Schmidt</del> | <del>Helmut</del> |

Duplikat-  
Elimination!

Alle Mitarbeiter ohne diejenigen, die auch Studierende sind:

**Mitarbeiter – Studierende:**

| 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  $a_1, a_2, \dots, a_s$  die Attribute von  $S$  und  $b_1, b_2, \dots, b_t$  die Attribute von  $T$
- Dann ist  $R = S \times T$  die folgende Menge (Relation):  
 $\{(a_1, \dots, a_s, b_1, \dots, b_t) \mid (a_1, \dots, a_s) \in S \wedge (b_1, \dots, b_t) \in T\}$
- 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  $\times$  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/gewünscht/intuitiv?



# 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:

$$\sigma_F(S) = \{t \mid t \in S \wedge F(t)\}$$

- $R$  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:

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

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

Kann jetzt die Frage von S. 9 beantwortet werden?



# Beispiel

Mitarbeiter  $\times$  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     |

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

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

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

**Grund:** Nach dem Streichen von Spalten können Duplikat-Tupel entstanden sein, die eliminiert werden, damit wieder eine Relation (Menge von Tupeln) entsteht.



# Projektion: Beispiel

Mitarbeiter

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

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

Zwischenergebnis (Multimenge):

| Name   | Abteilung |
|--------|-----------|
| Huber  | 01        |
| Mayer  | 01        |
| Müller | 02        |
| Mayer  | 01        |

**Duplikate**   
**Elimination**

Endergebnis (Menge):

| Name   | Abteilung |
|--------|-----------|
| Huber  | 01        |
| Mayer  | 01        |
| Müller | 02        |



# Duplikat-Elimination

- Erforderlich nach...
  - Projektion
  - Vereinigung
- Wie funktioniert Duplikat-Elimination?

```
for (int i = 0 ; i < R.length ; i++)  
    for (int j = 0 ; j < i ; j++)  
        if (R[i] == R[j])  
            // R[j] markieren um es später zu löschen
```
- Aufwand?  $n=R.length$ :  $O(n^2)$
- Besserer Algorithmus mit Sortieren:  $O(n \log n)$ 

⇒ An sich billige Grund-Operationen werden nur durch die Duplikat-Elimination 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 ( $\geq 500.000$ ) und ihre Einwohner

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

- In welchem Land liegt die Stadt Passau?

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

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

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



# 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

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

oder auch:

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

- Welche Länder werden von der SPD *allein* regiert

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



# Beispiel Bundesländer

| Länder: | LName             | LEinw      | Partei |
|---------|-------------------|------------|--------|
|         | Baden-Württemberg | 10.745.000 | Grüne  |
|         | Baden-Württemberg | 10.745.000 | SPD    |
|         | Bayern            | 12.510.000 | CSU    |
|         | Bayern            | 12.510.000 | FDP    |
|         | Berlin            | 3.443.000  | SPD    |
|         | Berlin            | 3.443.000  | Linke  |
|         | Brandenburg       | 2.512.000  | SPD    |
|         | Brandenburg       | 2.512.000  | Linke  |
|         | Bremen            | 662.000    | SPD    |
|         | Bremen            | 662.000    | Grüne  |
|         | Hamburg           | 1.774.000  | SPD    |
|         | ...               | ...        | ...    |



# Beispiel Bundesländer

$S = \sigma_{\text{Partei} = \text{SPD}}(\text{Länder})$ :

| LName       | LEinw      | Partei |
|-------------|------------|--------|
| Baden-W.    | 10.745.000 | SPD    |
| Berlin      | 3.443.000  | SPD    |
| Brandenburg | 2.512.000  | SPD    |
| Bremen      | 662.000    | SPD    |
| Hamburg     | 1.774.000  | SPD    |
| ...         | ...        | ...    |

$T = \sigma_{\text{Partei} \neq \text{SPD}}(\text{Länder})$ :

| LName       | LEinw      | Partei |
|-------------|------------|--------|
| Baden-W.    | 10.745.000 | Grüne  |
| Bayern      | 12.510.000 | CSU    |
| Bayern      | 12.510.000 | FDP    |
| Berlin      | 3.443.000  | Linke  |
| Brandenburg | 2.512.000  | Linke  |
| Bremen      | 662.000    | Grüne  |
| ...         | ...        | ...    |

- $S - T$  würde nicht das gewünschte Ergebnis liefern, da bei der Mengendifferenz nur exakt gleiche Tupel berücksichtigt werden
- Attribut *Partei* stört; Ergebnis wäre daher wieder Zwischenergebnis  $S$
- $\pi_{\text{LName}}(S) - \pi_{\text{LName}}(T)$ :  
(gewünschtes Ergebnis)

| LName   |
|---------|
| Hamburg |



# 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):

$$S \cap T = \{t \mid t \in S \wedge t \in T\}$$

- Beispiel:  
Welche Personen sind gleichzeitig Mitarbeiter und studieren?

Mitarbeiter:

| Name    | Vorname  |
|---------|----------|
| Huber   | Egon     |
| Maier   | Wolfgang |
| Schmidt | Helmut   |

Studierende:

| Name    | Vorname |
|---------|---------|
| Müller  | Heinz   |
| Schmidt | Helmut  |

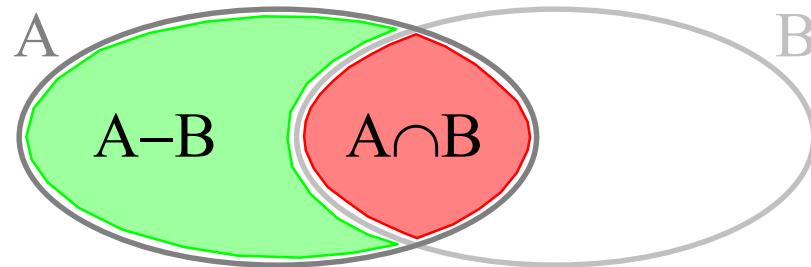
Mitarbeiter  $\cap$  Studierende:

| 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:

| $R_1$ | <b>Programmierer</b> | <b>Sprache</b> | $R_2$ | <b>Sprache</b> |
|-------|----------------------|----------------|-------|----------------|
|       | Müller               | Java           |       | Basic          |
|       | Müller               | Basic          |       | C++            |
|       | Müller               | C++            |       | Java           |
|       | Huber                | C++            |       |                |
|       | Huber                | Java           |       |                |

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

| $R_1 \div R_2$ | <b>Programmierer</b> |
|----------------|----------------------|
|                | Müller               |

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



# Join

- Wie vorher erwähnt:  
Selektion über Kreuzprodukt zweier Relationen
  - Theta-Join ( $\Theta$ ):  $R \underset{A \Theta B}{\bowtie} S$   
Allgemeiner Vergleich:  
 $A$  ist ein Attribut von  $R$  und  $B$  ein Attribut von  $S$   
 $\Theta$  ist einer der Operatoren:  
 $=, <, \leq, >, \geq, \neq$
  - Equi-Join:  $R \underset{A = B}{\bowtie} S$
  - Natural Join:  $R \bowtie S$ :
    1. Ein Equi-Join bezüglich aller gleich-nameden Attribute in  $R$  und  $S$ .
    2. Gleiche Spalten werden gestrichen (Projektion)



# Join

- Implementierung mit Hilfe der Grundoperationen

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

---

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

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

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

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



# SQL

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

Projektion → **SELECT**    ⟨Liste von Attributnamen 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 ...
    - SELECT A<sub>1</sub>, A<sub>2</sub>, ... FROM ...
    - SELECT DISTINCT A<sub>1</sub>, A<sub>2</sub>, ...
  - Bei der zweiten Form kann die Ergebnis-*relation*“ also u.U. Duplike enthalten
  - Grund: Performanz
- Keine Projektion
  - Projektion ohne Duplikatelimination
  - Projektion mit Duplikatelimination



# SELECT

- Bei den Attributen  $A_1, A_2, \dots$  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  $A_1 \text{ AS } A_2$ :  
 $A_2$  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
    - $\text{FROM } R_1, R_2, \dots$
- so wird das kartesische Produkt gebildet:
- $R_1 \times R_2 \times \dots$
  - 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 | <b>Mitarbeiter</b> .Name, <b>Abteilung</b> .Name, ... |
| FROM   | Mitarbeiter, Abteilung                                |
| WHERE  | ...   |



# FROM

- Man kann Schreibarbeit sparen, indem man den Relationen lokal (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 ein logisches Prädikat  $\Phi$  (Funktion die einen booleschen Wert (wahr/falsch) zurück gibt).
- Das logische Prädikat besteht aus
  - Vergleichen zwischen Attributwerten und Konstanten
  - Vergleichen zwischen verschiedenen Attributen
  - Vergleichsoperatoren\*:  $=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $\neq$
  - Test auf *Wert undefiniert*:  $A_1 \text{ IS NULL/IS NOT NULL}$
  - Inexakter Stringvergleich:  $A_1 \text{ LIKE } 'Datenbank\%'$
  - $A_1 \text{ IN } (2, 3, 5, 7, 11, 13)$

---

\*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 der Anfrageauswertung:  
Alle Tupel des kartesischen Produktes aus der FROM-Klausel werden getestet, ob sie  $\Phi$  erfüllen
- Effizientere Ausführung ist meist möglich
- Das DBMS optimiert die Anfragen automatisch sehr gut



# WHERE

- Inexakte Stringsuche:  $A_1 \text{ LIKE } 'Datenbank\%'$ 
  - bedeutet: Alle Datensätze, bei denen Attribut  $A_1$  mit dem Präfix *Datenbank* beginnt.
  - Entsprechend:  $A_1 \text{ LIKE } '\%\text{Daten}\%'$
  - In dem Spezialstring hinter LIKE ...
    - $\%$  steht für einen beliebig belegbaren Teilstring
    - $_$  steht für genau 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.KAdr  
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)

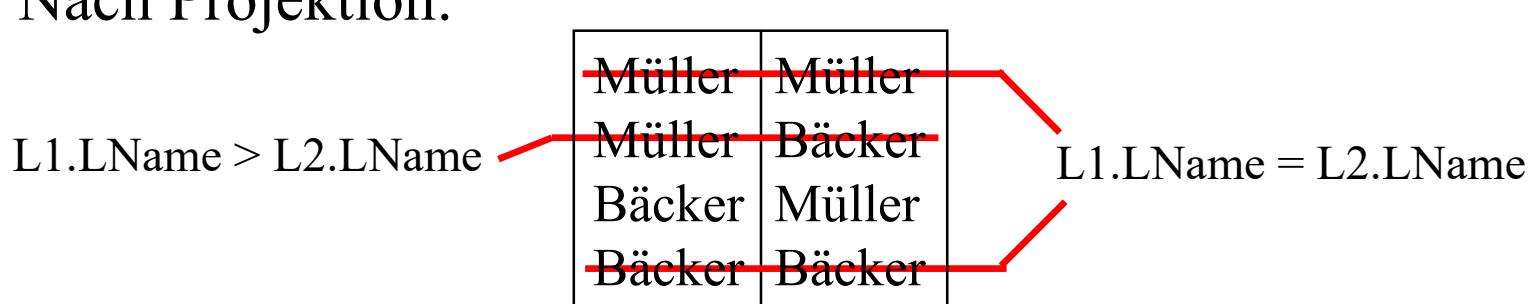
Lieferant\*

|        |         |
|--------|---------|
| Müller | Mehl    |
| Müller | Haferfl |
| Bäcker | Mehl    |

**select \* from Lieferant L1, Lieferant L2 where L1.Ware=L2.Ware:**

|        |         |        |         |
|--------|---------|--------|---------|
| Müller | Mehl    | Müller | Mehl    |
| Müller | Mehl    | Bäcker | Mehl    |
| Müller | Haferfl | Müller | Haferfl |
| Bäcker | Mehl    | Müller | Mehl    |
| Bäcker | Mehl    | Bäcker | Mehl    |

Nach Projektion:





# Outer Join

- Problem:

Beim gewöhnlichen („inner“) Join gehen diejenigen Tupel verloren, die keine Join-Partner 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    | M    | 10  |
| Huber    | M    | 25  |
| Geizhals | RO   | 0   |

**Auftrag:**

| KName | Ware  | ... |
|-------|-------|-----|
| Maier | Brot  |     |
| Maier | Milch |     |
| Huber | Mehl  |     |

**Kunde  $\bowtie$  Auftrag:**

| KName | KAdr | Kto | Ware  | ... |
|-------|------|-----|-------|-----|
| Maier | M    | 10  | Brot  |     |
| Maier | M    | 10  | Milch |     |
| Huber | M    | 25  | Mehl  |     |

Geizhals erscheint nicht mehr in der erweiterten Liste.



# Outer Join

- Ein Outer Join ergänzt das Join-Ergebnis 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    | M    | 10  |
| Huber    | M    | 25  |
| Geizhals | RO   | 0   |

Auftrag:

| KName | Ware  | ... |
|-------|-------|-----|
| Maier | Brot  |     |
| Maier | Milch |     |
| Huber | Mehl  |     |

Kunde nat. outer join Auftrag:

| KName    | KAdr | Kto | Ware  | ... |
|----------|------|-----|-------|-----|
| Maier    | M    | 10  | Brot  |     |
| Maier    | M    | 10  | Milch |     |
| Huber    | M    | 25  | Mehl  |     |
| Geizhals | RO   | 0   | NULL  |     |



# Outer Join

- Aufstellung aller Möglichkeiten:
  - **[inner] join**: keine verlustfreie Relation
  - **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**, **using**:  
**select \* from L [left/right/full [outer]] natural join R**

| 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 |



# UNION, INTERSECT, EXCEPT

- Üblicherweise werden mit diesen Operationen die Ergebnisse zweier **select-from-where**-Blöcke verknüpft:

```
select * from Mitarbeiter where name like 'A%'  
union  
select * from Studenten where name like 'A%'
```
- Bei neueren Datenbanksystemen ist auch möglich:

```
select * from Mitarbeiter union Studenten where ...
```
- Es gibt folgende Mengen-Operationen:
  - **union**: Vereinigung mit Duplikatelimination
  - **union all**: Vereinigung ohne Duplikatelimination
  - **intersect**: Schnittmenge
  - **except, minus**: Mengen-Differenz
- Achtung: Zwei Tupel sind nur dann gleich, wenn alle ihre Attributwerte gleich sind



# 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 Wertebereiche sind **character** (Länge usw. egal)
  - beide Wertebereiche sind Zahlen (Genauigkeit egal)
  - oder beide Wertebereiche sind gleich
- Die Namen der Attribute müssen nicht gleich sein
- Befinden sich Attribute gleichen Namens auf unterschiedlichen Positionen, sind trotzdem nur die Positionen maßgeblich (das ist oft irritierend)



# UNION, INTERSECT, EXCEPT

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

*R:*

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 3 | 4 |

`select * from R`

**union**

`select * from S:`

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

*S:*

| A | C | D |
|---|---|---|
| 2 | 2 | 3 |
| 5 | 3 | 2 |

`select * from R`

**union corresponding**

`select * from S:`

| A | C |
|---|---|
| 1 | 3 |
| 2 | 4 |
| 2 | 2 |
| 5 | 3 |



# UNION, INTERSECT, EXCEPT

- Bei **corresponding** wird *vor* der Vereinigung automatisch eine Projektion bezüglich aller **gleich benannten** Attribute durchgeführt
- Dies kann man besser durch explizites Aufzählen der Attribute in den **select**-Klauseln erreichen.
- So ist auch möglich:

select A,B,C from R  
**union**  
select A,D,C from S:

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

Oder auch:

select A,B,C,**null as D** from R  
**union**  
select A,**null**,C,D from S:

| A | B           | C | D           |
|---|-------------|---|-------------|
| 1 | 2           | 3 | <b>null</b> |
| 2 | 3           | 4 | <b>null</b> |
| 2 | <b>null</b> | 2 | 3           |
| 5 | <b>null</b> | 3 | 2           |



# UNION, INTERSECT, EXCEPT

- Bei **corresponding** ist ggf. die Reihenfolge der Attribute der erstgenannten Teilanfrage maßgeblich:

| T: | X | Y |
|----|---|---|
|    | 1 | 2 |
|    |   |   |

| U: | Y | X |
|----|---|---|
|    | 3 | 4 |
|    |   |   |

select \* from T  
**union**  
select \* from U:

| X | Y |
|---|---|
| 1 | 2 |
| 3 | 4 |

| X | Y |
|---|---|
| 1 | 2 |
| 4 | 3 |

select \* from T  
**union corresponding**  
select \* from U:

- Die Attribute der zweiten Teil-Anfrage ändern evtl...
  - ohne **corresponding**: sie ändern ihren Namen
  - mit **corresponding**: sie ändern ihre Reihenfolge



# 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<sub>1</sub> = ausdruck<sub>1</sub>*

[ , ... ,

*attribut<sub>n</sub> = ausdruck<sub>n</sub> ]*

**[where**

*bedingung]*

- Wirkung:

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



# 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-Bedingung**
- 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<sub>1</sub>, attribut<sub>2</sub>,...)***  
***values (konstante<sub>1</sub>, konstante<sub>2</sub>, ...)***
- oder:  
***insert into relation***  
***values (konstante<sub>1</sub>, konstante<sub>2</sub>, ...)***



# 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 Attributliste zugeordnet
- Beispiel:  
**insert into Angestellte (Vorname, 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*<sub>1</sub> , ...)]  
( 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)
- Dies ist eigentlich eine Form einer Unteranfrage (Subquery: siehe auch Kapitel 4)



# 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)**



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 4: Relationen-Kalkül

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani  
Skript © 2019 Christian Böhm



# Begriff

**Kalkül** 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

**der Kalkül ...**

**...das Kalkül**

**Kalkül** der; -s, -e <zur<sup>1</sup> 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  $\{x \mid x \in \mathbb{N} \wedge x^3 > 0 \wedge x^3 < 1000\}$
- Anwendung solcher Formeln für DB-Anfragen
  - Bezugnahme auf DB-Relationen im Bedingungsteil:  
 $(x_1, y_1, z_1) \in \text{Mitarbeiter}, t_1 \in \text{Abteilungen}$
  - 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:  
 $F_1 \wedge F_2, F_1 \vee F_2, \neg F_1, \exists x: F_1, \forall x: F_1$
- 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  $t \in \text{Städte}$



# Unterschied zur Rel. Algebra

- Relationale Algebra ist **prozedurale** Sprache --- WIE
  - Ausdruck gibt an, unter Benutzung welcher Operationen das Ergebnis berechnet werden soll
- Relationen-Kalkül ist **deklarative** Sprache --- WAS
  - Ausdruck beschreibt, welche Eigenschaften die Tupel der Ergebnisrelation haben müssen ohne eine Berechnungsprozedur dafür anzugeben
- 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  $\psi$  **erfüllen** (wahr machen)
- Ein Kalkül besteht immer aus
  - Syntax: Wie sind Ausdrücke aufgebaut?
  - Semantik: Was bedeuten die Ausdrücke?



# Tupelvariablen

- Tupelvariablen haben ein definiertes Schema:
  - $\text{Schema}(t) = (A_1: D_1, A_2: D_2, \dots)$
  - $\text{Schema}(t) = R_1$  ( $t$  hat dasselbe Schema wie Relation)
- Für Zugriff auf die Komponenten
  - $t[A]$  oder  $t.A$  für einen Attributnamen  $A \in \text{Schema}(t)$
  - oder auch  $t[1], t[2]$  usw.
- Tupelvariable kann in einer Formel  $\psi$  **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  $\Theta$  zu ...*
  - $t.A \Theta c$   $t$  ist Tupelvariable und  $c$  eine passende Konstante

---

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



# Formeln

Der Aufbau von Formeln  $\psi$  ist rekursiv definiert:

- **Atome:** Jedes Atom ist eine Formel  
Alle vorkommenden Variablen sind **frei**
- **Verknüpfungen:** Sind  $\psi_1$  und  $\psi_2$  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  $\psi$  eine Formel, in der  $t$  als **freie Variable** auftritt, sind auch Formeln...
  - $(\exists t)(\psi)$  **es gibt ein  $t$ , für das  $\psi$**
  - $(\forall t)(\psi)$  **für alle  $t$  gilt  $\psi$**die Variable  $t$  wird **gebunden**.



# Formeln

- Gebräuchliche vereinfachende Schreibweisen:
  - $\psi_1 \Rightarrow \psi_2$  für  $(\neg \psi_1) \vee \psi_2$  (**Implikation**)
  - $\exists t_1, \dots, t_k : \psi(t_1, \dots, t_k)$  für  $(\exists t_1) (\dots ((\exists t_k) (\psi(t_1, \dots, t_k)))) \dots$
  - $(\exists t \in R) (\psi(t))$  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:
  - $(\forall s) (s.A \leq u.B \vee (\exists u)(R(u) \wedge u.C > t.D))$
  - $t$  ist **frei**
  - $s$  ist **gebunden**
  - $u$  ist **beim ersten Auftreten frei, dann gebunden**



# Wiederholung: Implikation

$$\Psi_1 \Rightarrow \Psi_2$$

|               | $\neg \Psi_1$ | $\Psi_1$ |
|---------------|---------------|----------|
| $\neg \Psi_2$ | wahr          | falsch   |
| $\Psi_2$      | wahr          | wahr     |

$\left. \begin{matrix} \\ = \Psi_2 \end{matrix} \right\}$

$$(\neg \Psi_1) \vee \Psi_2$$

|               | $\neg \Psi_1$ | $\Psi_1$ |
|---------------|---------------|----------|
| $\neg \Psi_2$ | wahr          | falsch   |
| $\Psi_2$      | wahr          | wahr     |

↔



# Wiederholung: Implikation

$\Psi_1 \Rightarrow \Psi_2$ :

|               | $\neg \Psi_1$ | $\Psi_1$ |
|---------------|---------------|----------|
| $\neg \Psi_2$ | wahr          | falsch   |
| $\Psi_2$      | wahr          | wahr     |

$\Psi_1 \vee \Psi_2$ :

|               | $\neg \Psi_1$ | $\Psi_1$ |
|---------------|---------------|----------|
| $\neg \Psi_2$ | falsch        | wahr     |
| $\Psi_2$      | wahr          | wahr     |



$(\neg \Psi_1) \vee \Psi_2$ :

|               | $\neg \Psi_1$ | $\Psi_1$ |
|---------------|---------------|----------|
| $\neg \Psi_2$ | wahr          | falsch   |
| $\Psi_2$      | wahr          | wahr     |

- Wenn  $\Psi_1 = \text{falsch}$  (linke Spalte), dann folgt nichts für  $\Psi_2$ , d.h. Gesamtaussage  $\Psi_1 \Rightarrow \Psi_2$  ist dann auf jeden Fall *wahr*.
- Die Gesamtaussage ist nur *falsch*, wenn  $\Psi_1 = \text{wahr}$  und  $\Psi_2 = \text{falsch}$ .

- Die ODER-Verknüpfung hat ähnliche Logiktabelle: 3 x *wahr*, 1 x *falsch*.
- Aber die Spalten sind vertauscht.

- Die Negation von  $\Psi_1$  erzeugt die „richtige“ Logiktabelle, identisch mit  $\Psi_1 \Rightarrow \Psi_2$



# Wiederholung: Quantoren

$$(\exists t \in R) (\psi(t))$$

für  $(\exists t) (R(t) \wedge \psi(t))$

$$(\forall t \in R) (\psi(t))$$

für  $(\forall t) (R(t) \Rightarrow \psi(t))$

Sei  $X = \{x_1, x_2, x_3, \dots\}$ .

- Allquantor ist implizite UND-Verknüpfung:  
 $(\forall x)(\psi(x)) \Leftrightarrow \psi(x_1) \wedge \psi(x_2) \wedge \psi(x_3) \wedge \dots$
- Existenzquantor ist implizite ODER-Verknüpfung:  
 $(\exists x)(\psi(x)) \Leftrightarrow \psi(x_1) \vee \psi(x_2) \vee \psi(x_3) \vee \dots$
- Wir wollen uns nicht auf alle Elemente der Grundmenge sondern nur auf die gespeicherten Elemente beziehen:
- Wende auf nicht gespeicherte Tupel statt  $\psi(t)$  **neutrales** Element der Verknüpfung an: ODER: *false*; UND: *true*.



# Wiederholung: Quantoren

$(\exists t \in R) (\psi(t))$

für  $(\exists t) (R(t) \wedge \psi(t))$

$(\forall t \in R) (\psi(t))$

für  $(\forall t) (R(t) \Rightarrow \psi(t))$

- Wie mache ich die nicht gespeicherten Tupel zum neutralen Element?
  - ODER ( $\exists$ -Quantor):  $(R(t) \wedge \psi(t))$   
die  $\wedge$  -Verknüpfung mit  $R(t)$  erzeugt *false*,
  - UND ( $\forall$ -Quantor):  $(R(t) \Rightarrow \psi(t))$   
die  $\vee$  -Verknüpfung mit  $\neg R(t)$  erzeugt *true*,  
unabhängig von  $\psi(t)$  bei nicht gespeicherten Tupeln.  
Bei  $R(t) = \text{true}$  wird  $\psi(t)$  als Gesamtergebnis übernommen  
(siehe Logiktabellen nächste Seite).



# Wiederholung: Quantoren

$R(t) \Rightarrow \psi(t)$

|             | $\neg R(t)$ | $R(t)$ |
|-------------|-------------|--------|
| $\neg \psi$ | wahr        | falsch |
| $\psi$      | wahr        | wahr   |

$\underbrace{\neg R(t)}_{\text{neutr.}} \quad \underbrace{R(t)}_{= \psi(t)}$   
 $für \forall$

$R(t) \wedge \psi(t)$

|             | $\neg R(t)$ | $R(t)$ |
|-------------|-------------|--------|
| $\neg \psi$ | falsch      | falsch |
| $\psi$      | falsch      | wahr   |

$\underbrace{\neg R(t)}_{\text{neutr.}} \quad \underbrace{R(t)}_{= \psi(t)}$   
 $für \exists$



# Ausdruck (Anfrage)

- Ein Ausdruck des Tupelkalküls hat die Form

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

- In Formel  $\psi$  ist  $t$  die einzige freie Variable



# Semantik

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

Syntax → Semantik

Tupelvariablen → konkrete Tupel  
Formeln → true, false  
Ausdrücke → Relationen



# Belegung von Variablen

- Gegeben:
  - eine Tupelvariable  $t$  mit  $\text{Schema}(t) = (D_1, D_2, \dots)$
  - 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)
  - $r_1 = (\text{Passau}, 49.800, \text{Bayern})$ :  
 $\psi(r_1 | t) = (\text{Bayern} = \text{Bayern} \wedge 49.800 \geq 500.000)$
  - $r_2 = (\text{Bremen}, 535.058, \text{Bremen})$ :  
 $\psi(r_2 | t) = (\text{Bremen} = \text{Bayern} \wedge 535.058 \geq 500.000)$



# Interpretation von Formeln

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

- Anm: Alle Variablen sind durch konkrete Tupel belegt
- Atome:
  - $R(r)$ :  $I(R(r)) = \mathbf{true} \Leftrightarrow r$  ist in  $R$  enthalten
  - $c_i \Theta c_j$ :  $I(c_i \Theta c_j) = \mathbf{true} \Leftrightarrow$  der Vergleich ist erfüllt
- Logische Operatoren:
  - $\neg\psi$ :  $I(\neg\psi) = \mathbf{true} \Leftrightarrow I(\psi) = \mathbf{false}$
  - $\psi_1 \wedge \psi_2$ :  $I(\psi_1 \wedge \psi_2) = \mathbf{true} \Leftrightarrow I(\psi_1) = \mathbf{true}$  und  $I(\psi_2) = \mathbf{true}$
  - $\psi_1 \vee \psi_2$ :  $I(\psi_1 \vee \psi_2) = \mathbf{true} \Leftrightarrow I(\psi_1) = \mathbf{true}$  oder  $I(\psi_2) = \mathbf{true}$



# Beispiele

- Atome:
  - $I(\text{Städte}(\text{Passau}, 49.800, \text{Bayern}))$  = **true**
  - $I(49.800 \geq 500.000)$  = **false**
- Logische Operatoren:
  - $I(\neg 49.800 \geq 500.000)$  = **true**
  - $I(\text{Städte}(\text{Passau}, 49.800, \text{Bayern}) \vee 49.800 \geq 500.000)$  = **true**
  - $I(\text{Städte}(\text{Passau}, 49.800, \text{Bayern}) \wedge 49.800 \geq 500.000)$  = **false**



# Interpretation von Quantoren

- Interpretation  $I((\exists s)(\psi))$  bzw.  $I((\forall s)(\psi))$ :
  - In  $\psi$  darf **nur**  $s$  als freie Variable auftreten.
  - $I((\exists s)(\psi)) = \text{true} \Leftrightarrow$  ein Tupel  $r \in D_1 \times D_2 \times \dots$  existiert, daß bei Belegung der Variablen  $s$  die Formel  $\psi$  gilt:
$$I(\psi(r | s)) = \text{true}$$
  - $I((\forall s)(\psi)) = \text{true} \Leftrightarrow$  für alle Tupel  $r \in D_1 \times D_2 \times \dots$  gilt die Formel  $\psi$ .
- Beispiele:
  - $I((\exists s)(\text{Städte}(s) \wedge s.\text{Land} = \text{Bayern})) = \text{true}$
  - $I((\forall s)(s.\text{Name} = \text{Passau})) = \text{false}$



# 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( $t$ ) =  $D_1 \times D_2 \times \dots$
- Dann ist der Wert von  $E$  die Menge aller\* (denkbaren) Tupel  $r \in D_1 \times D_2 \times \dots$  für die gilt:

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

---

\*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:  
 $\text{Schema}(t) = \text{Schema}(\text{Städte})$   
 $\{t \mid \text{Städte}(t) \wedge t.\text{Land} = \text{Bayern} \wedge t.\text{SEinw} \geq 500.000\}$
- In welchem Land liegt Passau?  
 $\text{Schema}(t) = (\text{Land:String})$   
 $\{t \mid (\exists u \in \text{Städte})(u.\text{Sname} = \text{Passau} \wedge u.\text{Land} = t.\text{Land})\}$
- Finde alle Städte in CDU-regierten Ländern:  
 $\text{Schema}(t) = \text{Schema}(\text{Städte})$   
 $\{t \mid \text{Städte}(t) \wedge (\exists u \in \text{Länder})(u.\text{Lname} = t.\text{Land} \wedge u.\text{Partei} = \text{CDU})\}$



# 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?  
 $\text{Schema}(t) = \text{Schema}(\text{Länder})$   
 $\{t | \text{Länder}(t) \wedge (\forall u \in \text{Länder})(u.\text{LName} = t.\text{LName} \Rightarrow u.\text{Partei} = \text{SPD})\}$
- Gleichbedeutend mit:  
 $\text{Schema}(t) = \text{Schema}(\text{Länder})$   
 $\{t | \text{Länder}(t) \wedge (\forall u \in \text{Länder}) \neg(u.\text{LName} = t.\text{LName} \wedge u.\text{Partei} \neq \text{SPD})\}$



# Beispiel Bundesländer

| Länder: | LName             | LEinw      | Partei |
|---------|-------------------|------------|--------|
|         | Baden-Württemberg | 10.745.000 | Grüne  |
|         | Baden-Württemberg | 10.745.000 | SPD    |
|         | Bayern            | 12.510.000 | CSU    |
|         | Bayern            | 12.510.000 | FDP    |
|         | Berlin            | 3.443.000  | SPD    |
|         | Berlin            | 3.443.000  | Linke  |
|         | Brandenburg       | 2.512.000  | SPD    |
|         | Brandenburg       | 2.512.000  | Linke  |
|         | Bremen            | 662.000    | SPD    |
|         | Bremen            | 662.000    | Grüne  |
|         | Hamburg           | 1.774.000  | SPD    |
|         | ...               | ...        | ...    |



# Sichere Ausdrücke

- Mit den bisherigen Definitionen ist es möglich, unendliche Relationen zu beschreiben:
  - $\text{Schema}(t) = \{\text{String}, \text{String}\}$
  - $\{t \mid t.1 = t.2\}$
  - Ergebnis:  $\{(A,A), (B,B), \dots, (AA,AA), (AB,AB), \dots\}$
- Probleme:
  - Ergebnis kann nicht gespeichert werden
  - Ergebnis kann nicht in endlicher Zeit berechnet werden
- Definition:

Ein Ausdruck heißt **sicher**, wenn jede Tupelvariable sich nur auf Werte bezieht, die einer gespeicherten Relation annehmen kann, also positiv in einem Atom **R(t)** vorkommt.



# Der Bereichskalkül

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

Ein **Ausdruck** hat die Form:

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

**Atome** haben die Form:

- $R_1(x_1, x_2, \dots)$ : Tupel  $(x_1, x_2, \dots)$  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

$\{l \mid \exists e: (\text{Städte}(\text{Passau}, e, l))\}$

d.h. Konstanten dürfen direkt in Relationen eingetragen werden.

- Finde alle Städte in CDU-regierten Ländern:

$\{x_1 \mid \exists x_2, x_3, y_2: (\text{Städte}(x_1, x_2, x_3) \wedge \text{Länder}(x_3, y_2, \text{CDU}))\}$

d.h. „Equijoin“ direkt über Verwendung derselben Bereichsvariable.

- Welche Länder werden von der SPD allein regiert?

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

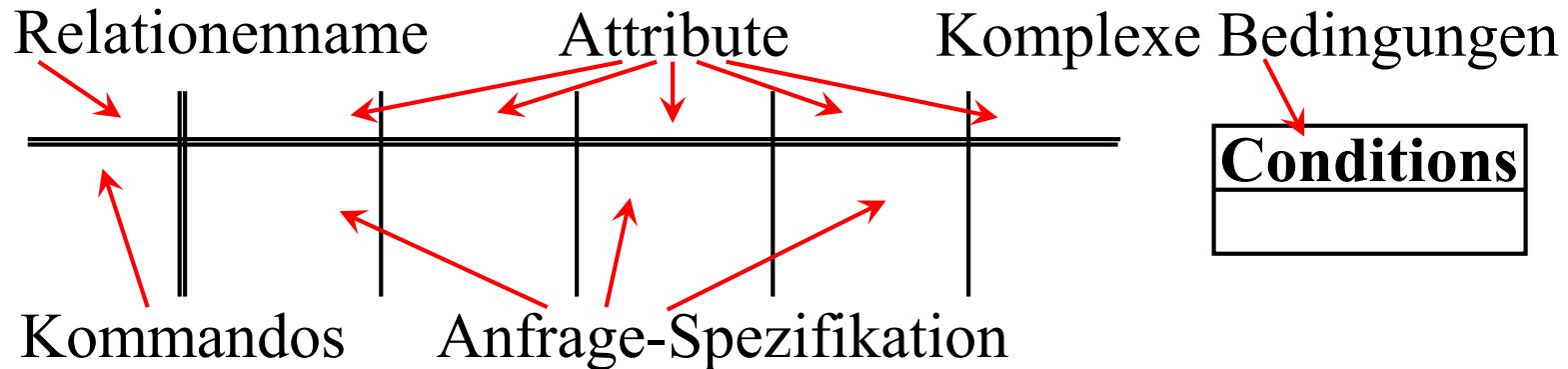


# Query By Example (QBE)

- 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 Attributnamen der Relation ein  
Kunde   || KName   || KAdr   || Kto
- Benutzer stellt Anfrage  
Kunde   || KName   || KAdr   || Kto  
            || P.       || P.       || <0
- System füllt Tabelle mit Ergebnis-Werten

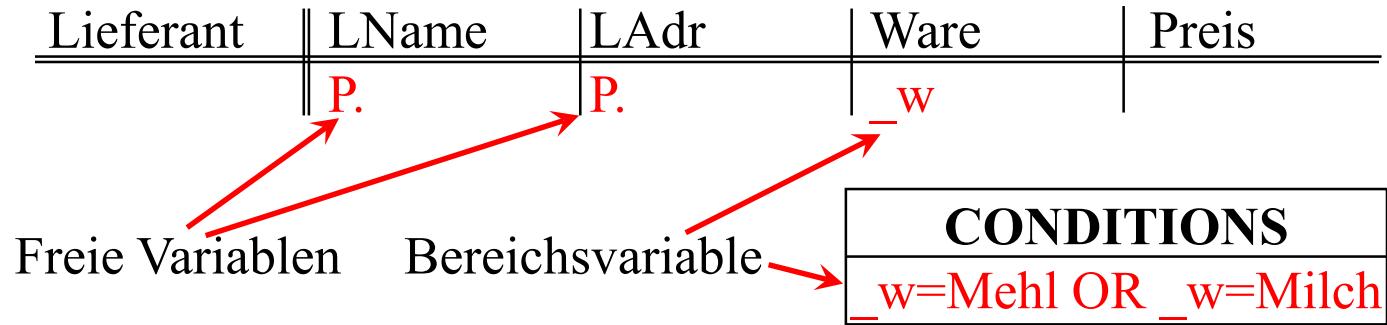
| Kunde | KName | KAdr      |
|-------|-------|-----------|
|       | Huber | Innsbruck |
|       | Maier | München   |

evtl. weitere Tabelle (Join)



# Anfragen mit Bedingungen

- Welche Lieferanten liefern Mehl oder Milch?



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

|                     |
|---------------------|
| <b>CONDITIONS</b>   |
| $_y + _z \leq 7.00$ |

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



# Join-Anfragen

- Welcher Lieferant liefert etwas das Huber bestellt hat?

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

| Auftrag | KName | Ware | Menge |
|---------|-------|------|-------|
| Huber   |       | _w   |       |

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

Abkürzung!

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



# Anfragen mit Ungleichung

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

| Lieferant | LName    | LAdr | Ware  | Preis      |
|-----------|----------|------|-------|------------|
| P.        | <u>L</u> |      | Milch | $\geq 0.5$ |
|           | <u>L</u> |      | Milch | $\leq 0.6$ |

- Variante mit Condition-Box

| Lieferant | LName | LAdr | Ware  | Preis    |
|-----------|-------|------|-------|----------|
| P.        |       |      | Milch | <u>p</u> |

| CONDITIONS                               |
|--|
| $\_p \geq 0.5 \text{ AND } \_p \leq 0.6$ |



# Anfragen mit Negation

- Finde für jede Ware den billigsten Lieferanten

| Lieferant | LName | LAdr | Ware | Preis |
|-----------|-------|------|------|-------|
| P.        |       |      | _W   | _p    |
| _         |       |      | _W   | < _p  |

- Das Symbol  $\neg$  in der ersten Spalte bedeutet:  
Es gibt kein solches Tupel
- Bedeutung:  
 $\{x_1, x_2, w, p \mid \neg \exists y_1, y_2, y_3 : \text{Lieferant}(x_1, x_2, w, p) \wedge \text{Lieferant}(y_1, y_2, w, y_3) \wedge y_3 < p\}$



# 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

| <b>QBE</b>        | <b>Bereichskalkül</b>  |
|-------------------|--|
| Konstanten        | Konstanten   |
| Bereichsvariablen | Bereichsvariablen  |
| leere Spalten     | paarweise verschiedene<br>Bereichsvariablen,<br>$\exists$ -quantifiziert |
| Spalten mit P.    | freie Variablen  |
| Spalten ohne P.   | $\exists$ -quantifizierte Variablen                                      |

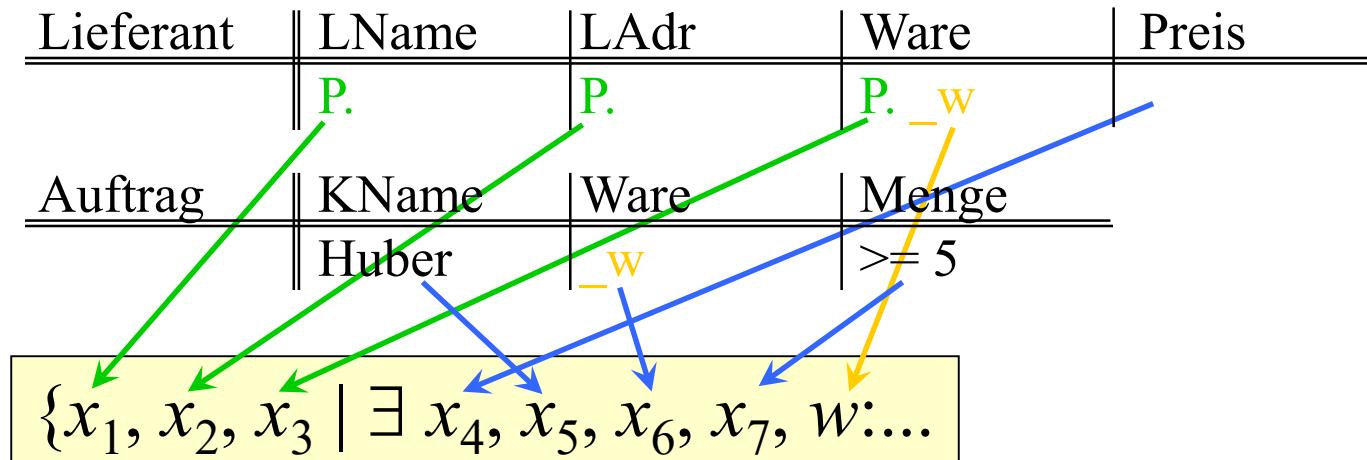
Anmerkung: QBE ist relational vollständig, jedoch ist für manche Anfragen der relationalen Algebra eine Folge von QBE-Anfragen nötig



# 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 ( $\{ \dots x_i, \dots | \dots \}$ ), sonst binde  $x_i$  mit einem  **$\exists$ -Quantor** ( $\{ \dots | \exists \dots, x_i, \dots \}$ )
- Binde alle Variablen der Anfrage mit einem  **$\exists$ -Quantor**





# Umsetzung einer QBE-Anfrage

| Lieferant | LName | LAdr | Ware     | Preis |
|-----------|-------|------|----------|-------|
|           | P.    | P.   | P. _w    |       |
| Auftrag   | KName | Ware | Menge    |       |
|           | Huber | _w   | $\geq 5$ |       |

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

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

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

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



# Umsetzung einer QBE-Anfrage

| Lieferant | LName | LAdr     | Ware        | Preis |
|-----------|-------|----------|-------------|-------|
|           | P.    | P.       | P. <u>w</u> |       |
| Auftrag   | KName | Ware     | Menge       |       |
|           | Huber | <u>w</u> | $\geq 5$    |       |

- Gleiches Vorgehen bei Zusätzen der Form Variable bzw.  $\leq$  Variable usw:

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

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

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



# Quantoren und Subqueries in SQL

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



# Quantoren und Subqueries in SQL

- 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  $\exists$ -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:

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

- 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** L2.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 Kurstabellen

```
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:
  - $A_i \Theta \text{all} (\text{select...from...where...})$        $\forall$ -Quantor
  - $A_i \Theta \text{some} (\text{select...from...where...})$
  - $A_i \Theta \text{any} (\text{select...from...where...})$
- Vergleichsoperatoren  $\Theta \in \{ =, <, \leq, >, \geq, \neq \}$
- Bedeutung:
  - $A_i \Theta \text{all} (\text{Subquery}) \equiv \{ \dots | \forall t \in \text{Subquery}: A_i \Theta t \}$
  - ist größer als **alle** Werte, die sich aus Subquery ergeben
- Einschränkung bezüglich Subquery:
  - Darf nur ein Ergebnis-Attribut ermitteln
  - Aber mehrere Tupel sind erlaubt



# 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 k1
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 $\leq$ 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)**
  - bzw.: **select \* from MagicNumbers where**  
**not (Wert = any (select Zahl from Primzahlen))**



# Typische Form der Subquery

- Bei **exists** bzw. **not exists** ist für die Haupt-Query nur relevant, ob das Ergebnis die leere Menge ist oder nicht.
  - Deshalb muss keine Projektion durchgeführt werden:  
**select ... from ... where exists (select \* from ...)**
- Bei **some**, **any**, **all** und **in** ist das Ergebnis der Subquery eine *Menge von Werten* (d.h. ein Attribut, mehrere Tupel), die in die Hauptquery eingesetzt werden.
  - Deshalb muss in der Subquery eine Projektion auf *genau ein* Attribut durchgeführt werden:  
**select ... from ... where A <= all (select B from ...)**
- Das Ergebnis der direkten Subquery ist *genau ein* Wert.
  - Projektion auf ein Attribut, Selektion eines Tupels:  
**... where A <= (select B from ... where Schlüssel=...)**



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 5: Sortieren, Gruppieren und Views in SQL

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani

Skript © 2019 Christian Böhm



# Sortieren

- In SQL mit ORDER BY  $A_1, A_2, \dots$
- Bei mehreren Attributen: Lexikographisch

|   |   | order by A, B |   |   |   | order by B, A |   |  |  |
|---|---|---------------|---|---|---|---------------|---|--|--|
| A | B | A             | B | A | B | A             | B |  |  |
| 1 | 1 | 1             | 1 |   |   | 1             | 1 |  |  |
| 3 | 1 | 2             | 2 |   |   | 3             | 1 |  |  |
| 2 | 2 | 3             | 1 |   |   | 4             | 1 |  |  |
| 4 | 1 | 3             | 3 |   |   | 2             | 2 |  |  |
| 3 | 3 | 4             | 1 |   |   | 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. Kapitel 4, Folie 43)



# 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

Aggregationen:

| <u>PNr</u> | Name   | Vorname | Abteilung | Gehalt | $\Sigma$ Gehalt | COUNT |
|------------|--------|---------|-----------|--------|-----------------|-------|
| 001        | Huber  | Erwin   | 01        | 2000   |                 |       |
| 002        | Mayer  | Hugo    | 01        | 2500   | 6300            | 3     |
| 003        | Müller | Anton   | 01        | 1800   |                 |       |
| 004        | Schulz | Egon    | 02        | 2500   |                 |       |
| 005        | Bauer  | Gustav  | 02        | 1700   | 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   $A_1, A_2, \dots$ 
 [having    ...]]       ← siehe Seite 13ff.
[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, z.B. **count (\*)**
  - in der Regel **select \* from...group by...** nicht erlaubt



# Gruppierung

- Beispiel: Nicht möglich!!!

Mitarbeiter

| <u>PNr</u> | 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~~

| <del>„PNr“</del>         | Abteilung | Gehalt |
|--------------------------|-----------|--------|
| <del>„001,002,003“</del> | 01        | 6300   |
| <del>„004,005“</del>     | 02        | 4200   |



# Gruppierung mehrerer Attribute

- Etwa sinnvoll in folgender Situation:

| <u>PNr</u> | Name   | Vorname | Abteilung | Einheit | 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, Einheit,  
            sum(Gehalt)  
from        Mitarbeiter  
group by    Abteilung, Einheit
```

| Abt. | Ei. | Σ Geh. |
|------|-----|--------|
| 01   | 01  | 2000   |
| 01   | 02  | 4300   |
| 02   | 01  | 4200   |



# Gruppierung mehrerer Attribute

Oft künstlich wegen **select**-Einschränkung:

Mitarbeiter  $\bowtie$  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
group by    ANr
having     count (*) >= 5
```

GEHT NICHT !

STATT DESSEN:
- Grund: Gruppierung wird erst nach SELECT-FROM-WHERE-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. Schritt:

**Aggregation**

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



# 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:

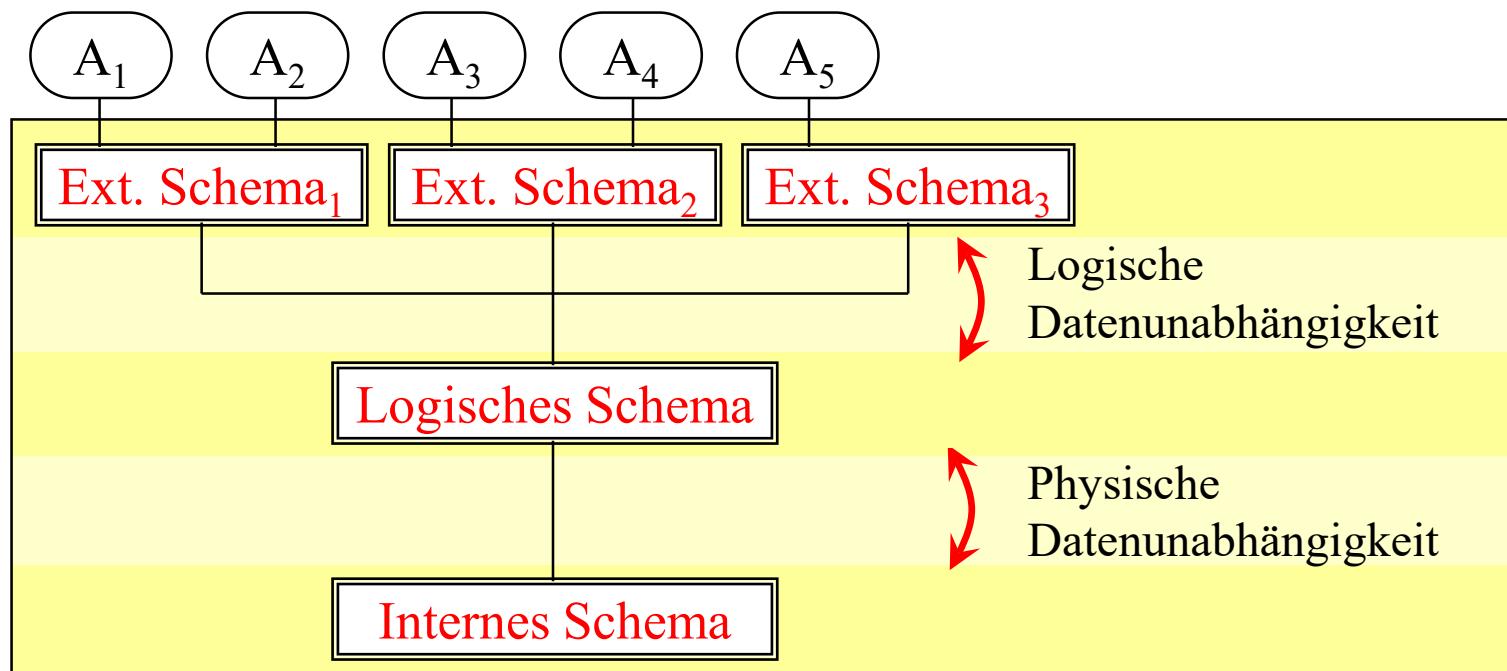
**Projektion**

| A | sum(D) |
|---|--------|
| 1 | 9      |



# 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<sub>1</sub>, Relation<sub>2</sub>, ... 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* [( $A_1, A_2, \dots$ )]\* 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   |



# 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:  
**update 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 übersetzbare:  
**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**), kein Standard-SQL

Bei dem Begriff *View* meinen wir *nicht* materialisierte Views



# Rechtevergabe

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

```
grant Rechteleiste
on Relation
to Benutzerliste
[with grant option]
```

- *Rechteleiste*:
  - 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*



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 6: Das E/R-Modell

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani  
Skript © 2019 Christian Böhm



# 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 (...);`

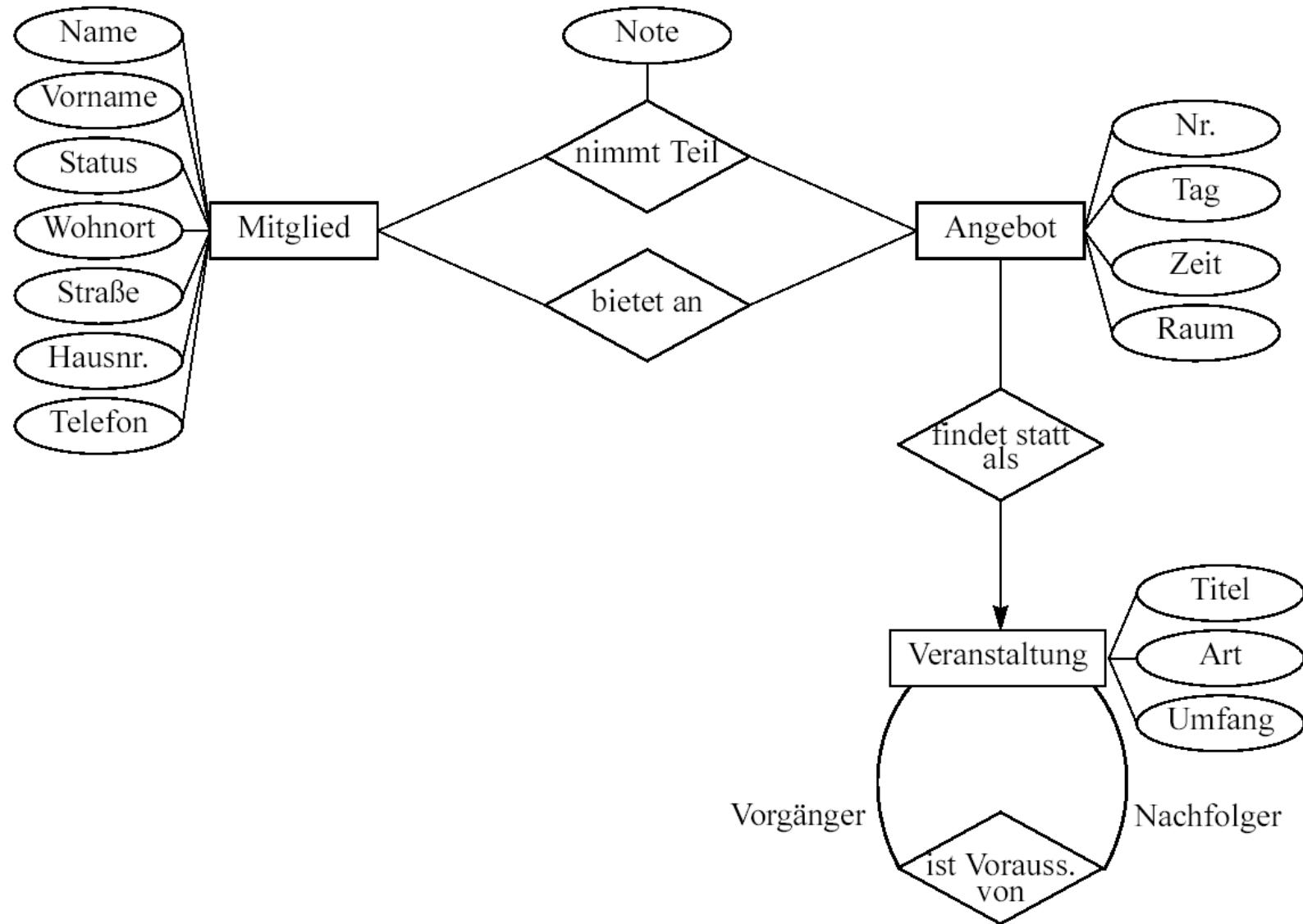


# Das Entity/Relationship Modell

- 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

Student

Name

belegt

Entscheidende Aufgabe des Schema-Entwurfs:

- Finde *geeignete* Entities, Attribute und Relationships



# Entities und Attribute

## Entities

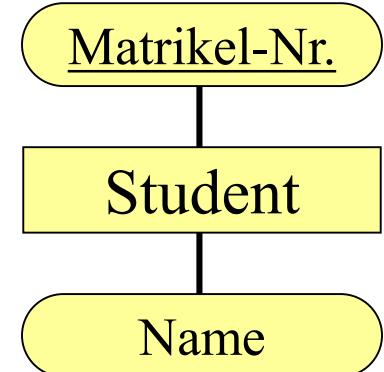
- Objekte, Typen, "Seiendes"
- Objekte der realen Welt, unterscheidbar
- Bsp: Mensch, Haus, Vorlesung, Bestellung, ...

## 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



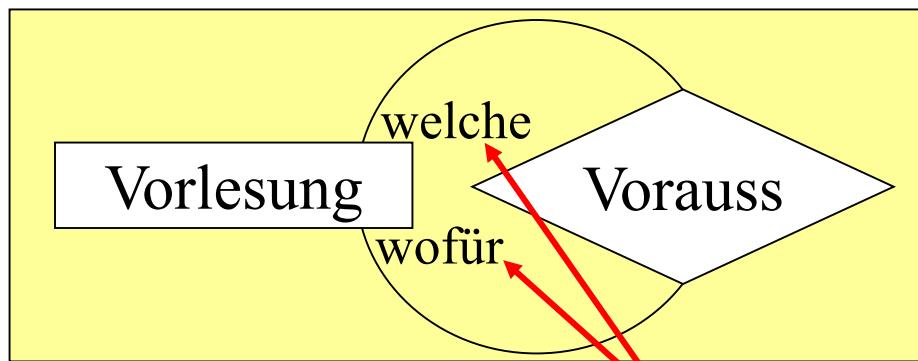


# Relationships (Beziehungen)

- Stellen Zusammenhänge zwischen Entities dar
- Beispiele:



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



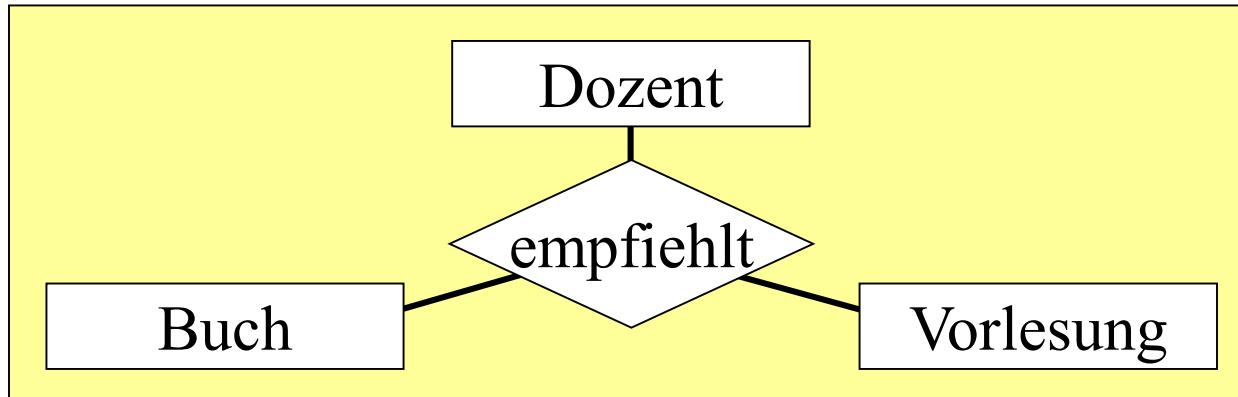
Ausprägung:  
Vorauss (I 1, DBS 1)  
Vorauss (I 1, Software-Eng.)  
Vorauss (DBS 1, DBS 2)  
Vorauss (I 1, Wissensrepr.)

7  
unterschiedliche Rollen



# Relationships (Beziehungen)

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

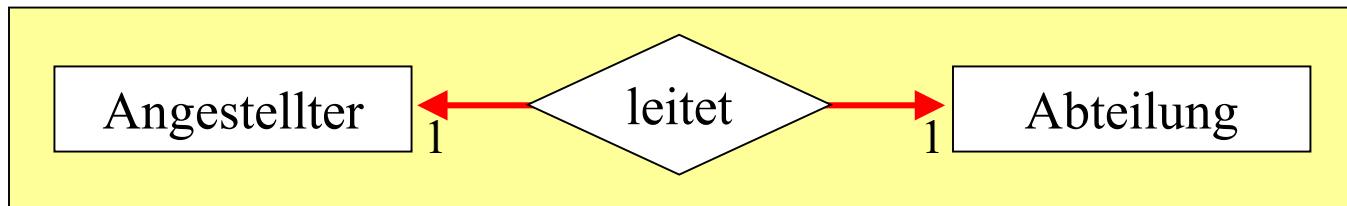


Ausprägung: empfiehlt (Böhm, Heuer&Saake, DBS 1)  
empfiehlt (Böhm, Kemper&Eickler, DBS 1)  
empfiehlt (Böhm, Bishop, Informatik 1)  
empfiehlt (Böhm, Bishop, Programmierkurs)



# Funktionalität von Relationships

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

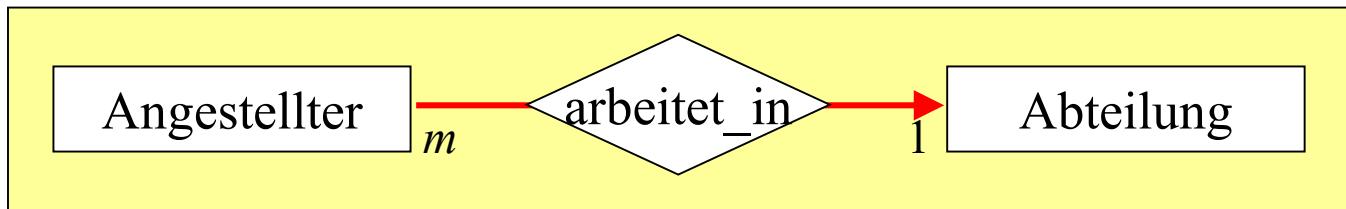


- Charakteristik:  
Jedes Objekt aus dem linken Entity steht in Beziehung zu **höchstens** einem Objekt aus dem rechten Entity und umgekehrt.
- Grafische Notation: Pfeile auf jeder Seite
- Beispiel gilt unter der Voraussetzung, dass jede Abteilung max. einen Leiter hat und kein Angestellter mehrere Abteilungen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



# Funktionalität von Relationships

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

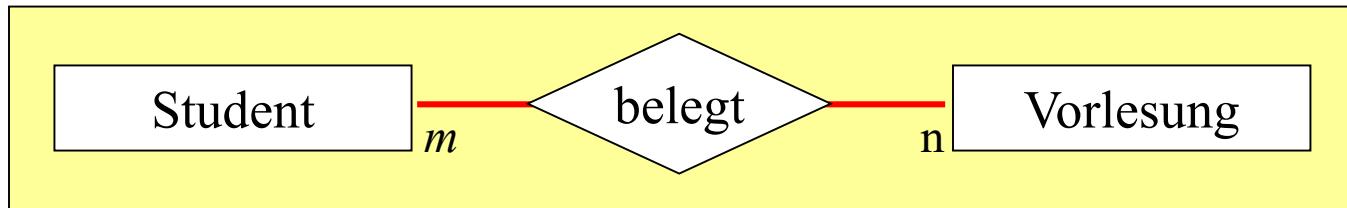


- Charakteristik:  
Jedes Objekt auf der "*many*"-Seite steht in Beziehung zu höchstens einem Objekt auf der "*one*"-Seite (im Allgemeinen nicht umgekehrt)
- Grafische Notation:  
Pfeil in Richtung zur "*one*"-Seite



# Funktionalität von Relationships

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



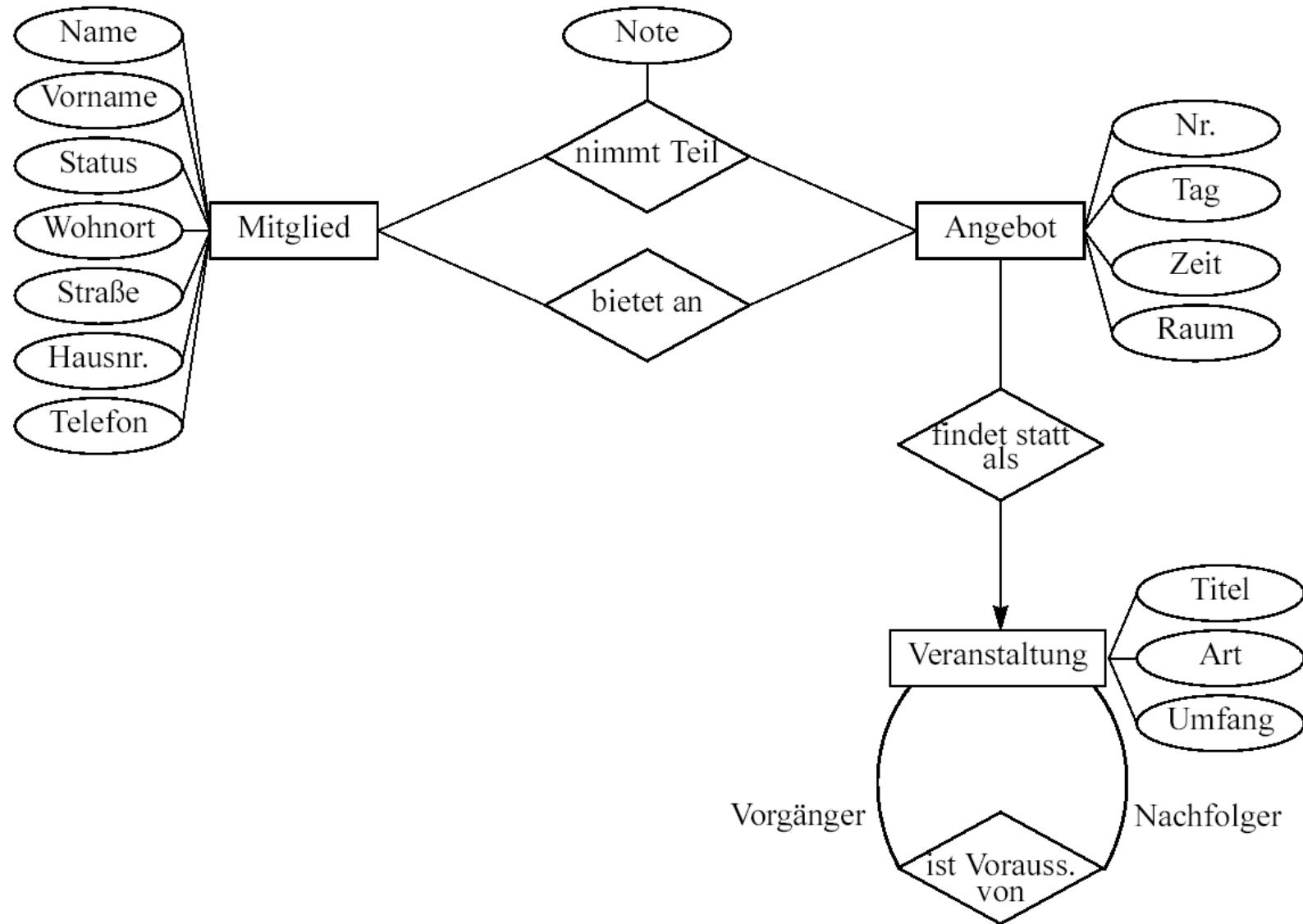
- Charakteristik:  
Jedes Objekt auf der linken Seite kann zu mehreren Objekten auf der rechten Seite in Beziehung stehen (d.h. keine Einschränkung)
- Grafische Notation:  
Kein Pfeil

**Beispiel-Ausprägung:**

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



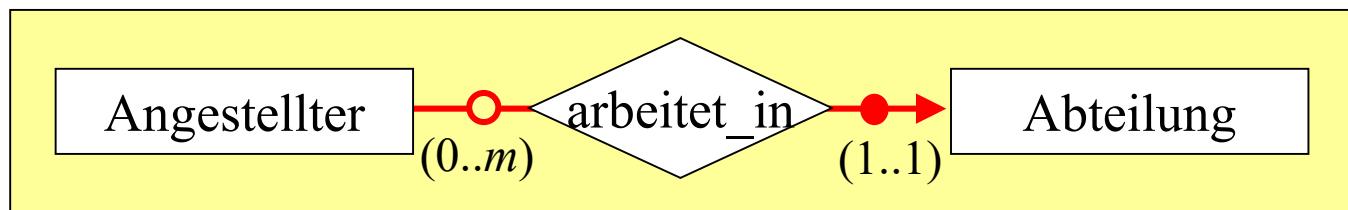
# Beispiel: Lehrveranstaltungen





# Optionalität

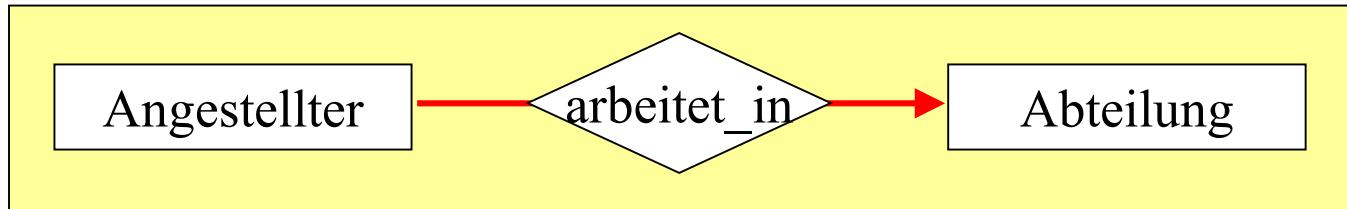
- Das E/R-Modell trifft lediglich Aussagen darüber, ob ein Objekt zu mehreren Objekten in Beziehung stehen darf oder zu max. einem
- Darüber, ob es zu mindestens einem Objekt in Beziehung stehen muss, keine Aussage
- Erweiterung der Notation mit:
  - Geschlossener Kreis: Verpflichtend mindestens eins.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



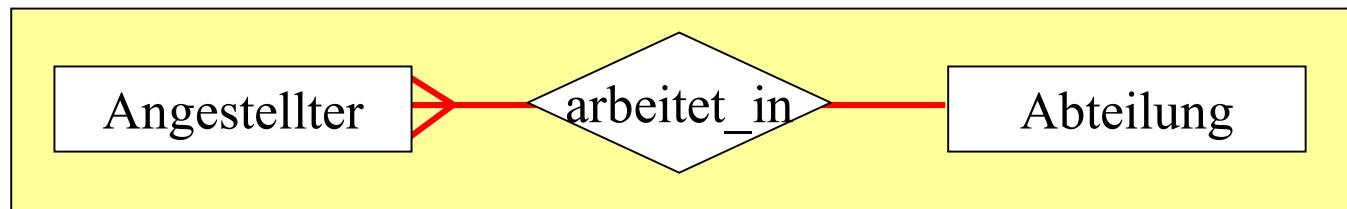


# Verschiedene Notationen

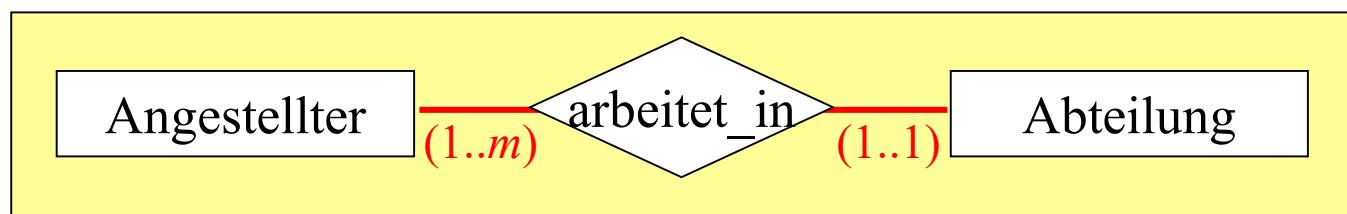
- Pfeil-Notation der Funktionalität:



- Krähenfuß-Notation:



- Kardinalitäts-Notation:

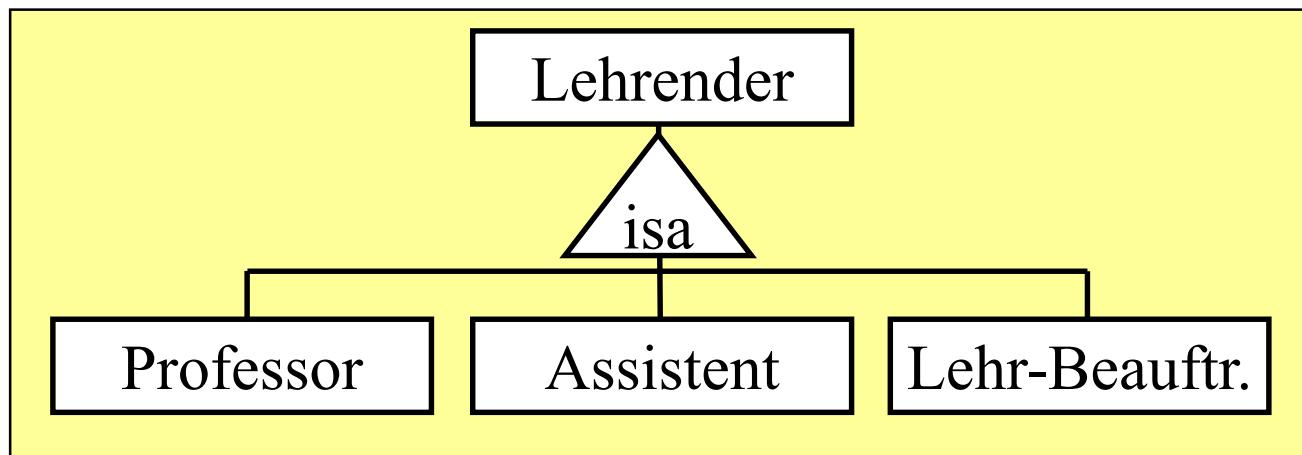


auch  $m$ ,  $(1..\infty)$ ,  $(1..*)$ , verschiedene Kombinationsformen



# ISA-Beziehung/Vererbung

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



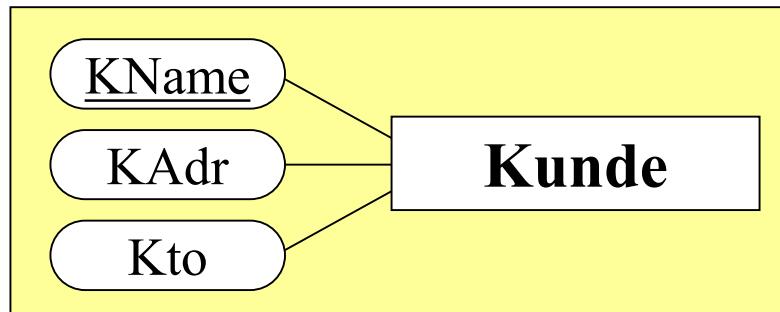
- Charakteristik und Bedeutung:
  - Assistent ist ein Lehrender (engl.: is a)
  - Vererbung aller Attribute und Beziehungen



# 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:
    - $1:1$
    - $1:n$
    - $n:m$
- Zusätzliche Attribute in bestehende Relationen
- 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:



- Eine zusätzliche Tabelle wird nicht 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, ...)

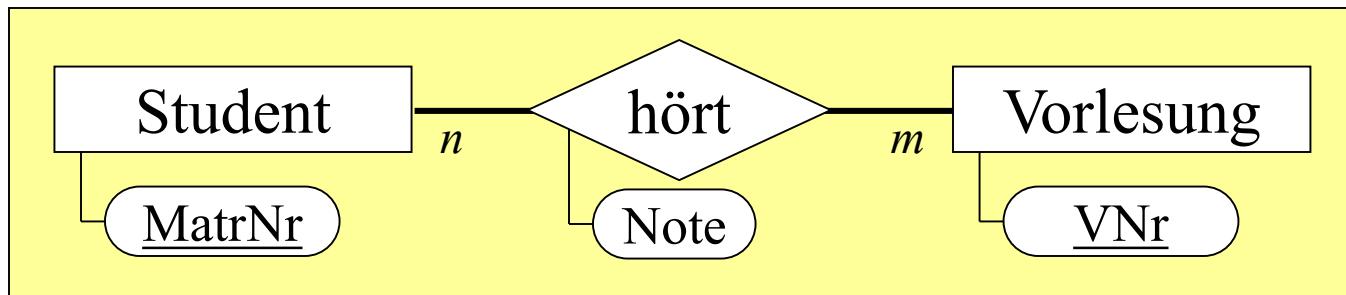
Mitarbeiter (PNr, Name, Vorname, ..., ANr)

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



# Vom E/R-Modell zur Relation

- Many-To-Many-Relationships:

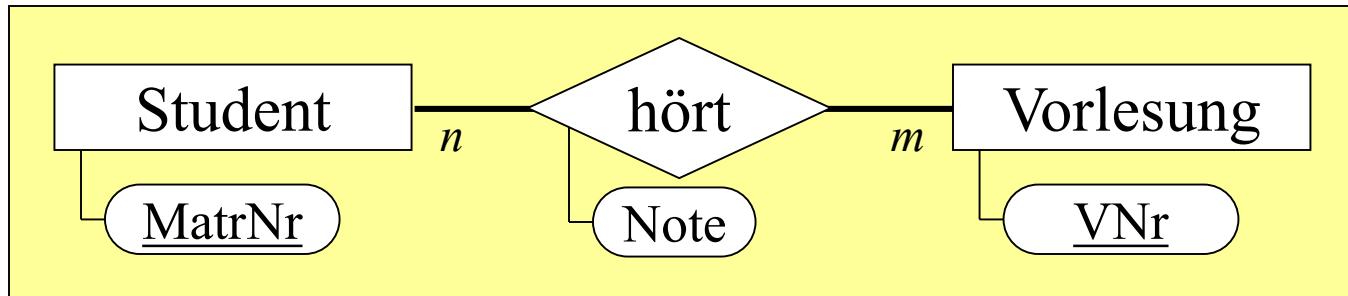


- Einführung einer zusätzlichen Relation mit dem Namen der Relationship
- Attribute: Die Primärschlüssel-Attribute der Relationen, den Entities beider Seiten zugeordnet sind
- Diese Attribute sind jeweils Fremdschlüssel
- Zusammen sind diese Attribute Primärschlüssel der neuen Relation
- Attribute der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

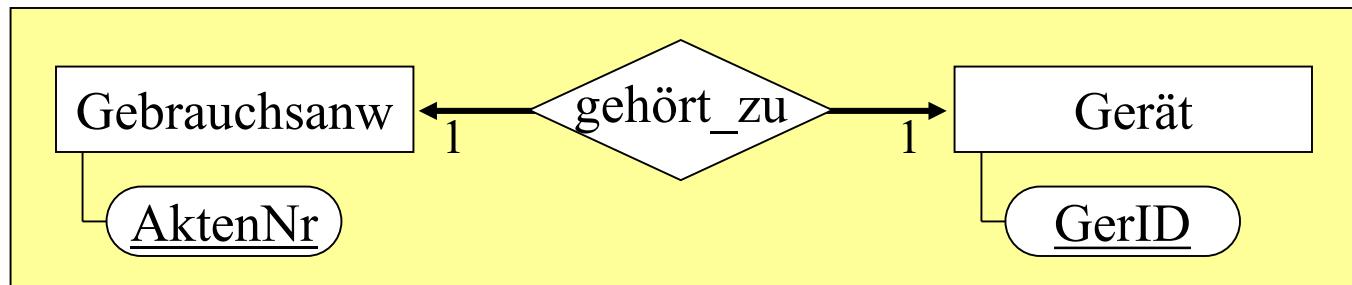
**foreign key** MatrNr **references** Student,

**foreign key** VNr **references** Vorlesung...



# Vom E/R-Modell zur Relation

- One-To-One-Relationships:

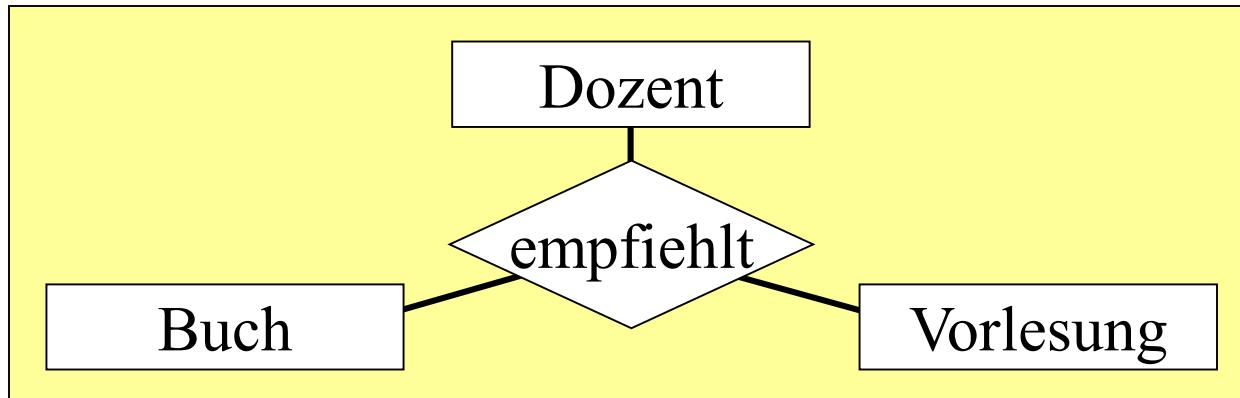


- Die beiden Entities werden zu einer Relation zusammengefasst
- Einer der Primärschlüssel der Entities wird Primärschlüssel der Relation
  - Geraet (GerID, ..., AktenNr, ...)
- Häufig auch Umsetzung wie bei 1:n-Beziehung (insbes. wenn eine Seite optional ist), wobei die Rollen der beteiligten Relationen 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, ...)

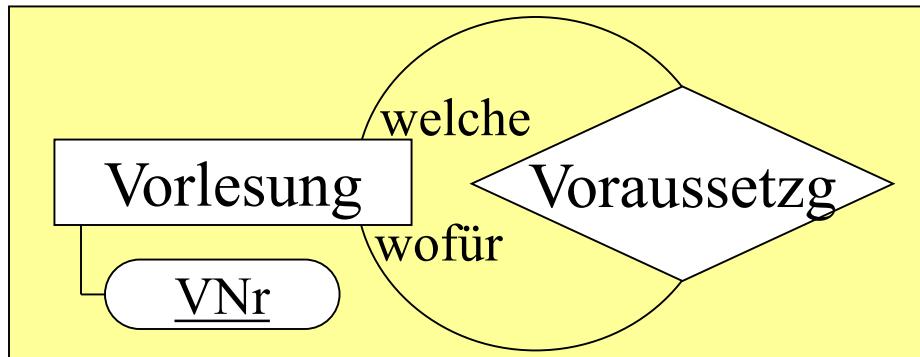
Vorlesung (VNr, ...)

empfiehlt (PNr, ISBN, VNr)



# Vom E/R-Modell zur Relation

- Selbstbezug



Keine Besonderheiten:  
Vorgehen je nach Funktionalität.

Vorlesung (VNr, ...)

Voraussetzg (Welche, Wofuer)

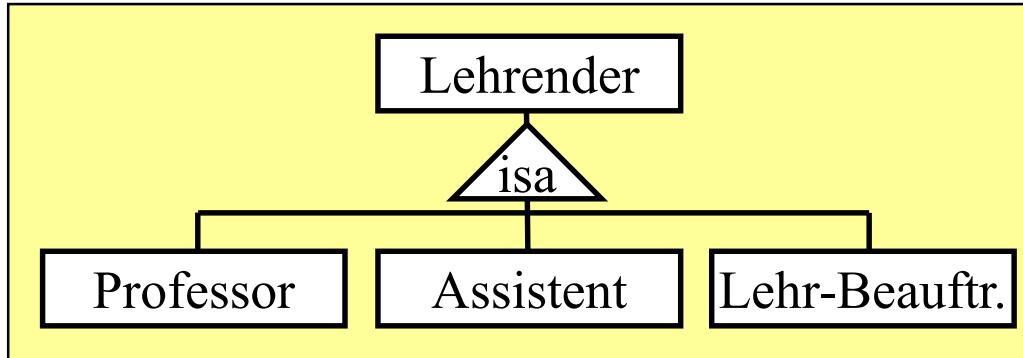
**foreign key** Welche **references** Vorlesung (VNr),

**foreign key** Wofuer **references** Vorlesung (VNr)

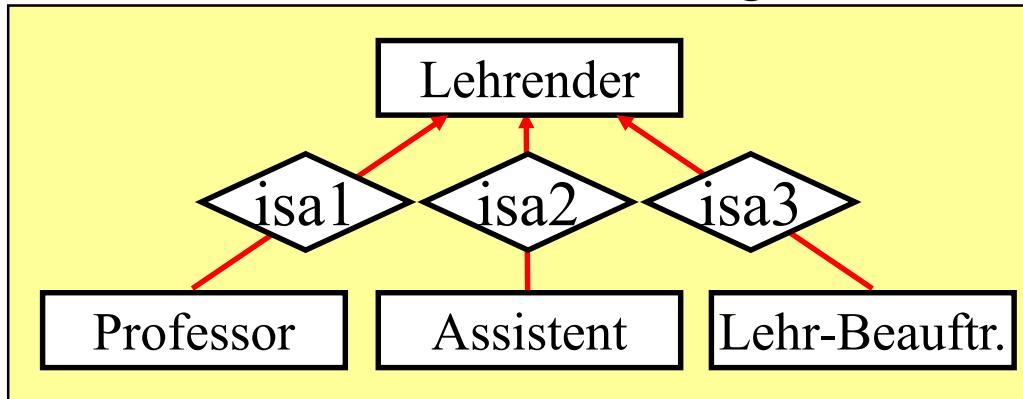


# Vom E/R-Modell zur Relation

- 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)





Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 7: Normalformen

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani

Skript © 2019 Christian Böhm, LMU



# 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

- **Relation Lieferant:**

| LNr | LName | LStadt | LLand       | Ware     | Preis |
|-----|-------|--------|-------------|----------|-------|
| 103 | Huber | Berlin | Deutschland | Schraube | 50    |
| 103 | Huber | Berlin | Deutschland | Draht    | 20    |
| 103 | Huber | Berlin | Deutschland | Nagel    | 40    |
| ... | ...   | ...    | ...         | ...      | ...   |
| 762 | Maier | Zürich | Schweiz     | Nagel    | 45    |
| 762 | Maier | Zürich | Schweiz     | Schraube | 55    |

- Hier gibt es offensichtlich einige Redundanzen:
  - Wenn bei mehreren Tupeln der Attributwert LNr gleich ist, dann müssen auch die Attributwerte von LName, LStadt und LLand gleich sein.
  - Wenn (auch bei verschiedenen LNr) LStadt gleich ist, dann muss auch LLand gleich sein
- Redundanzen durch funktionale Abhängigkeiten
  - Wir sagen: Die Attribute LName, LStadt und LLand sind **funktional abhängig** vom Attribut LNr (ebenso LLand von LStadt); Exakte Definition siehe Seite 8ff.



# Motivation Normalisierung

- **Relation Lieferant:**

| LNr | LName | LStadt | LLand       | Ware     | Preis |
|-----|-------|--------|-------------|----------|-------|
| 103 | Huber | Berlin | Deutschland | Schraube | 50    |
| 103 | Huber | Berlin | Deutschland | Draht    | 20    |
| 103 | Huber | Berlin | Deutschland | Nagel    | 40    |
| ... | ...   | ...    | ...         | ...      | ...   |
| 762 | Maier | Zürich | Schweiz     | Nagel    | 45    |
| 762 | Maier | Zürich | Schweiz     | Schraube | 55    |

- Redundanzen führen zu Speicherplatzverschwendung;
- Das eigentliche Problem sind aber Anomalien (Inkonsistenzen durch Änderungsoperationen) und dass das Schema nicht intuitiv ist:
  - **Update-Anomalie:** Änderung der Adresse (LName, LStadt, LLand) in nur *einem* Tupel statt in *allen* Tupeln zu einer LNr.
  - **Insert-Anomalie:** Einfügen eines Tupels mit inkonsistenter Adresse; Einfügen eines Lieferanten erfordert Ware.
  - **Delete-Anomalie:** Löschen der letzten Ware löscht die Adresse.



# Verbesserung

- Neues Datenbankschema:

|                     |                                     |
|---------------------|-------------------------------------|
| <b>LieferantAdr</b> | ( <u>LNr</u> , LName, LStadt)       |
| <b>Stadt</b>        | ( <u>LStadt</u> , LLand)            |
| <b>Angebot</b>      | ( <u>LNr</u> , <u>Ware</u> , Preis) |

- Vorteile:
  - keine Redundanz
  - keine Anomalien
- Nachteil:
  - Um zu einer Ware die Länder der Lieferanten zu finden, ist ein zweifacher 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.LNr, LName, L.LStadt, LLand, Ware, Preis
  from        LieferantAdr L, Stadt S, Angebot A
  where       L.LNr = A.LNr and L.LStadt = S.LStadt
```

- Dies spart die Schreibarbeit beim Formulieren von Anfragen und macht die Anfragen übersichtlicher.
- In diesem Fall ist die View *nicht updatable* (die Änderungsoperationen verursachen ja die Probleme).
- Da die View nur eine *virtuelle Relation* ist, muss der Join trotzdem ausgewertet werden (weniger effizient).



# 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 Abhangigkeit

(engl. Functional Dependency, FD)

- beschreibt Beziehungen zwischen den Attributen einer Relation
- Schrankt das Auftreten gleicher bzw. ungleicher Attributwerte innerhalb einer Relation ein
  - spezielle Integritatsbedingung (nicht in SQL)

Wiederholung Integritatsbedingungen in SQL:

- Primarschlssel
- Fremdschlssel (referentielle Integritat)
- **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



# Konventionen zur Notation

- Ab jetzt gilt die Notation:
  - $A, B, C$  bezeichnen einzelne Attribute
  - $X, Y, Z$  bezeichnen Mengen von Attributen
- Zur Vereinfachung gilt außerdem:
  - $A, B \rightarrow C$  bezeichne  $\{A, B\} \rightarrow \{C\}$
  - $X \rightarrow Y, Z$  bezeichne  $X \rightarrow Y \cup Z$
  - $t.A$  bezeichne das Attribut  $A$  des Tupels  $t$
  - $t.X$  bezeichne die Menge  $X$  von Attributen des Tupels  $t$
  - $t.X = r.X$  bezeichne  $\forall A \in X : t.A = r.A$



# Definition: *funktional abhängig*

- **Gegeben:**
  - Ein Relationenschema  $R$
  - $X, Y$ : Zwei *Mengen* von Attributen von  $R$  ( $X, Y \subseteq R$ )
- **Definition:**

$Y$  ist von  $X$  funktional abhängig ( $X \rightarrow Y$ )  
gdw.  $\forall$  Tupel  $t$  und  $r : t.X = r.X \Rightarrow t.Y = r.Y$

(für alle möglichen Ausprägungen von  $R$  gilt:  
Zu jedem Wert in  $X$  existiert genau ein Wert von  $Y$ .)
- Bsp.: **Lieferant** (LNr, LName, LStadt, LLand, Ware, Preis):
  - $\text{LNr} \rightarrow \text{LName}$
  - $\text{LNr} \rightarrow \text{LStadt}$
  - $\text{LStadt} \rightarrow \text{LLand}$
  - $\text{LNr}, \text{Ware} \rightarrow \text{LName}$
  - $\text{LNr}, \text{Ware} \rightarrow \text{LStadt}$
  - $\text{LNr}, \text{Ware} \rightarrow \text{Preis}$



# Vergleich mit *Schlüssel*

- Gemeinsamkeiten zwischen dem *Schlüssel* im relationalen Modell und *Funktionaler Abhängigkeit*:
  - Für jeden Schlüsselkandidaten  $S = \{A, B, \dots\}$  gilt:  
Alle Attribute der Relation sind von  $S$  funktional abhängig (wegen der Eindeutigkeit):
$$S \rightarrow R$$
  - Jede Menge, von der  $R$  FD ist, ist Superschlüssel.
- 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 eines Schlüssels

→ FD ist Verallgemeinerung des **Schlüssel-Konzepts**.



# Vergleich mit *Schlüssel*

- 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

## Prime Attribute

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



# 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  $X \rightarrow Y$
  - Wenn es keine echte Teilmenge  $X' \subsetneq X$  gibt, von der  $Y$  ebenfalls funktional abhängt,
  - dann heißt  $X \rightarrow Y$  eine **volle funktionale Abhängigkeit ( $X \xrightarrow{\bullet} Y$ )**
  - andernfalls eine **partielle funktionale Abhängigkeit**



# Partielle und volle FD

- Beispiele:
  - $\text{LNR} \rightarrow \text{LName}$  voll funktional abhängig  
(ein-elementige Menge ist immer minimal)
  - $\text{LNR, Ware} \rightarrow \text{LName}$  partiell funktional abhängig  
(da schon  $\text{LNR} \rightarrow \text{LName}$  gilt)
  - $\text{Ware } ? \text{ Preis}$  nicht funktional abhängig (siehe Beispiel)
  - $\text{LNR } ? \text{ Preis}$  nicht funktional abhängig (siehe Beispiel)
  - $\text{LNR, Ware} \rightarrow \text{Preis}$  voll funktional abhängig (weder von LNR noch von Ware alleine funktional abhängig)



# Volle FD vs Minimalität des Schlüssels

- Definitionen sehr ähnlich:
  - Schlüssel: Minimale Menge, die Eindeutigkeit erfüllt
  - Volle FD: Minimale Menge, von der ein Attribut abhängt  
(Und die Eindeutigkeit ist äquivalent zur FD, s. S. 12)
- Frage: Folgt aus der Minimalität des Schlüssels, dass alle Attribute (immer) voll funktional abhängig sind?
- Antwort: Leider nicht. (Warum eigentlich „leider“?)
- Aber wo liegt der Denkfehler?  
Der Allquantor ist bei beiden Definitionen verschieden:
  - Schlüssel: Minimale Menge, von der alle Attribute funktional abhängig sind.
  - Volle FD  $X \rightarrow Y$ : Für jedes Attribut  $A \in Y$  gilt:  $X$  ist die minimale Menge von der  $A$  funktional abhängig ist.



# Herleitung funktionaler Abhangigkeit

## Armstrong Axiome

- Reflexivitat (R): Falls  $Y$  eine Teilmenge von  $X$  ist ( $Y \subseteq X$ ), dann gilt immer  $X \rightarrow Y$ . Insbesondere gilt also immer  $X \rightarrow X$ .
- Verstarkung (VS): Falls  $X \rightarrow Y$  gilt, dann gilt auch  $XZ \rightarrow YZ$ . Hierbei steht  $XZ$  fur  $X \cup Z$ .
- Transitivitat (T): Falls  $X \rightarrow Y$  und  $Y \rightarrow Z$  gilt, dann gilt auch  $X \rightarrow Z$ .

Diese Axiome sind **vollstandig** und **korrekt**:

Sei  $F$  eine Menge von FDs:

- Es lassen sich nur FDs von  $F$  ableiten, die von jeder Relationen-Auspragung erfullt werden, fur die auch  $F$  erfullt ist.
- Es sind alle FDs ableitbar, die durch  $F$  impliziert sind.



# Herleitung funktionaler Abhangigkeit

## Triviale funktionale Abhangigkeit:

- Wegen Reflexivitat ist jedes Attribut funktional abhangig:
  - von sich selbst
  - von jeder Obermenge von sich selbst

Solche Abhangigkeiten bezeichnet man als **trivial**.

## Symmetrieeigenschaften funktionaler Abhangigkeiten

- Bei den funktionalen Abhangigkeiten gibt es kein Gesetz zur Symmetrie oder Antisymmetrie, d.h. bei zwei bel. Attribut (-Mengen)  $X, Y$  sind alle 4 Falle moglich:
  - Es gilt nur  $X \rightarrow Y$ ,
  - Es gilt nur  $Y \rightarrow X$ ,
  - Es gilt  $X \rightarrow Y$  und  $Y \rightarrow X$ ,
  - Es gibt keine FD. zw.  $X$  und  $Y$ .



# Hülle einer Attributmenge

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

$\text{AttrHülle}(F, X)$

Erg := X

**while** ( Änderungen an Erg) do

**foreach** FD  $Y \rightarrow Z \in F$  **do**

**if**  $Y \subseteq \text{Erg}$  **then**  $\text{Erg} := \text{Erg} \cup Z$

Ausgabe  $X^+ = \text{Erg}$



# Hülle einer Attributmenge

- **Beispiel:** AttrHülle( $F$ ,  $\{\text{LNr}\}$ ) mit  
 $F = \{\text{LNr} \rightarrow \text{LName}; \text{LNr} \rightarrow \text{LStadt}; \text{LStadt} \rightarrow \text{LLand};$   
 $\text{LNr}, \text{Ware} \rightarrow \text{Preis}\}$
- $\text{Erg}_i = \text{Erg}$  nach  $i$ -tem Durchlauf der **while**-Schleife
  - $\text{Erg}_0 = \{\text{LNr}\}$
  - $\text{Erg}_1 = \{\text{LNr}, \text{LName}, \text{LStadt}\}$
  - $\text{Erg}_2 = \{\text{LNr}, \text{LName}, \text{LStadt}, \text{LLand}\}$
  - $\text{Erg}_3 = \{\text{LNr}, \text{LName}, \text{LStadt}, \text{LLand}\} = \text{Erg}_2$



# Herleitung funktionaler Abhangigkeit

## Weitere Regeln zu funktionalen Abhangigkeiten

(vereinfachen Herleitungsprozess, aber nicht notwendig,  
da Armstrong-Axiome vollstandig sind):

- Vereinigungsregel (VE):  
Falls  $X \rightarrow Y$  und  $X \rightarrow Z$  gilt, dann gilt auch  $X \rightarrow YZ$ .
- Dekompositionsregel (D):  
Falls  $X \rightarrow YZ$  gilt, dann gilt auch  $X \rightarrow Y$  und  $X \rightarrow Z$ .
- Pseudotransitivitatsregel (P):  
Falls  $X \rightarrow Y$  und  $ZY \rightarrow V$  gilt, dann gilt auch  $XZ \rightarrow V$ .



# Herleitung funktionaler Abhangigkeit

**Beispiel:** Zeige, dass  $(\text{LNR}, \text{Ware})$  Schlsselkandidat ist.

- Gegeben:  $F = \{ \text{LNR} \rightarrow \text{LName}; \text{LNR} \rightarrow \text{LStadt}; \text{LStadt} \rightarrow \text{LLand}; \text{LNR}, \text{Ware} \rightarrow \text{Preis} \}.$
- Zu zeigen:  $(\text{LNR}, \text{Ware})$  eindeutig und minimal.
- Beweis:
  - (I)  $(\text{LNR}, \text{Ware})$  eindeutig gdw.  
 $(\text{LNR}, \text{Ware} \rightarrow \text{LNR}, \text{LName}, \text{LStadt}, \text{LLand}, \text{Ware}, \text{Preis}) \in F^+,$   
d.h. ist aus  $F$  herleitbar.
  - (a)  $(\text{LNR} \rightarrow \text{LName}) \in F \subseteq F^+ \xrightarrow{\text{VS}} (\text{LNR}, \text{Ware} \rightarrow \text{LName}, \text{Ware}) \in F^+$
  - (b)  $(\text{LNR} \rightarrow \text{LStadt}) \in F^+ \xrightarrow{\text{VS}} (\text{LNR}, \text{Ware} \rightarrow \text{LStadt}, \text{Ware}) \in F^+$
  - (c)  $(\text{LNR} \rightarrow \text{LStadt}) \in F^+$  und  $(\text{LStadt} \rightarrow \text{LLand}) \in F^+$   
 $\xrightarrow{\text{T}} (\text{LNR} \rightarrow \text{LLand}) \in F^+ \xrightarrow{\text{VS}} (\text{LNR}, \text{Ware} \rightarrow \text{LLand}, \text{Ware}) \in F^+$
  - (d) Wegen R gilt  $(\text{LNR} \rightarrow \text{LNR}) \in F^+ \xrightarrow{\text{VS}} (\text{LNR}, \text{Ware} \rightarrow \text{LNR}, \text{Ware}) \in F^+$
  - (e) aus (a) bis (e) und  $(\text{LNR}, \text{Ware} \rightarrow \text{Preis})$  folgt (I) nach VE, qed.



# Herleitung funktionaler Abhangigkeit

**Beispiel:** Zeige, dass  $(\text{LNr}, \text{Ware})$  Schlsselkandidat ist.

- Gegeben:  $F = \{ \text{LNr} \rightarrow \text{LName}; \text{LNr} \rightarrow \text{LStadt}; \text{LStadt} \rightarrow \text{LLand}; \text{LNr}, \text{Ware} \rightarrow \text{Preis} \}$ .
- Zu zeigen:  $(\text{LNr}, \text{Ware})$  eindeutig und minimal.
- Beweis:
  - (II)  $(\text{LNr}, \text{Ware})$  minimal gdw.  
 $(\text{LNr} \rightarrow \text{LNr}, \text{LName}, \text{LStadt}, \text{LLand}, \text{Ware}, \text{Preis})$  und  
 $(\text{Ware} \rightarrow \text{LNr}, \text{LName}, \text{LStadt}, \text{LLand}, \text{Ware}, \text{Preis})$  gelten nicht.
  - (a) Preis ist nur von  $\text{LNr}$  und  $\text{Ware}$  gemeinsam funktional abhangig.
  - (b) Weder  $\text{LNr}$  kann aus  $\text{Ware}$  hergeleitet werden, noch  $\text{Ware}$  von  $\text{LNr}$ .
  - (c) aus (a) und (b) folgt  $(\text{LNr}, \text{Ware})$  minimal, qed.



# Normalisierung

- In einem Relationenschema sollen also 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

*impliziert*  
*impliziert*  
*impliziert*  
*impliziert*



# 1. Normalform

- Keine Einschränkung bezüglich der FDs
- Ein Relationenschema ist in erster Normalform, wenn alle Attributwerte ***atomar*** sind
- In relationalen Datenbanken 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 | 5 |
|   |   | 6 | 7 |

„nested relation“  
non first normal form (NFNF, NF2)



In SQL nur temporär erlaubt  
(vgl. GROUP BY-Auswertung)



## 2. Normalform

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



| <u>LNr</u> | LName | LStadt | LLand       | Ware     | Preis |
|------------|-------|--------|-------------|----------|-------|
| 103        | Huber | Berlin | Deutschland | Schraube | 50    |
| 103        | Huber | Berlin | Deutschland | Draht    | 20    |
| 103        | Huber | Berlin | Deutschland | Nagel    | 40    |
| ...        | ...   | ...    | ...         | ...      | ...   |
| 762        | Maier | Zürich | Schweiz     | Nagel    | 45    |
| 762        | Maier | Zürich | Schweiz     | Schraube | 55    |

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, siehe Seite 44ff)
- Definition

Ein Schema ist in zweiter Normalform, wenn jedes Attribut
  - voll funktional abhängig von **allen** Schlüsselkandidaten oder
  - prim ist (d.h. Teil eines Schlüsselkandidaten)
- Beobachtung:

Zweite Normalform kann nur verletzt sein, wenn...

  - ...ein zusammengesetzter Schlüssel (-Kandidat) existiert
  - ...und wenn nicht-prime Attribute existieren

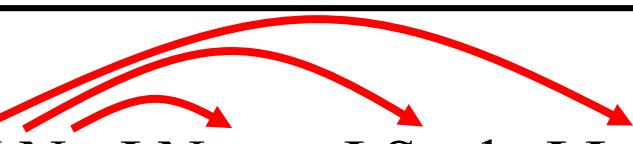


## 2. Normalform

- 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  $X_i \rightarrow Y_i$  von einem Teil eines Schlüssels ( $X_i \subsetneq S$ ) geht man folgendermaßen vor:
    - Lösche die Attribute  $Y_i$  aus  $R$
    - Gruppiere die Abhängigkeiten nach gleichen linken Seiten  $X_i$
    - Für jede Gruppe führe eine neue Relation ein mit allen enthaltenen Attributen aus  $X_i$  und  $Y_i$
    - $X_i$  wird Schlüssel in der neuen Relation



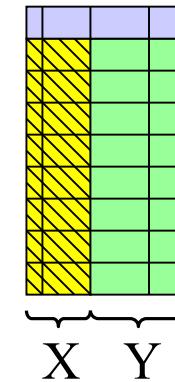
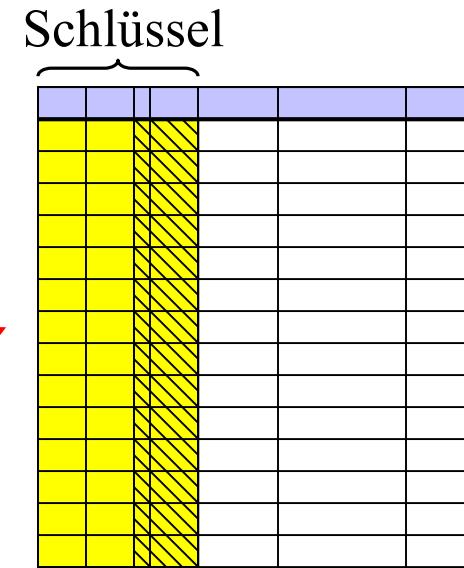
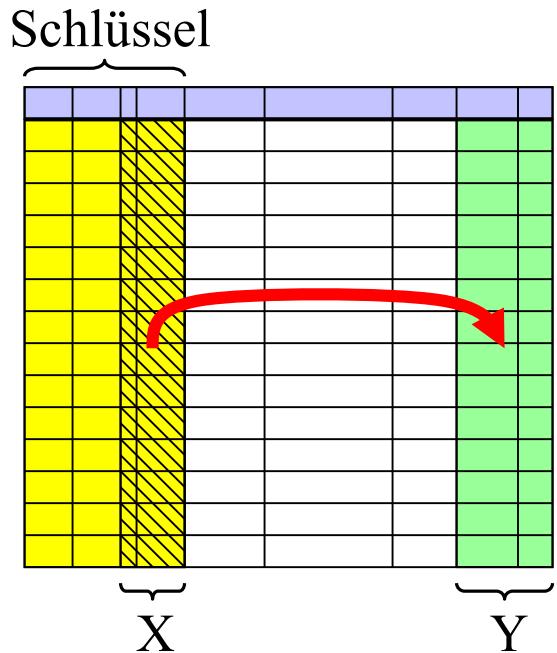
## 2. Normalform

- Beispiel:  
**Lieferant** (LNr, LName, LStadt, LLand, Ware, Preis)  
A red curved arrow originates from the LNr attribute and points to the Ware attribute. Another red curved arrow originates from the Ware attribute and points to the Preis attribute, illustrating partial dependencies.

partielle Abhangigkeiten
- Vorgehen:
  - LName, LStadt und LLand werden aus Lieferant geloscht
  - Gruppierung:  
Nur eine Gruppe mit LNr auf der linken Seite
    - es konnten Attribute von Ware abhangen (2. Gruppe)
  - Erzeugen einer Relation mit LNr, LName, LStadt und LLand
- Ergebnis:  
**Lieferung**      (LNr, Ware, Preis)  
**LieferAdr**      (LNr, LName, LStadt, LLand)



# Grafische Darstellung



(nach Heuer, Saake, Sattler)



# 3. Normalform

- Motivation:  
Man möchte zusätzlich verhindern, dass Attribute von nicht-primen Attributen funktional abhängig sind.
- Beispiel:

**LieferAdr**

(LNr, LName, LStadt, LLand)

|     |        |          |             |
|-----|--------|----------|-------------|
| 001 | Huber  | München  | Deutschland |
| 002 | Meier  | Berlin   | Deutschland |
| 003 | Müller | Berlin   | Deutschland |
| 004 | Hinz   | Salzburg | Österreich  |
| 005 | Kunz   | Salzburg | Österreich  |



- Redundanz: Land mehrfach gespeichert
- Anomalien?



# 3. Normalform

- Abhangigkeit von Nicht-Schlssel-Attribut bezeichnet man haufig auch als *transitive* (d.h. durch Armstrong-Transitivitatsaxiom hergeleitete) **Abhangigkeit** vom Primarschlssel
  - weil Abhangigkeit *uber* ein drittes Attribut besteht:  

- Definition:  
Ein Relationenschema ist in 3. Normalform, wenn fur jede nicht-triviale funktionale Abhangigkeit  $X \rightarrow A$  gilt:
  - $X$  enthalt einen Schlsselkandidaten
  - oder  $A$  ist prim.



## 3. Normalform

Anmerkungen zum Verständnis dieser Definition:

- Wieder Beschränkung auf die FDs, bei denen die rechte Seite nicht-prim ist  
(Abhängigkeiten unter Schlüsselkandidaten sind schwieriger, siehe Boyce-Codd-Normalform S. 44ff)
- Intuitiv möchte die Definition sagen:  
Nicht-prime Attribute sind nur von (ganzen)  
Schlüsselkandidaten funktional abhängig, also:  
Für jede FD gilt: Linke Seite *ist* ein Schlüsselkandidat.
  - Keine Partiellen FDs von Schlüsselkandidaten:  
 $\rightarrow$  2. Normalform ist mit 3. Normalform impliziert
  - Keine FDs, bei denen linke Seite kein Schlüssel ist bzw. Teile enthält, die nicht prim sind.



# 3. Normalform

Anmerkungen zum Verständnis dieser Definition:

- Intuitiv möchte die Definition sagen:  
Nicht-prime Attribute sind nur vom (ganzen)  
Schlüsselkandidaten funktional abhängig, also:  
Für jede FD gilt: Linke Seite *ist* ein Schlüsselkandidat.
- Wegen Reflexivitäts- und Verstärkungsaxiom und  
Allquantor bei den FDs muss man aber berücksichtigen:
  - Es gelten immer auch die trivialen FDs, z.B.  $A \rightarrow A$   
(für jedes beliebige Attribut  $A$ ). Deshalb Ergänzung  
der Definition: „Für jede **nicht-triviale** FD gilt...“
  - Ist  $S$  Schlüssel, dann gilt immer auch  $S^c \rightarrow R$  für jede  
Obermenge  $S^c \supseteq S$ . Deshalb heißt es in der Definition:  
 $X$  **enthält** einen Schlüssel. (statt  $X$  **ist** ein Schlüssel.)



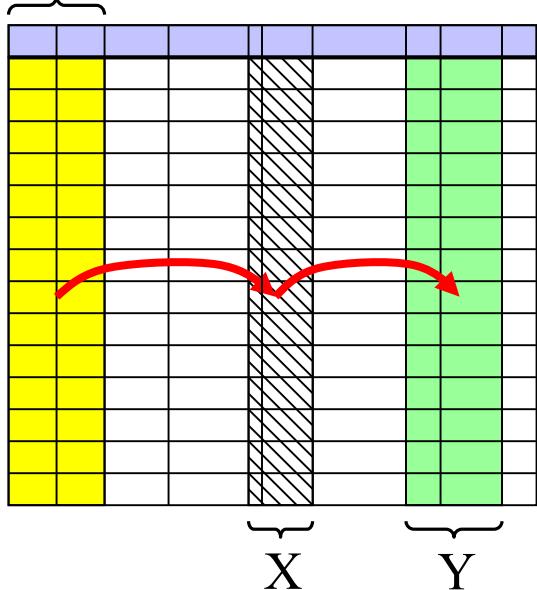
# 3. Normalform

- 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  $X_i \rightarrow Y_i$  von einem Teil eines Schlüssels ( $X_i \subsetneq S$ ) oder von Nicht-Schlüssel-Attribut:
    - Lösche die Attribute  $Y_i$  aus  $R$
    - Gruppiere die Abhängigkeiten nach gleichen linken Seiten  $X_i$
    - Für jede Gruppe führe eine neue Relation ein mit allen enthaltenen Attributen aus  $X_i$  und  $Y_i$
    - $X_i$  wird Schlüssel in der neuen Relation

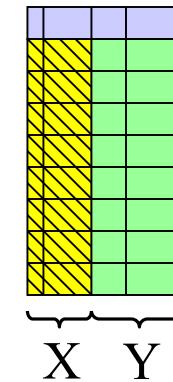
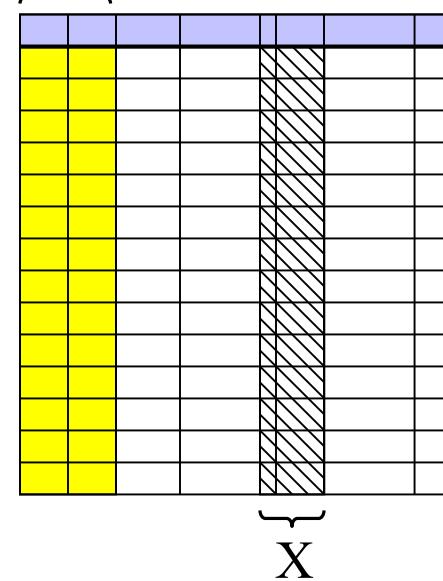


# Grafische Darstellung

Schlüssel



Schlüssel



(nach Heuer, Saake, Sattler)



# Verlustlosigkeit der Zerlegung

- Die Zerlegung einer Relation  $R$  in die beiden Teilrelationen  $R_1$  und  $R_2$  heißt *verlustlos* (oder auch *verbundtreu*), wenn gilt:

$$R = R_1 \bowtie R_2$$

- Man kann zeigen, dass die Zerlegung verlustlos ist, wenn mindestens eine der beiden folgenden FDs gilt:
  - $(R_1 \cap R_2) \rightarrow R_1$
  - $(R_1 \cap R_2) \rightarrow R_2$
- Intuitiv ist klar, dass Verlustlosigkeit von größter Wichtigkeit ist, weil andernfalls gar nicht dieselbe Information in den Relationenschemata  $R_1$  und  $R_2$  gespeichert werden kann, wie in  $R$ .



# Abhängigkeitserhaltung

- Die Zerlegung einer Relation  $R$  in die beiden Teilrelationen  $R_1$  und  $R_2$  heißt *abhängigkeitserhaltend* (oder auch *hüllentreu*), wenn gilt:
$$F_R = (F_{R_1} \cup F_{R_2}) \quad \text{bzw.} \quad F_R^+ = (F_{R_1} \cup F_{R_2})^+$$
- Das heißt, jede funktionale Abhängigkeit soll mindestens einer der Teilrelationen vollständig zugeordnet sein (also alle Attribute der linken Seite und das Attribut der rechten Seite müssen in einer der Teilrelationen enthalten sein).
- Zwar führt die Verletzung der Abhängigkeitserhaltung nicht zu einer Veränderung der speicherbaren Information, aber eine „verlorene“ FD ist nicht mehr überprüfbar, ohne dass der Join durchgeführt wird.  
→ Verschlechterung der Situation (Redundanzen)



# Synthesealgorithmus für 3NF

## Synthesealgorithmus für 3NF

- Der sogenannte *Synthesealgorithmus* ermittelt zu einem gegebenen Relationenschema  $R$  mit funktionalen Abhängigkeiten  $F$  eine Zerlegung in Relationen  $R_1, \dots, R_n$  die folgende Kriterien erfüllt:
  - $R_1, \dots, R_n$  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. Erzeuge neue Relationenschemata aus  $F_c$
3. Rekonstruiere einen Schlüsselkandidaten
4. Eliminiere überflüssige Relationen



# Synthesealgorithmus für 3NF

1. Bestimme die **kanonische Überdeckung**  $F_c$  zu einer gegebenen Menge  $F$  von funktionalen Abhängigkeiten (FDs):
  - a) Führe für jede FD  $X \rightarrow Y \in F$  die Linksreduktion durch, also:
    - Überprüfe für alle  $A \in X$ , ob  $A$  überflüssig ist, d.h. ob
$$Y \subseteq AttrH\"{u}lle(F, X - A)$$
gilt. Falls dies der Fall ist, ersetze  $X \rightarrow Y$  durch  $(X - A) \rightarrow Y$ .
  - b) Führe für jede (verbliebene) FD  $X \rightarrow Y$  die Rechtsreduktion durch, also:
    - Überprüfe für alle  $B \in Y$ , ob
$$B \subseteq AttrH\"{u}lle((F - (X \rightarrow Y)) \cup (X \rightarrow (Y - B)), X)$$
gilt. In diesem Fall ist  $B$  auf der rechten Seite überflüssig und kann eliminiert werden, d.h.  $X \rightarrow Y$  wird durch  $X \rightarrow (Y - B)$  ersetzt.
  - c) Entferne die FDs der Form  $X \rightarrow \{\}$ , die in (b) möglicherweise entstanden sind.
  - d) Fasse FDs der Form  $X \rightarrow Y_1, \dots, X \rightarrow Y_n$  zusammen, so dass  $X \rightarrow (Y_1 \cup \dots \cup Y_n)$  verbleibt.



# Synthesealgorithmus für 3NF

2. Für jede FD  $X \rightarrow Y \in F_c$ 
  - Erzeuge ein Relationenschema  $R_X := X \cup Y$ .
  - Ordne  $R_X$  die FDs  $F_X := \{X' \rightarrow Y' \in F_c \mid X' \cup Y' \in R_X\}$  zu.
3. Rekonstruiere einen Schlüsselkandidaten
  - Falls eines der in Schritt 2 erzeugten Schemata einen Schlüsselkandidaten von  $R$  bzgl.  $F_c$  enthält, sind wir fertig.
  - Andernfalls wähle einen Schlüsselkandidaten  $S \subseteq R$  aus und definiere folgendes zusätzliche Schema:
    - $R_S := S$
    - $F_S := \{\}$
4. Eliminiere überflüssige Relationen
  - Eliminiere diejenigen Schemata  $R_X$ , die in einem anderen Relationenschema  $R_{X'}$  enthalten sind, d.h.
    - $R_X \subseteq R_{X'}$



# Synthesealgorithmus für 3NF

## Beispiel:

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

Es gelten folgende FDs:  $F = \{$

Kunde, Ware  $\rightarrow$  KLand

Kunde, WGruppe  $\rightarrow$  Anbieter

Anbieter  $\rightarrow$  WGruppe

Ware  $\rightarrow$  WGruppe

Kunde  $\rightarrow$  KOrt

KOrt  $\rightarrow$  KLand

}

## Schritte des Synthesealgorithmus:

1. Kanonische Überdeckung  $F_c$  der funktionalen Abhängigkeiten:

a) Linksreduktion:

Kunde, Ware  $\rightarrow$  KLand,

$\{KLand\} \subseteq AttrHülle(F, \{Kunde, Ware\} - \{Ware\})$

weil KLand von Kunde alleine (transitiv) funktional abhängig ist

b) Rechtsreduktion:

Kunde  $\rightarrow$  KLand,

$\{KLand\} \subseteq AttrHülle(F - (Kunde \rightarrow KLand) \cup (Kunde \rightarrow \{\}), Kunde)$

weil KLand transitiv von Kunde funktional abhängig ist

c) Kunde  $\rightarrow \{\}$  wird eliminiert

d) nichts zu tun.



# 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:

|   |  |
|---|--|
| <b>Bezugsquelle</b> ( <u>Kunde</u> , <u>WGruppe</u> , <u>Anbieter</u> ) | {Kunde, WGruppe → Anbieter,<br>Anbieter → WGruppe} |
| <b>Lieferant</b> ( <u>Anbieter</u> , <u>WGruppe</u> )                   | {Anbieter → WGruppe}                               |
| <b>Produkt</b> ( <u>Ware</u> , <u>WGruppe</u> )                         | {Ware → WGruppe}                                   |
| <b>Adresse</b> ( <u>Kunde</u> , <u>KOrt</u> )                           | {Kunde → KOrt}                                     |
| <b>Land</b> ( <u>KOrt</u> , <u>KLand</u> )                              | {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 Abhangigkeiten konnen in der dritten Normalform noch auftreten?  
**Abhangigkeiten unter Attributen, die prim sind, aber noch nicht vollstandig einen Schlssel bilden**
- Beispiel:  
**Autoverzeichnis (Hersteller, HerstellerNr, ModellNr)**
  - es gilt 1:1-Beziehung zw. Hersteller und HerstellerNr:  
 $\text{Hersteller} \rightarrow \text{HerstellerNr}$   
 $\text{HerstellerNr} \rightarrow \text{Hersteller}$
  - Schlsselkandidaten sind deshalb:  
 $\{\text{Hersteller}, \text{ModellNr}\}$   
 $\{\text{HerstellerNr}, \text{ModellNr}\}$
- 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  $X \rightarrow Y$  gilt:
  - $X$  enthält einen Schlüsselkandidaten von  $R$
- Verlustlose Zerlegung ist generell immer möglich.
- Abhängigkeitserhaltende Zerlegung nicht immer möglich.



# Mehrwertige Abhängigkeiten (MVD)

- 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)

|                     |                       |
|---------------------|-----------------------|
| 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)

|             |         |
|-------------|---------|
| Huber, P1,  | Heinz,  |
| Huber, P2,  | Hans,   |
| Huber, P3,  | Hubert, |
| Müller, P2, | Manfred |
| Müller, P3, | NULL    |



# Mehrwertige Abhängigkeiten (MVD)

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

Mitarbeiter (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 (engl. Multivalued Dependency, MVD)



# Mehrwertige Abhängigkeiten (MVD)

Gegeben:  $X, Y \subseteq R$ , und es sei  $Z = R \setminus (X \cup Y)$  d.h. der Rest  $Y$  ist **mehrwertig abhängig** von  $X$  ( $X \rightarrow\!\!\! \rightarrow Y$ ), wenn für jede gültige Ausprägung von  $R$  gilt:

- Für jedes Paar aus Tupeln  $t_1, t_2$  mit  $t_1.X = t_2.X$ , aber  $t_1 \neq t_2$ , existieren die (nicht immer zu  $t_1$  und  $t_2$  verschiedenen) Tupel  $t_3$  und  $t_4$  mit den Eigenschaften:

$$\begin{aligned}t_1.X &= t_2.X = t_3.X = t_4.X \\t_3.Y &= t_1.Y \\t_3.Z &= t_2.Z \\t_4.Y &= t_2.Y \\t_4.Z &= t_1.Z\end{aligned}$$

D.h. jedem  $X$  ist eine **Menge von  $Y$ -Werten** zugeordnet.

Jede FD ist auch MVD

(diese Menge ist dann jeweils ein-elementig und  $t_3=t_2, t_4=t_1$ )



# Beispiel MVD

| $R$   |                                    |  |  |
|-------|------------------------------------|--|--|
|       | $X$<br>$\overbrace{A_1 \dots A_i}$ | $Y$<br>$\overbrace{A_{i+1} \dots A_j}$ | $Z$<br>$\overbrace{A_{j+1} \dots A_n}$ |
| $t_1$ | $a_1 \dots a_i$                    | $a_{i+1} \dots a_j$                    | $a_{j+1} \dots a_n$                    |
| $t_2$ | $a_1 \dots a_i$                    | $b_{i+1} \dots b_j$                    | $b_{j+1} \dots b_n$                    |
| $t_3$ | $a_1 \dots a_i$                    | $a_{i+1} \dots a_j$                    | $b_{j+1} \dots b_n$                    |
| $t_4$ | $a_1 \dots a_i$                    | $b_{i+1} \dots b_j$                    | $a_{j+1} \dots a_n$                    |



# Weiteres Beispiel

Relation: Modelle

| ModellNr | Farbe   | Leistung |
|----------|---------|----------|
| E36      | blau    | 170 PS   |
| E36      | schwarz | 198 PS   |
| E36      | blau    | 198 PS   |
| E36      | schwarz | 170 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  $X \rightarrow\rightarrow Y$  bezogen auf  $R \supseteq X \cup Y$  ist *trivial*, wenn jede mögliche Ausprägung  $r$  von  $R$  diese MVD erfüllt. Man kann zeigen, dass  $X \rightarrow\rightarrow Y$  trivial ist, genau dann wenn:

- $Y \subseteq X$  oder
- $Y = R - X$ .

Eine Relation  $R$  mit zugeordneter Menge  $F$  von funktionalen und mehrwertigen Abhängigkeiten ist in 4. Normalform (**4NF**), wenn für jede MVD  $X \rightarrow\rightarrow Y \in F^+$  eine der folgenden Bedingungen gilt:

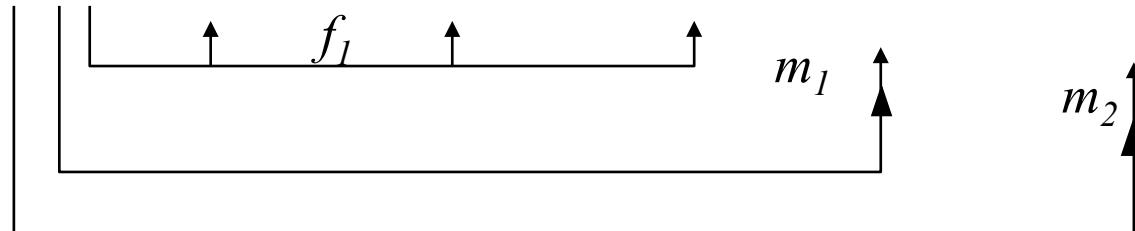
- Die MVD ist trivial oder
- $X$  ist ein Superschlüssel von  $R$ .



# Beispiel

Assistenten:

$\{[\text{PersNr}, \text{Name}, \text{Fachgebiet}, \text{Boss}, \text{Sprache}, \text{ProgSprache}]\}$



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



# 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:  
Synthesealgorismus 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 Abhangigkeit eines Nicht-Schlssel-Attributs von **Teil** eines Schlssels
- 3. Normalform:  
Zusätzlich keine nichttriviale funktionale Abhangigkeit eines Nicht-Schlssel-Attributs von Nicht-Schlssel-Attributen
- Boyce-Codd-Normalform:  
Zusätzlich keine nichttriviale funktionale Abhangigkeit unter den Schlssel-Attributen
- 4. Normalform:  
keine Redundanz durch MVDs.



Skript zur Vorlesung  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 8: Physische Datenorganisation

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani

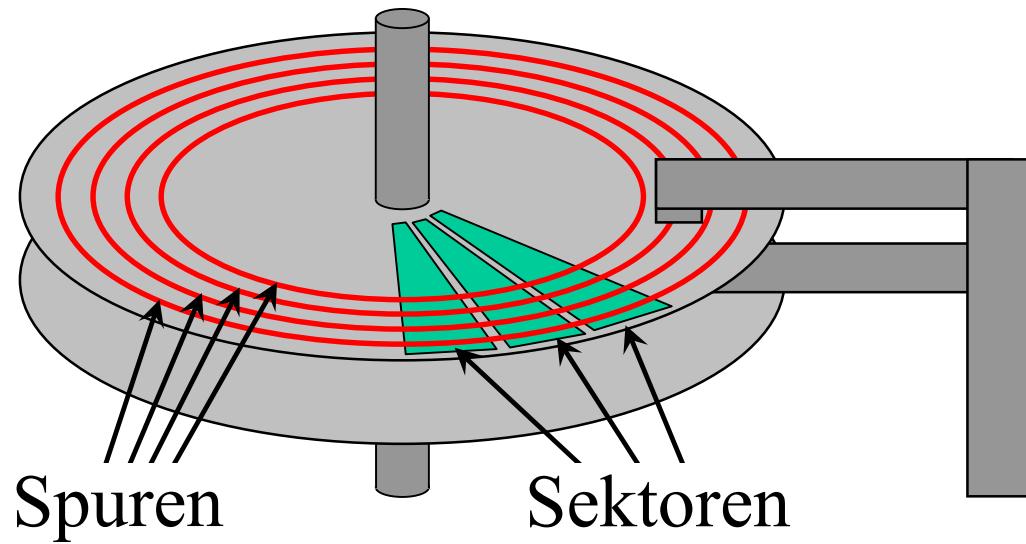
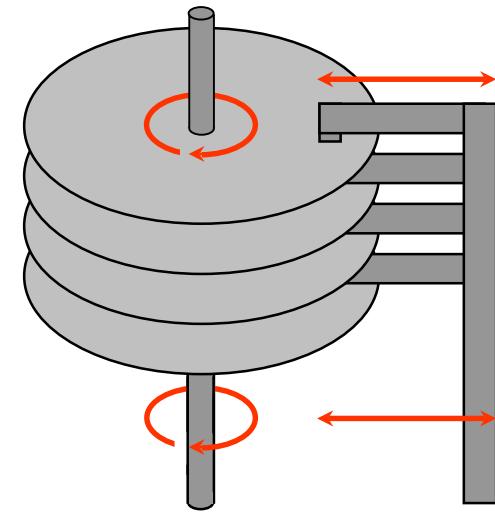
Skript © 2019 Christian Böhm



# Wiederholung (1)

## 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 (2)

## Intensionale Ebene vs. Extensionale Ebene

- Datenbankschema:

Name (10 Zeichen)

|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|

Vorname (8 Z.)

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

Jahr (4 Z.)

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|--|--|--|--|

- Ausprägung der Datenbank:

|   |   |   |   |   |   |   |   |  |  |
|---|---|---|---|---|---|---|---|--|--|
| 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 |  |  |  |  |
|---|---|---|---|---|---|--|--|--|--|

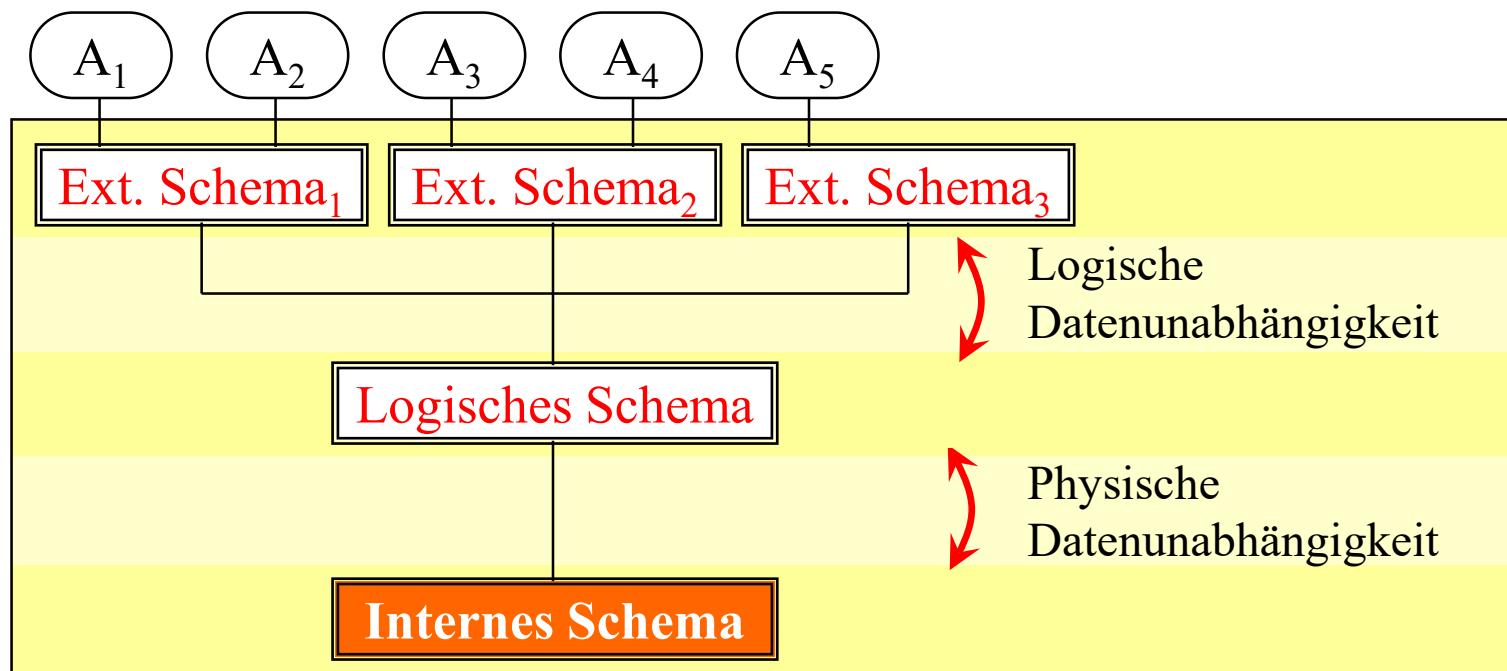
|   |   |   |   |
|---|---|---|---|
| 1 | 9 | 4 | 9 |
|---|---|---|---|

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



# Wiederholung (3)

Drei-Ebenen-Architektur zur Realisierung von  
– **physischer**  
– **und logischer**  
Datenunabhängigkeit (nach ANSI/SPARC)





# Wiederholung (4)

- 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**

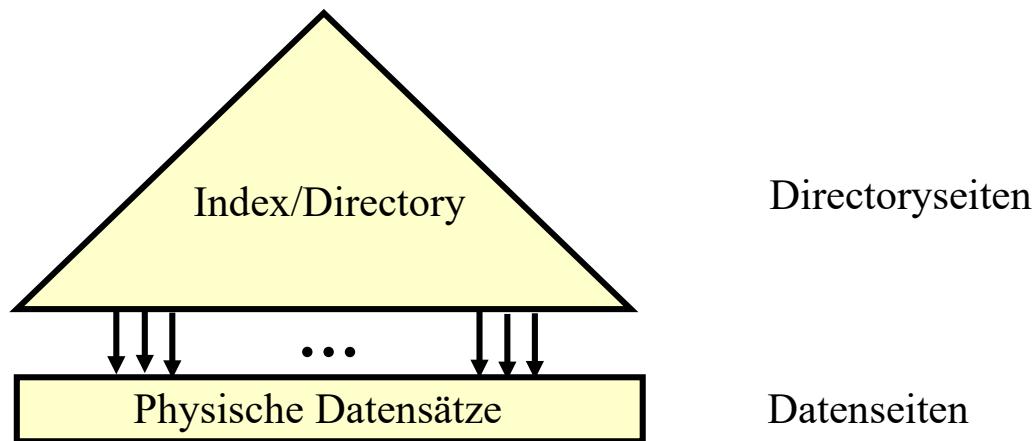
Strukturinformation zur Zuordnung von Suchschlüsseln und zur Organisation der Datei.

- **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 von Datensätzen wird 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:

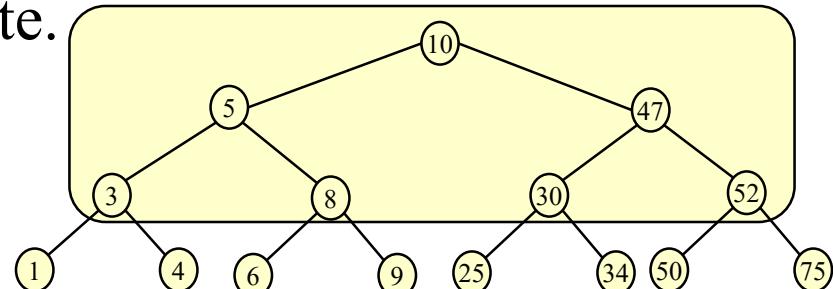
- **Primärschlüsselsuche** (B-Baum und lineares Hashing)
- **Sekundärschlüsselsuche** (invertierte Listen)



# B-Baum (1)

Idee:

- Daten auf der Festplatte sind in Blöcken organisiert (z.B. 4 Kb Blöcke)
- Bei Organisation der Schlüssel mit einem binären 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  $m$**

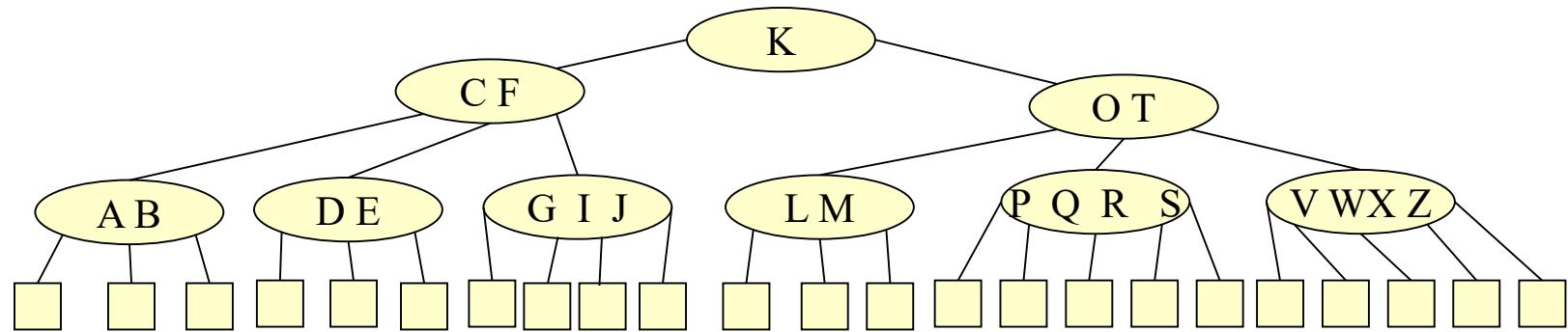
(Bayer und McCreight (1972))

- (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  $k+1$  Kinder.
- (5) Alle Blätter befinden sich auf demselben Level.



# B-Baum (3)

## Beispiel: B-Baum der Ordnung 2



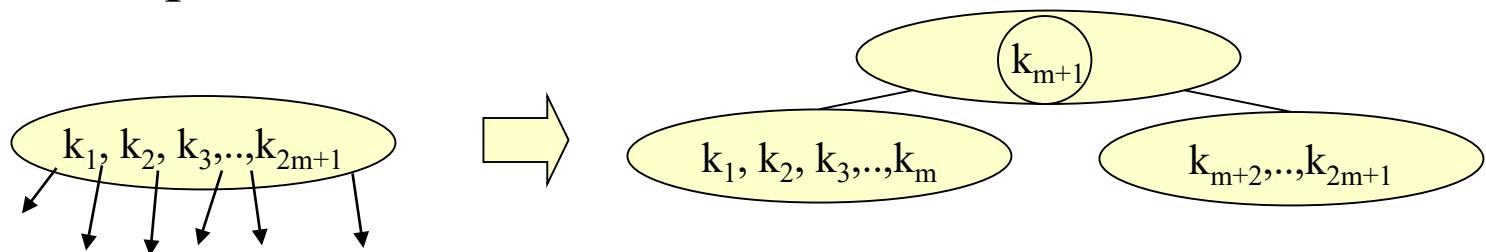
- max Höhe:  $h \leq \left\lfloor \log_{m+1} \left( \frac{n+1}{2} \right) \right\rfloor + 1$
- Ordnung in realen B-Bäumen: 600-900 Schlüssel pro Seite
- Effiziente Suche innerhalb Knoten ? => **binäre Suche**



# 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 weniger als  $2m$  Schlüssel  
=> füge  $k$  in  $B$  ein
- 2. Fall:  $B$  enthält  $2m$  Schlüssel  
=> Overflow Behandlung
  - Split des Blattknotens



- Split kann sich über mehrere Ebene fortsetzen bis zur Wurzel



# Entfernen aus B-Baum

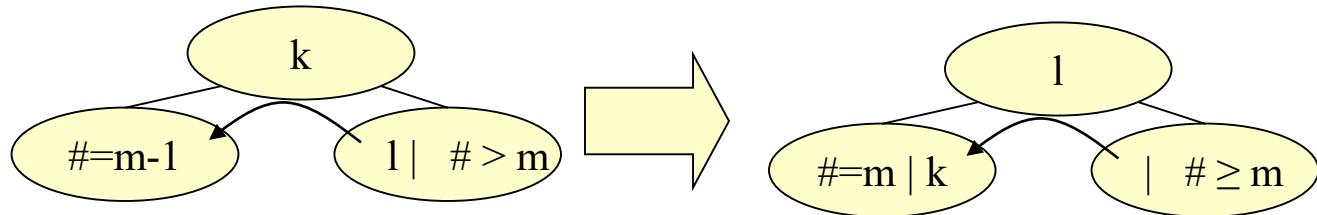
Lösche Schlüssel  $k$  aus Baum:

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

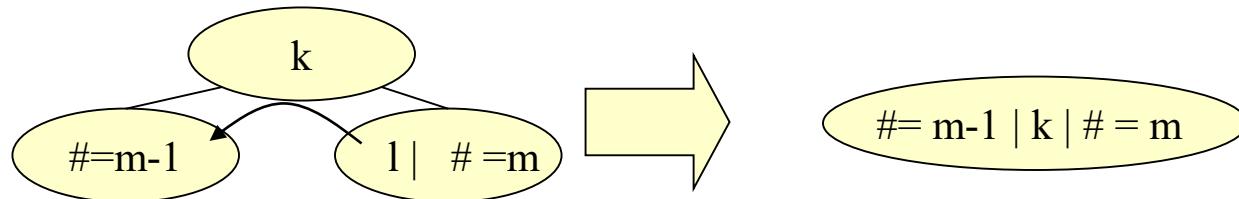


# Underflow-Behandlung im B-Baum

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



- 2. Fall: Nachbar hat genau m Knoten => Verschmelzen der Nachbarn



- 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 Studierenden, 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 rechten Teilbaum 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  $s(u)$ .

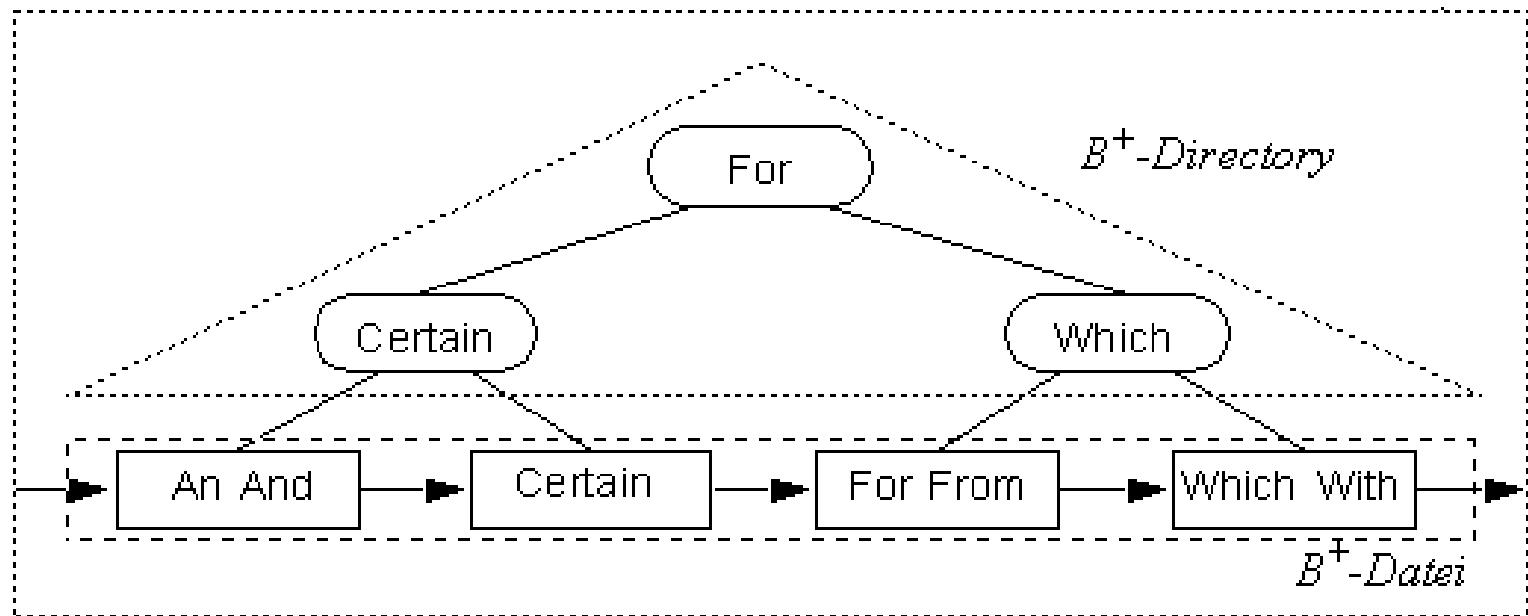


# B+-Baum (3)

## Beispiel:

B+-Baum für die Zeichenketten:

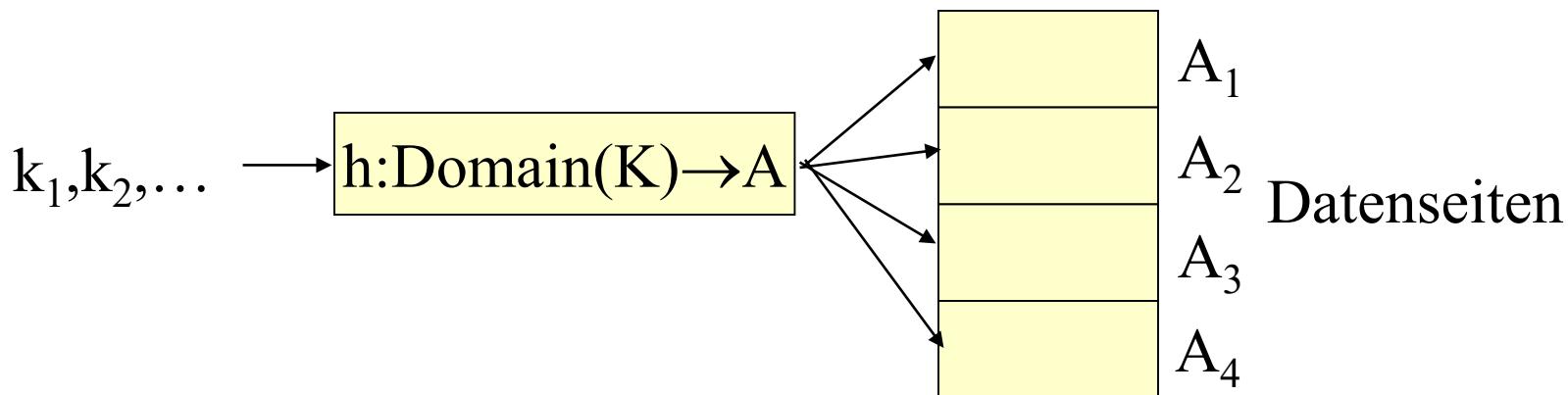
An, And, Certain, For, From, Which, With





# 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
  - $|Domain(K)| \gg |A| \Rightarrow$  Kollision





# Hash-Verfahren für Sekundärspeicher

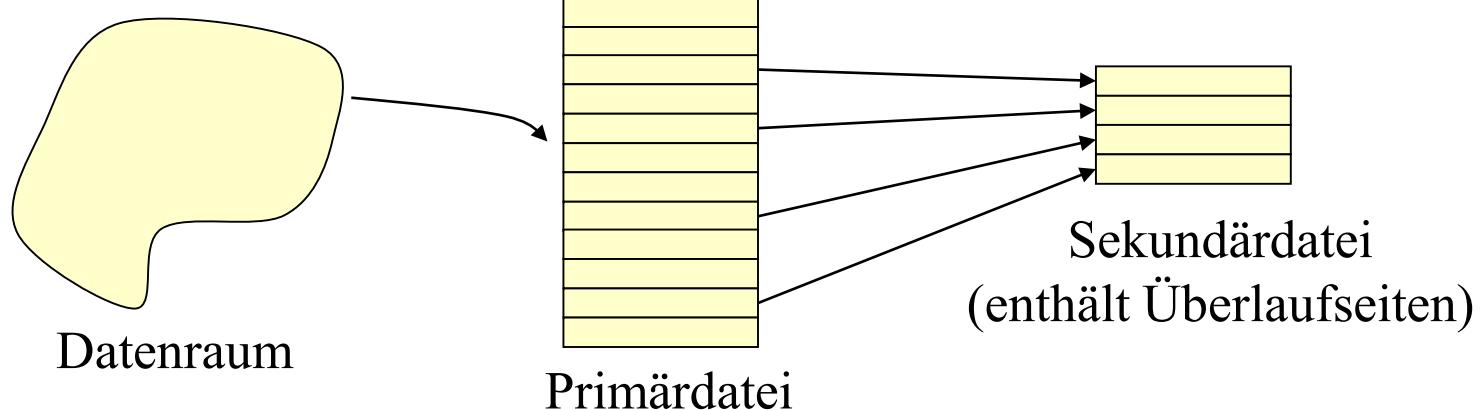
Für Sekundärspeicher sind weitere Anforderungen von Bedeutung:

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

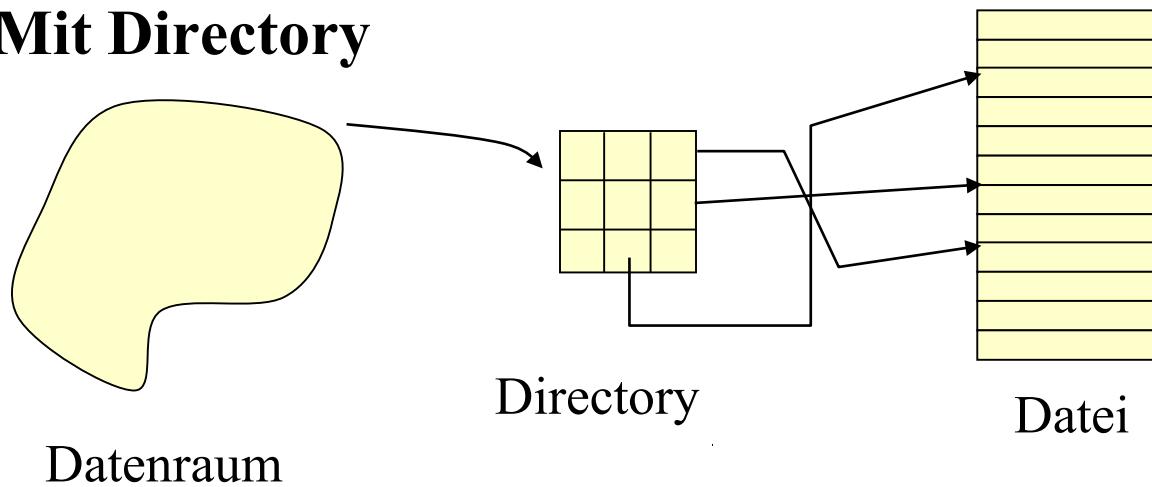


# Klassifizierung von Hash-Verfahren

- **Ohne Directory**



- **Mit Directory**

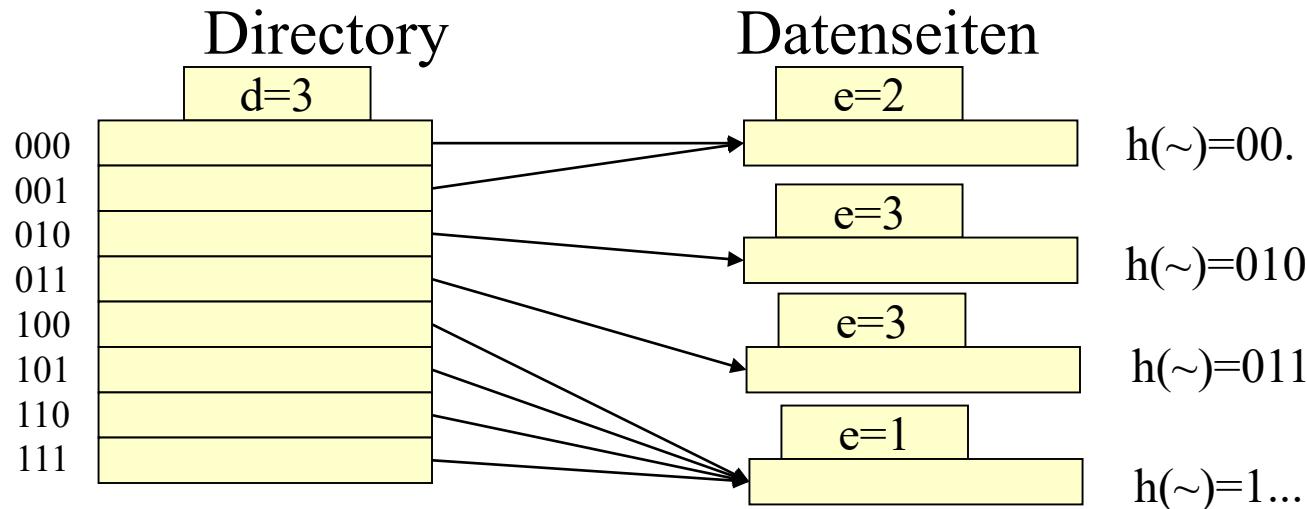




# Hash-Verfahren mit Directory

## Erweiterbares Hashing

- Hashfunktion:  $h(k)$  liefert Bitfolge  $(b_1, b_2, \dots, b_d, \dots)$
- 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  
 $h(k) = (b_1, b_2, \dots, b_d, \dots)$
2. Schritt:  
Der Directoryeintrag  $D[b_1, b_2, \dots, b_d]$  liefert Seitennummer.  
Datensatz wird in berechnete Seite eingefügt.

Falls Seite danach max. gefüllt:

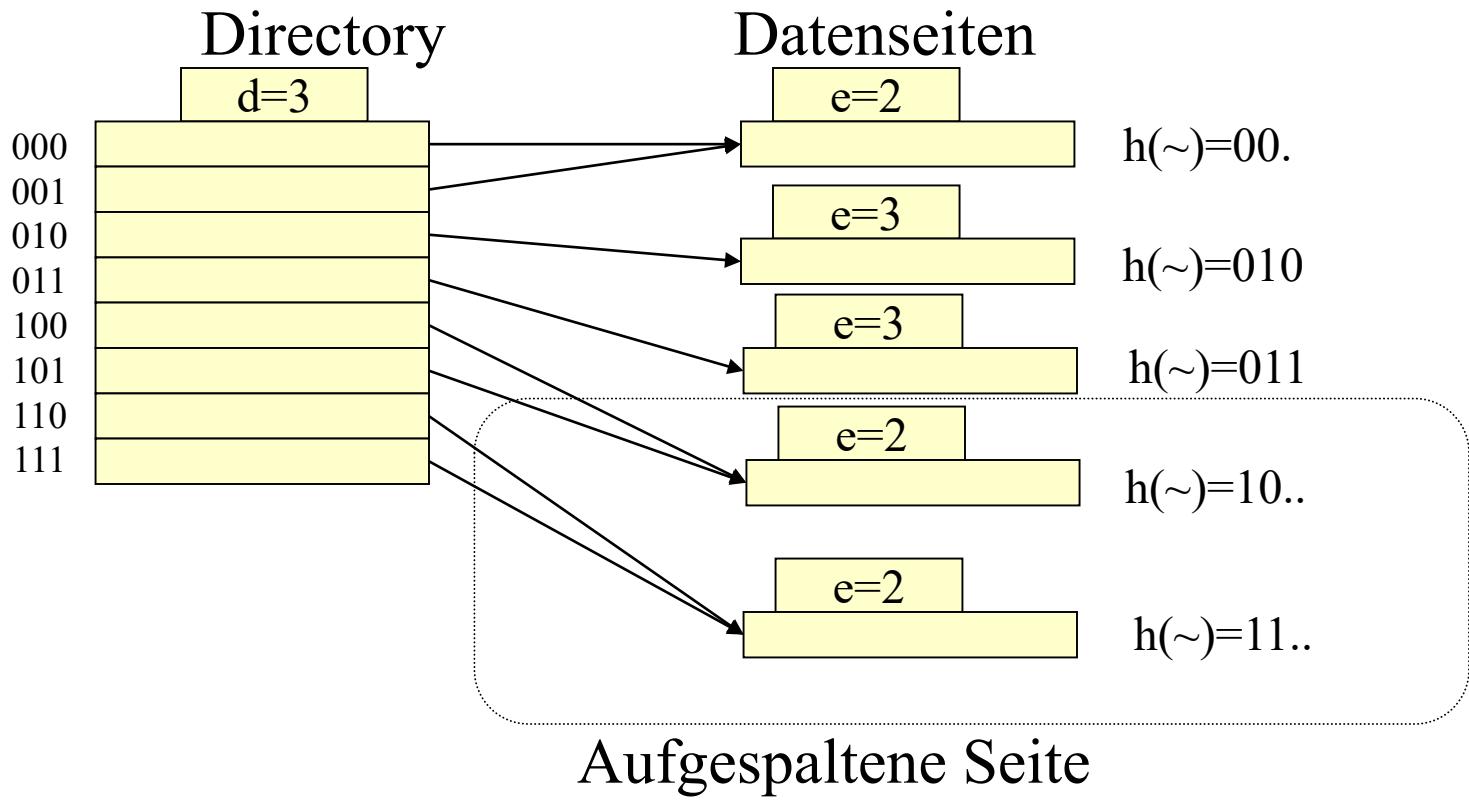
1. Aufspalten der Datenseite.
2. Verdoppeln des Directory.



# Einfügen Erweiterbares Hashing (2)

Aufspalten einer Datenseite

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

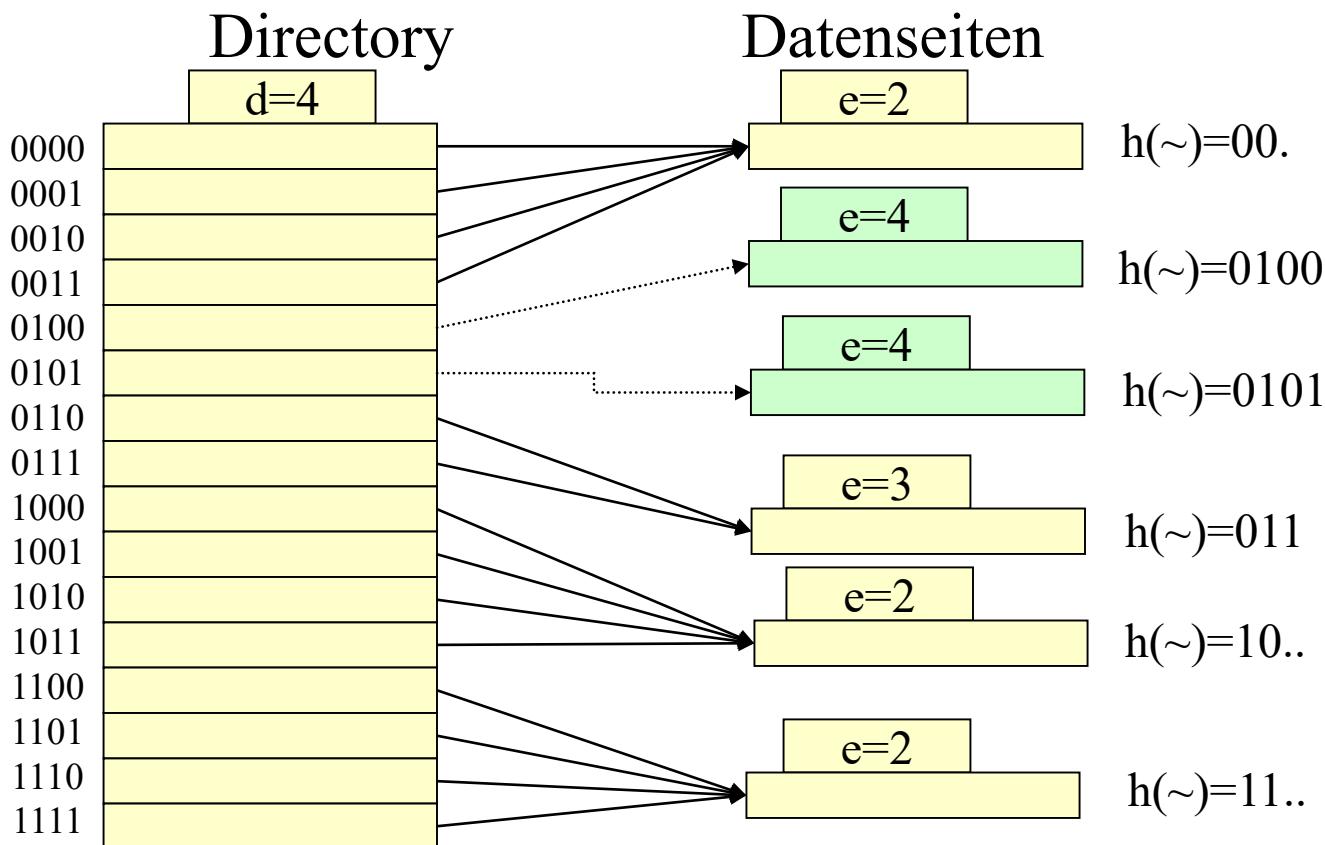




# Einfügen Erweiterbares Hashing (3)

Verdopplung des Directory

Datenseite läuft über und  $d = e$ .

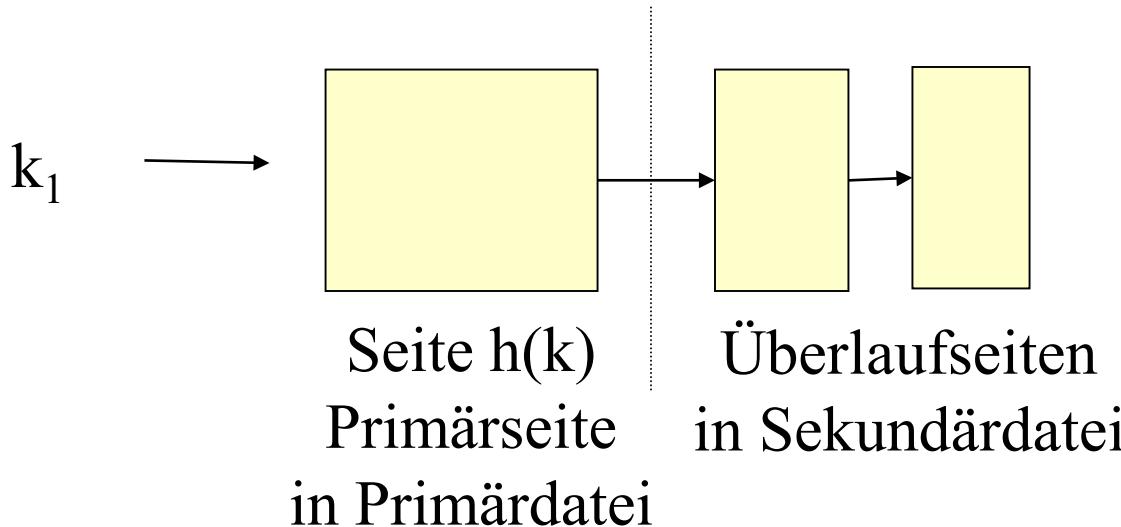




# Hashing ohne Directory

## Lineares Hashing

- Hash-Funktion  $h: K \rightarrow A$  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)

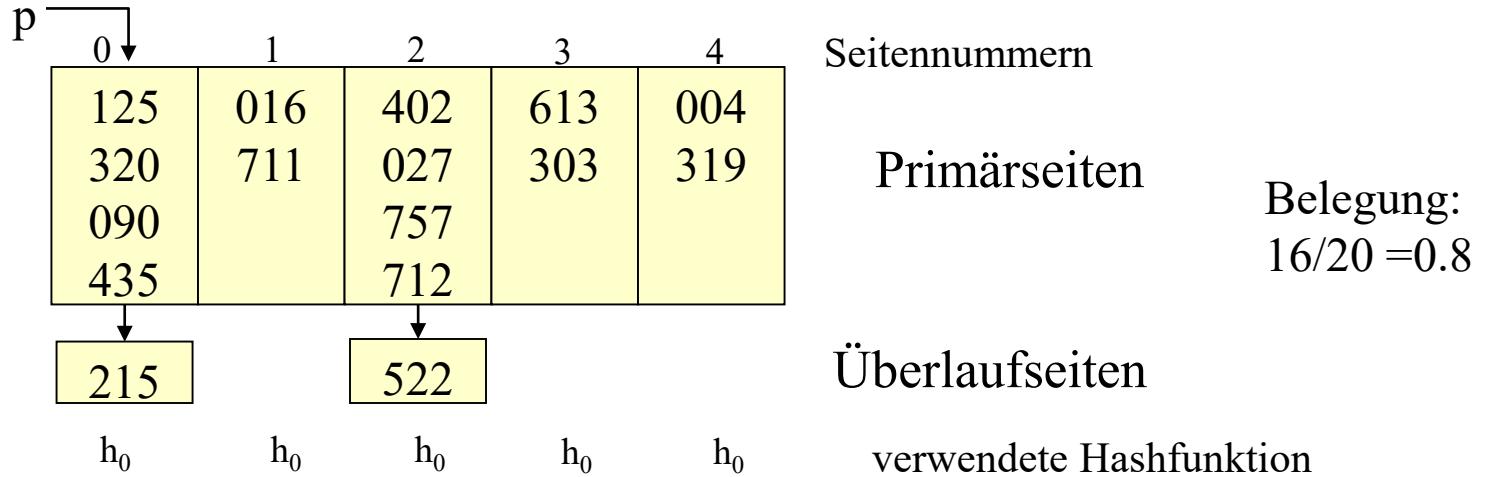
- dynamisches Wachstum der Primärdatei
- Folge von Hash-Funktionen:  $h_0, h_1, h_2, \dots$
- Erweitern der Primärdatei um jeweils eine Seite
- feste Splitreihenfolge
- Expansionzeiger zeigt an welche Seite gesplittet wird
- Kontrollfunktion: Wann wird gesplittet ?  
Belegungsfaktor übersteigt Schwellwert: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

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

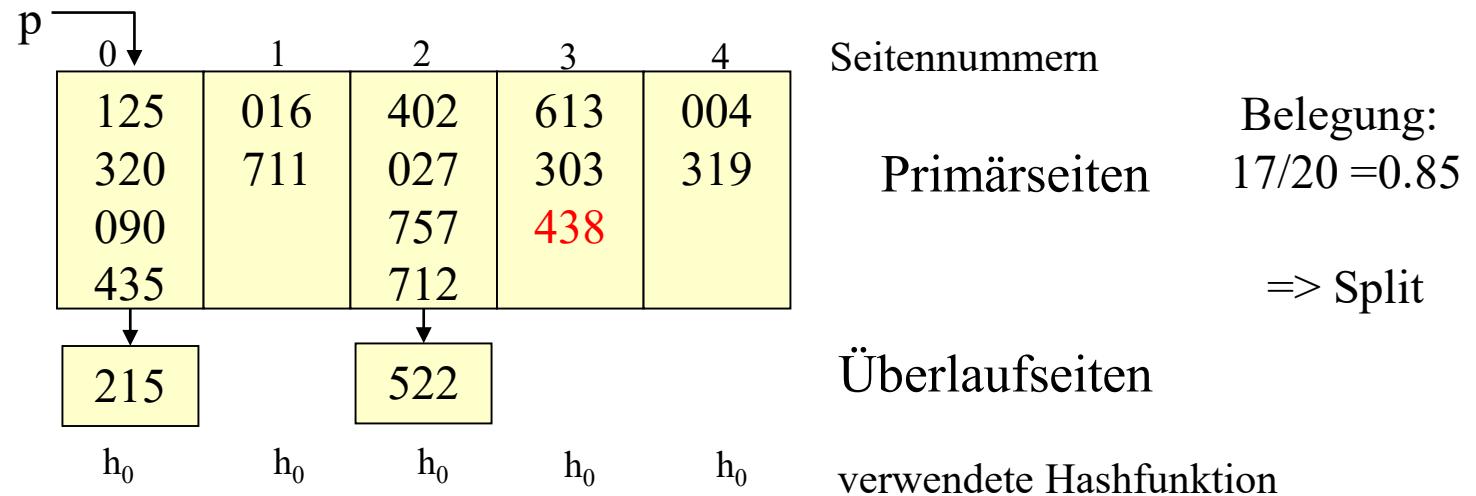


# Lineares Hashing (2)

- Hashfunktionen:  $h_0(k) = k \bmod 5$ ,  $h_1(k) = k \bmod 10$ , ...



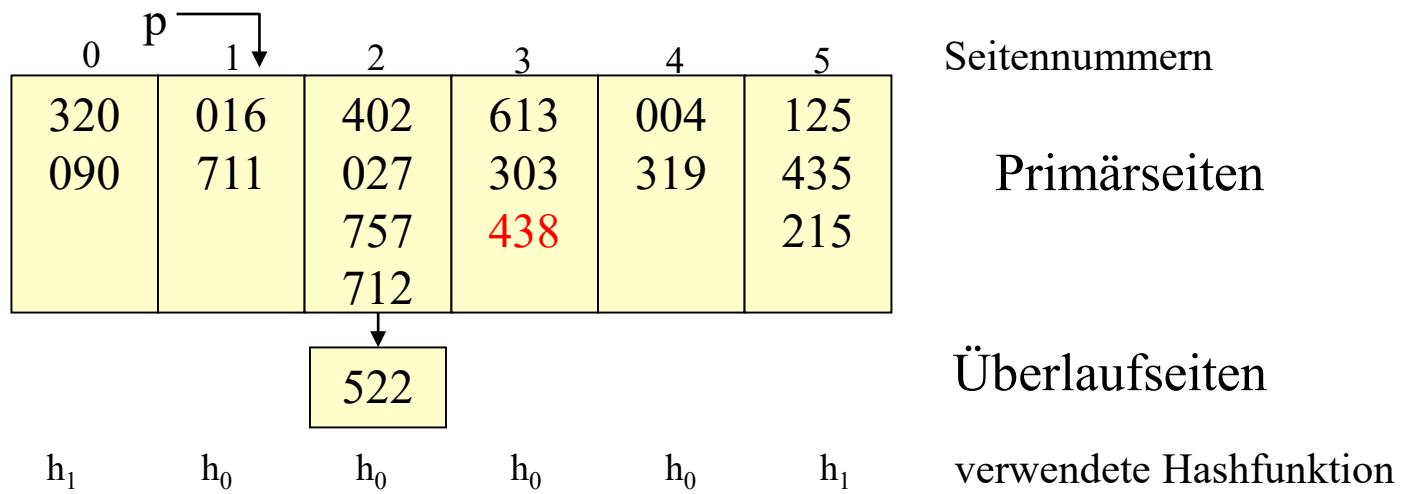
Einfügen von Schlüssel 438:





# Lineares Hashing (3)

Expansion der Seite 0 auf die Seiten 0 und 5:



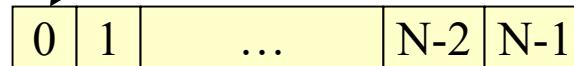
- Umspeichern aller Datensätze mit  $h_1(k) = 5$  in neue Seite
- Datensätze mit  $h_1(k) = 0$  bleiben



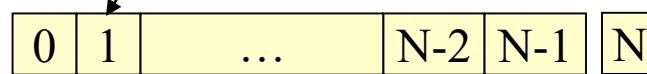
# Lineares Hashing (4)

Prinzip der Expansion:

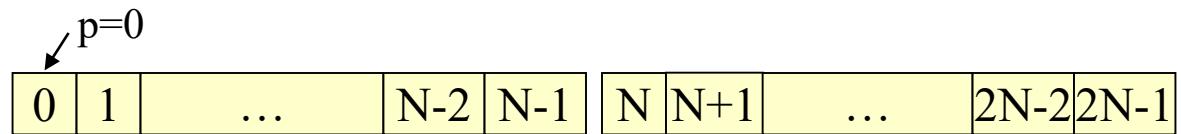
Ausgangssituation  $p=0$



nach dem ersten Split  $p=1$



nach Verdopplung der Datei



- 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  $\{h_i\}$ ,  $i \geq 0$ :

1.) Bereichsbedingung:

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

2.) Splitbedingung:

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

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

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

Beispiel:

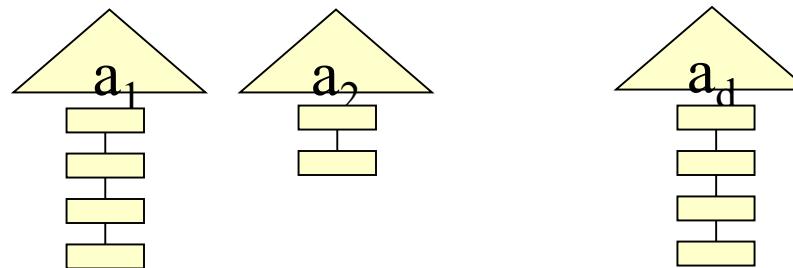
$$h(k) = k \bmod (2^L * N)$$



# Anfragen auf mehreren Attributen (1)

## Invertierte Listen (häufigste Lösung)

- jedes relevante Attribute wird mit eindimensionalem Index verwaltet.
- Suche nach mehreren Attributen  $a_1, a_2, \dots, a_d$
- 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<sub>1</sub>*, *a<sub>2</sub>*, ..., *a<sub>n</sub>*);**

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:

$t_1 < t_2 \text{ gdw.}$

$t_1.a_1 < t_2.a_1 \text{ oder } (t_1.a_1 = t_2.a_1 \text{ und } t_1.a_2 < t_2.a_2) \text{ oder ...}$

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  $a_1 = \dots \text{ AND } \dots \text{ AND } a_n = \dots$**

- **Partial match query:**

**SELECT \* FROM  $t$  WHERE  $a_1 = \dots \text{ AND } \dots \text{ AND } a_i = \dots$**

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  $a_1 = \dots \text{ AND } \dots \text{ AND } a_i = \dots \text{ AND } a_{i+1} \leq \dots$**

auch z.B. für ‘>’ oder ‘BETWEEN’

- **Pointset query:**

**SELECT \* FROM  $t$  WHERE  $a_1 = \dots \text{ AND } \dots \text{ AND } a_i = \dots \text{ AND } a_{i+1} \in \{7, 17, 77\}$**

auch z.B. ( $a_{i+1} = \dots \text{ OR } a_{i+1} = \dots \text{ OR } \dots$ )



# Durch Index unterstützte Anfragen

- Pattern matching query

**SELECT \* FROM  $t$  WHERE  $a_1=...$  AND ... AND  $a_i=...$  AND  $a_{i+1}$  LIKE ‘ $c_1c_2\dots c_k\%$ ’**

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.



# Indexstrukturen in SQL

Index Reihenfolge:

- 1.) **create index name\_geb on Dozent**  
(Name, Geburt)
- 2.) **create index geb\_name on Dozent**  
(Geburt, Name )

Welcher Index unterstützt folgende Anfrage besser:

Select Name from Dozent where Name like 'H%'

Index „name\_geb“ ist besser, da Name ein Praefix von Name, Geburt.



# 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**  
Wintersemester 2023/2024

# Kapitel 9

# Relationale

# Anfragebearbeitung

Vorlesung: Prof. Dr. Thomas Seidl

Übungen: Collin Leiber, Walid Durani

© Christian Böhm 2020

Dieses Skript basiert in Teilen auf den Skripten zur Vorlesung Datenbanksysteme II an der LMU München von

Prof. Dr. Christian Böhm (SoSe 2007),

Prof. Dr. Peer Kröger (SoSe 2008, 2013, 2014, 2015) und

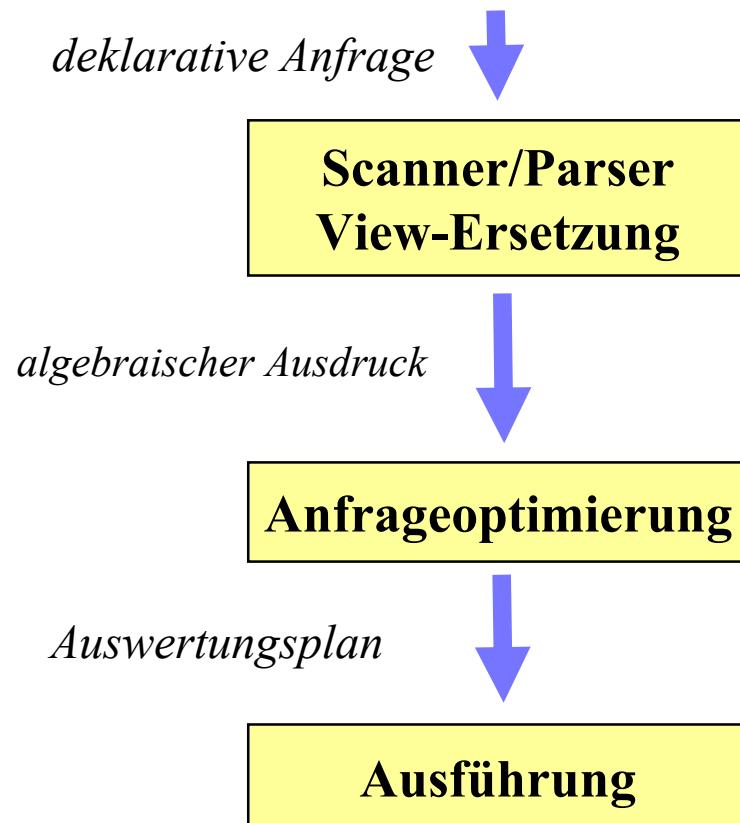
Prof. Dr. Matthias Schubert (SoSe 2009)





## Aufgabe der Anfragebearbeitung

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

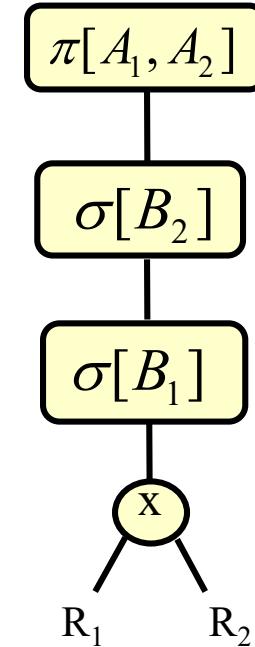


## Kanonischer Auswertungsplan

*SELECT A<sub>1</sub>, A<sub>2</sub>*

*FROM R<sub>1</sub>, R<sub>2</sub>*

*WHERE B<sub>1</sub> AND B<sub>2</sub>*

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


1. Bilde das kartesische Produkt der Relationen R<sub>1</sub>, R<sub>2</sub>
2. Führe Selektionen mit den Bedingungen B<sub>1</sub>, B<sub>2</sub> durch.
3. Projiziere die Ergebnis-Tupel auf die erforderlichen Attribute A<sub>1</sub>, A<sub>2</sub>



## Logische vs. physische Anfrageoptimierung

- Optimierungstechniken, die den Auswertungsplan umbauen (d.h. die Reihenfolge der Operatoren verändern), werden als **logische Anfrageoptimierung** bezeichnet.
- Unter **physischer Anfrageoptimierung** versteht man z.B. die Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation oder die Entscheidung, ob für eine Selektionsoperation ein Index verwendet wird oder nicht und wenn ja, welcher (bei unterschiedlichen Alternativen). Es handelt sich also um die Auswahl eines geeigneten Algorithmus für jede Operation im Auswertungsplan.



## Regel- vs. kostenbasierte Anfrageoptimierung

- Es gibt zahlreiche Regeln (Heuristiken), um die Reihenfolge der Operatoren im Auswertungsplan zu modifizieren und so eine Performanz-Verbesserung zu erreichen, z.B.:
  - Push Selection: Führe Selektionen möglichst frühzeitig (vor Joins) aus
  - Elimination leerer Teilbäume
  - Erkennen gemeinsamer Teilbäume
- Optimierer, die sich ausschließlich nach diesen starren Regeln richten, nennt man **regelbasierte** oder auch **algebraische Optimierer**.



## Regel-/kostenbasierte Optimierung (cont.)

- Optimierer, die zusätzlich zu den starren Regeln die voraussichtliche Performanz der Auswertungspläne ermitteln und den leistungsfähigsten Plan auswählen, werden als **kostenbasierte** oder auch (irreführend) **nicht-algebraische Optimierer** bezeichnet.
- Die Vorgehensweise kostenbasierter Anfrageoptimierer ist meist folgende:
  - Generiere einen initialen Plan (z.B. Standardauswertungsplan)
  - Schätze die bei der Auswertung entstehenden Kosten
  - Modifiziere den aktuellen Plan gemäß vorgegebener Heuristiken
  - Wiederhole die Schritte 2 und 3 bis ein Stop-Kriterium erreicht ist
  - Gib den besten erhaltenen Plan aus



# Logische Anfrageoptimierung

## Äquivalenzregeln der Relationalen Algebra

- Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ

$$\begin{array}{lll} R \bowtie S & = & S \bowtie R \\ R \cup S & = & S \cup R \\ R \cap S & = & S \cap R \\ R \times S & = & S \times R \end{array}$$

- Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ

$$\begin{array}{lll} R \bowtie (S \bowtie T) & = & (R \bowtie S) \bowtie T \\ R \cup (S \cup T) & = & (R \cup S) \cup T \\ R \cap (S \cap T) & = & (R \cap S) \cap T \\ R \times (S \times T) & = & (R \times S) \times T \end{array}$$

- Selektionen sind untereinander vertauschbar

$$\sigma_{Bed_1}(\sigma_{Bed_2}(R)) = \sigma_{Bed_2}(\sigma_{Bed_1}(R))$$



# Logische Anfrageoptimierung

## Äquivalenzregeln der Relationalen Algebra (cont.)

- Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen werden, bzw. nacheinander ausgeführte Selektionen können zu einer konjunktiven Selektion zusammengefasst werden

$$\sigma_{B_1 \wedge B_2 \wedge \dots \wedge B_n}(R) = \sigma_{B_1}(\sigma_{B_2}(\dots(\sigma_{B_n}(R))\dots))$$

- Geschachtelte Projektionen können eliminiert werden

$$\pi_{A1}(\pi_{A2}(\dots(\pi_{An}(R))\dots)) = \pi_{A1}(R)$$

Damit eine solche Schachtelung sinnvoll ist, muss gelten:  $A1 \subseteq A2 \subseteq \dots \subseteq An$

- Selektion und Projektion sind vertauschbar, falls die Projektion keine Attribute der Selektionsbedingung entfernt

$$\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R))$$

, falls  $attr(B) \subseteq A$



# Logische Anfrageoptimierung

## Äquivalenzregeln der Relationalen Algebra (cont.)

- Selektion und Join (Kreuzprodukt) können vertauscht werden, falls die Selektion nur Attribute eines der beiden Join-Argumente verwendet

$$\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$$

, falls  $attr(B) \subseteq attr(R)$

$$\sigma_B(R \times S) = \sigma_B(R) \times S$$

- Projektionen können teilweise in den Join verschoben werden

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A1}(R) \bowtie_B \pi_{A2}(S))$$

, falls  $A1 = attr(R) \cap (A \cup attr(B))$   
 $A2 = attr(S) \cap (A \cup attr(B))$

- Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden

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



## Äquivalenzregeln der Relationalen Algebra (cont.)

- Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden (siehe Übung!)

$$\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$$

- Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist (z.B. Equi-Join)
- Auch an Bedingungen können Veränderungen vorgenommen werden

- Kommutativgesetze, Assoziativgesetze, z.B.  $B_1 \wedge B_2 = B_2 \wedge B_1$

- Distributivgesetze, z.B.  $B_1 \vee (B_2 \wedge B_3) = (B_1 \vee B_2) \wedge (B_1 \vee B_3)$

- De Morgan, z.B.  $\neg(B_1 \wedge B_2) = \neg B_1 \vee \neg B_2$



## Restrukturierungsalgorithmus

- Aufbrechen der Selektionen
- Verschieben der Selektionen so weit wie möglich nach unten im Operatorbaum
- Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
- Einfügen und Verschieben von Projektionen so weit wie möglich nach unten
- Zusammenfassen einzelner Selektionen zu komplexen Selektionen



# Logische Anfrageoptimierung

## Beispiel

Fahrzeug-Datenbank

Kunde(KNr, Name, Adresse, Region, Saldo)

| KNr | Name   | Adresse    | Region       | Saldo   |
|-----|--------|------------|--------------|---------|
| 201 | Klein  | Lilienthal | Bremen       | 200 000 |
| 337 | Horn   | Dieburg    | Rhein-Main   | 100 000 |
| 444 | Berger | München    | München      | 300 000 |
| 108 | Weiss  | Würzburg   | Unterfranken | 500 000 |

Bestellt(BNr, Datum, KNr, KNr, PNr)

| BNr | Datum    | KNr | PNr |
|-----|----------|-----|-----|
| 221 | 10.05.04 | 201 | 12  |
| 312 | 11.05.04 | 201 | 4   |
| 401 | 20.05.04 | 337 | 330 |
| 456 | 13.05.04 | 444 | 330 |
| 458 | 14.05.04 | 444 | 98  |

Produkt(PNr, Bezeichnung,  
Anzahl, Preis)

| PNr | Bezeichnung | Anzahl | Preis   |
|-----|-------------|--------|---------|
| 12  | BMW 318i    | 10     | 40.000  |
| 4   | Golf 5      | 40     | 25.000  |
| 330 | Fiat Uno    | 5      | 18.000  |
| 98  | Ferrari 380 | 1      | 180.000 |
| 14  | Opel Corsa  | 14     | 17.000  |

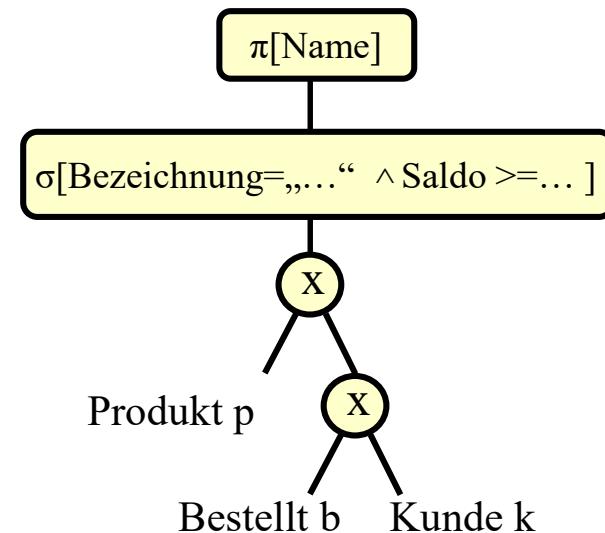
# Logische Anfrageoptimierung

## Beispiel (cont.)

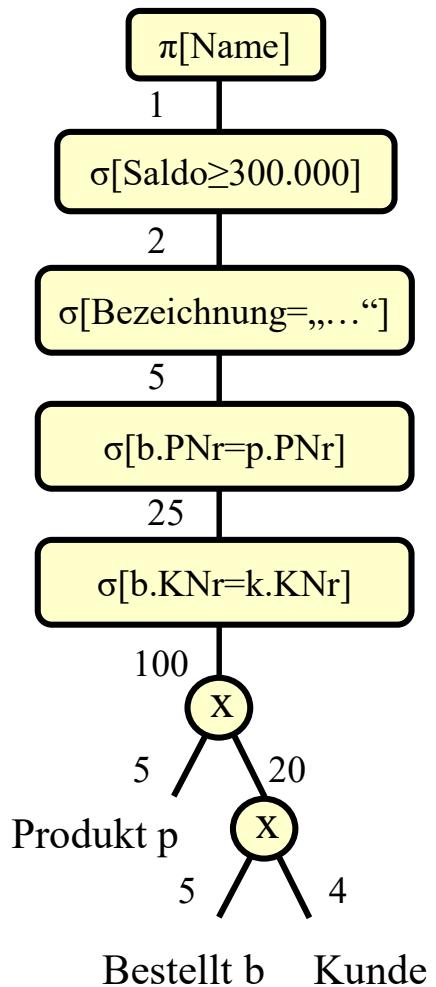
- SQL Anfrage:

```
select           Name,  
from            Kunde k, Bestellt b, Produkt p  
where           b.KNr = k.KNr  
and             b.PNr = p.PNr  
and             Bezeichnung = „Fiat Uno“  
and             Saldo ≥ 300.000
```

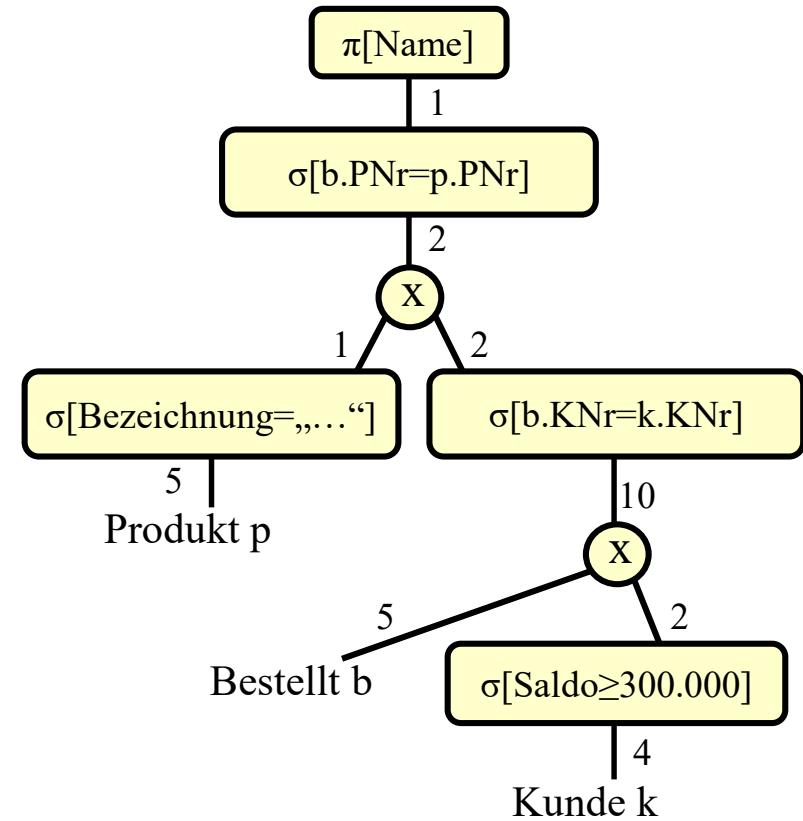
- Kanonischer Auswertungsplan:



## - Aufbrechen der Selektionen

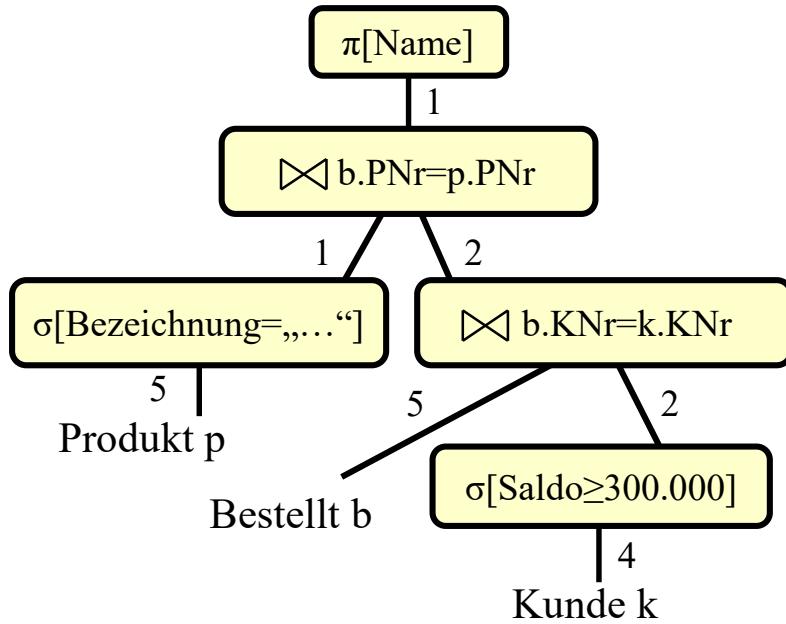


## - Verschieben der Selektionen

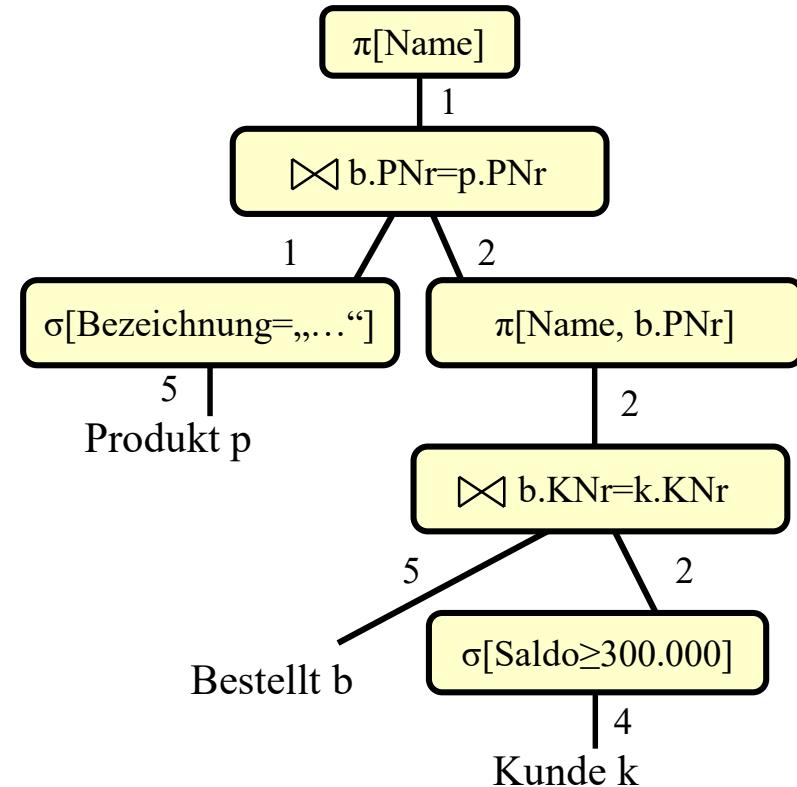


# Logische Anfrageoptimierung

## - Zusammenfassen zu Joins



## - Einfügen zusätzlicher Projektionen





## Selektivität

- Der Anteil der qualifizierenden Tupel wird *Selektivität sel* genannt.
- Für die Selektion und den Join ist sie folgendermaßen definiert:

- **Selektion** mit Bedingung  $B$ :

$$sel_B = \frac{|\sigma_B(R)|}{|R|}$$

(relativer Anteil der Tupel, die  $B$  erfüllen)

- **Join** von  $R$  und  $S$ :

$$sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

(Anteil relativ zur Kardinalität des Kreuzprodukts)



## Selektivität (cont.)

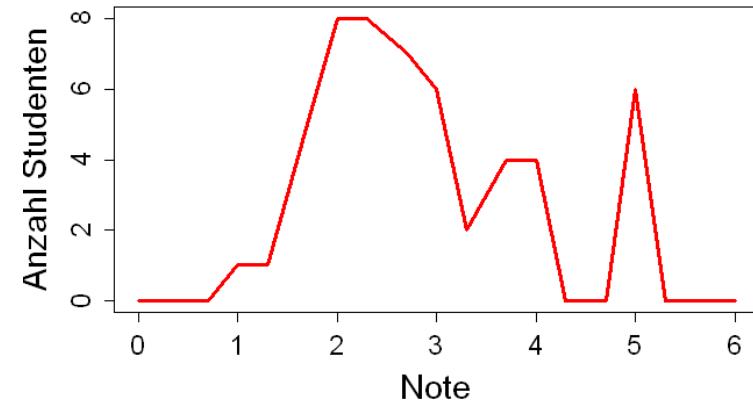
- Die Selektivität muss geschätzt werden, für Spezialfälle gibt es einfache Methoden:
  - Die Selektivität von  $\sigma_{R.A=c}$ , also Vergleich mit einer Konstante c beträgt  $1 / |R|$ , falls A ein Schlüssel ist
  - Falls A kein Schlüssel ist, aber die Werte gleichverteilt sind, ist  $sel=1 / I$  ( $I$  ist dabei die *image size*, d.h. die Anzahl versch. A-Werte in R)
  - Besitzt bei einem Equi-Join von R und S (Join Bed.: R.A=S.B) das Attribut A Schlüsseleigenschaft, kann die Größe des Join-Ergebnisses mit  $|S|$  abgeschätzt werden, da jedes Tupel aus S maximal einen Joinpartner findet. Die Selektivität ist also  $sel = 1/|R|$
  - logisches UND:  $sel_B(\sigma_{B1 \wedge B2}) = sel_B(\sigma_{B1}) \cdot sel_B(\sigma_{B2})$
  - logisches ODER:  $sel_B(\sigma_{B1 \vee B2}) = sel_B(\sigma_{B1}) + sel_B(\sigma_{B2}) - sel_B(\sigma_{B1}) \cdot sel_B(\sigma_{B2})$
  - logisches NICHT:  $sel_B(\sigma_{\neg B1}) = 1 - sel_B(\sigma_{B1})$



## Selektivität (cont.)

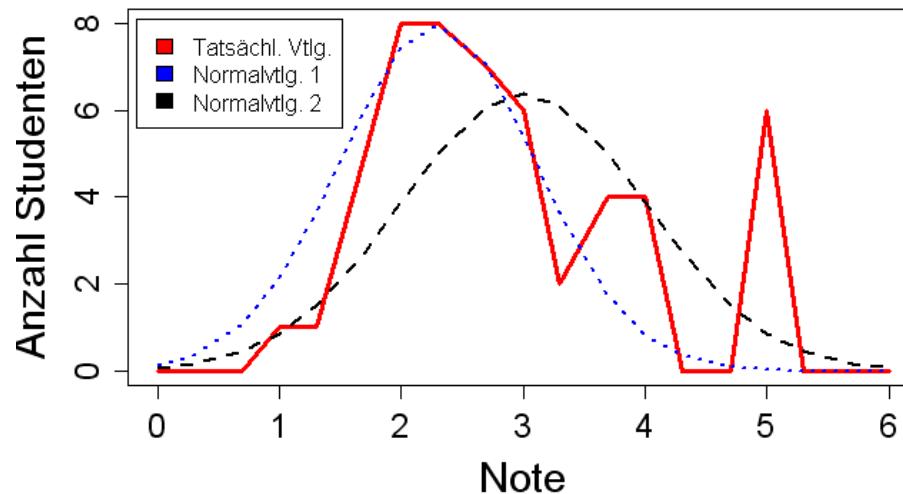
- Im Allgemeinen benötigt man anspruchsvollere Methoden um zu schätzen, wieviele Tupel sich in einem bestimmten Wertebereich befinden.
- Drei Grundsätzliche Arten von Schätzmethoden:
  - Parametrische Verteilungen
  - Histogramme
  - Stichproben

Beispiel: Schätzung der Verteilung der Noten der DBS II Klausur anhand des Ergebnisse von 2007:



## Selektivität (cont.)

- Parametrische Verteilungen
  - Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.

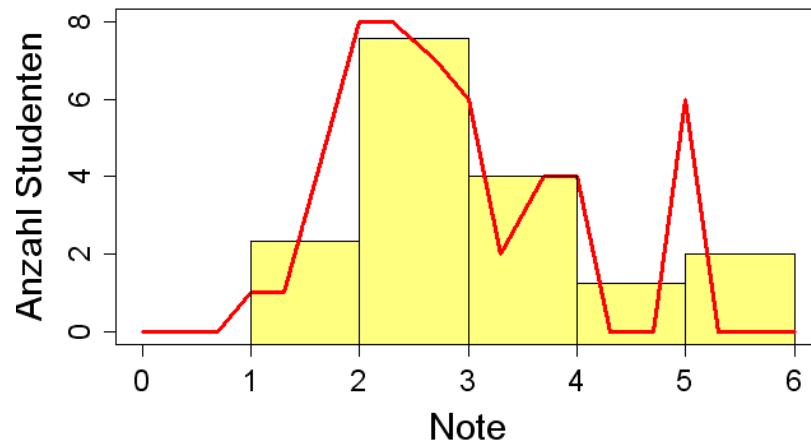


Probleme:

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

## Selektivität (cont.)

- Histogramme
  - Unterteile den Wertebereich des Attributs in Intervalle und zähle die Tupel, die in ein bestimmtes Intervall fallen
    - *Equi-Width-Histograms*: Intervalle gleicher Breite
    - *Equi-Depth-Histograms*: Unterteilung so, dass in jedem Intervall gleich viele Tupel sind



=> Flexible Annäherung an die Verteilung



## Selektivität (cont.)

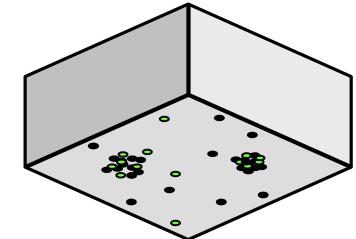
- Stichproben
  - Sehr einfaches Verfahren
  - Ziehe eine zufällige Menge von  $n$  Tupeln aus einer Relation, und betrachte deren Verteilung als repräsentativ für die gesamte Relation.
  - Problem der Größe des Stichprobenumfangs  $n$ :
    - $n$  zu klein: Wenig repräsentative Stichprobe
    - $n$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



- Probleme bei Anfragen über mehrere Attribute (mehr-dimensionale Anfragen)

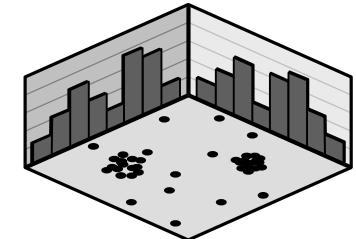
- Sampling

- **Problem:** Genauigkeit abhängig von der Samplegröße



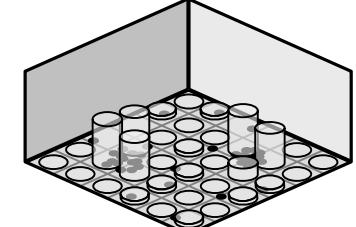
- 1D Histogramme

- **Problem:** Annahme der Unabhängigkeit zwischen den Attributen



- Multi-D Histogramme

- **Problem:** Anzahl der Gridzellen steigt exponentiell mit  $d$



- Parametrische Methoden

- **Problem:** nur für max. 2-3 Attribute geeignet



## Aufgabe: Finde geeignete Algorithmen & Datenstrukturen für einzelne Operationen

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

```
select *  
from R r, S s  
where r.A = s.B  
also  $R \bowtie_{A=B} S$ 
```
- Die Relationen R und S sind in Blöcken  $B_i(R)$  und  $B_j(S)$  auf dem Hintergrundspeicher abgelegt

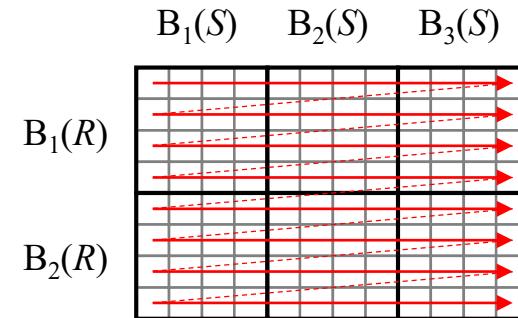


## Einfacher Nested-Loop-Join

- Algorithmus

```
for each Tupel  $r \in R$  do
    for each Tupel  $s \in S$  do
        if  $r.A = s.B$  then
             $result := result \cup (r \times s)$ 
```

## Matrixnotation (S 3 Blöcke, R 2 Blöcke)



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

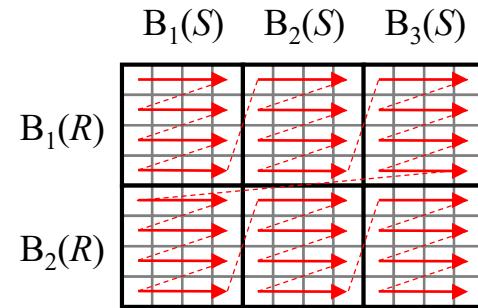


## Nested-Block-Loop-Join

- Berücksichtige, dass Relationen R und S auf Blöcke verteilt sind
- Algorithmus

Matrixnotation

```
for each Block  $B_R \in R$  do
    lade Block  $B_R$ 
    for each Block  $B_S \in S$  do
        lade Block  $B_S$ 
        for each Tupel  $r \in B_R$  do
            for each Tupel  $s \in B_S$  do
                if  $r.A = s.B$  then
                     $result := result \cup (r \times s)$ 
```



## Nested-Block-Loop-Join (cont.)

- Beispiel

| S         | Angestellter | Gehaltsgruppe |
|-----------|--------------|---------------|
| Müller    |              | 1             |
| Schneider |              | 2             |
| Schuster  |              | 1             |
| Schmidt   |              | 2             |
| Schütz    |              | 1             |

B<sub>S</sub>(1)  
B<sub>S</sub>(3)  
B<sub>S</sub>(3)

| R | Gehaltsgruppe | Gehalt |
|---|---------------|--------|
| 1 |               | 10.000 |
| 2 |               | 20.000 |
| 3 |               | 30.000 |

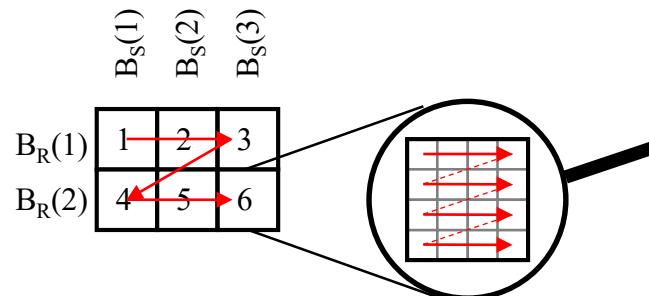
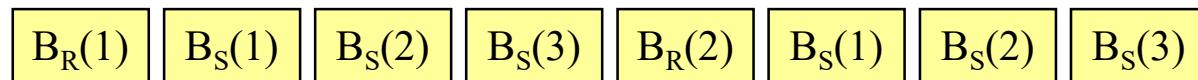
B<sub>R</sub>(1)  
B<sub>R</sub>(2)

- Anzahl Blockzugriffe:  $B_R + B_S \cdot B_R = 8$  Blockzugriffe ohne Cache  
( $B_R$  = Anzahl Blöcke der Relation R)
- D.h. die kleinere Relation sollte die äußere sein

## Cache Strategien für Nested-Block-Loop-Join

### Strategie 1

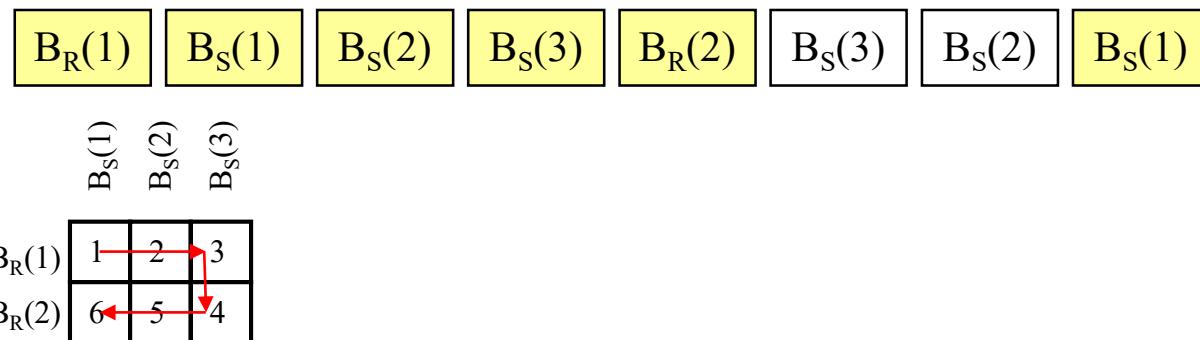
- Seiten der inneren Relation (S) im Cache halten
- Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R  
(  : Zugriff Platte)



## Cache Strategien für Nested-Block-Loop-Join

### Strategie 2

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

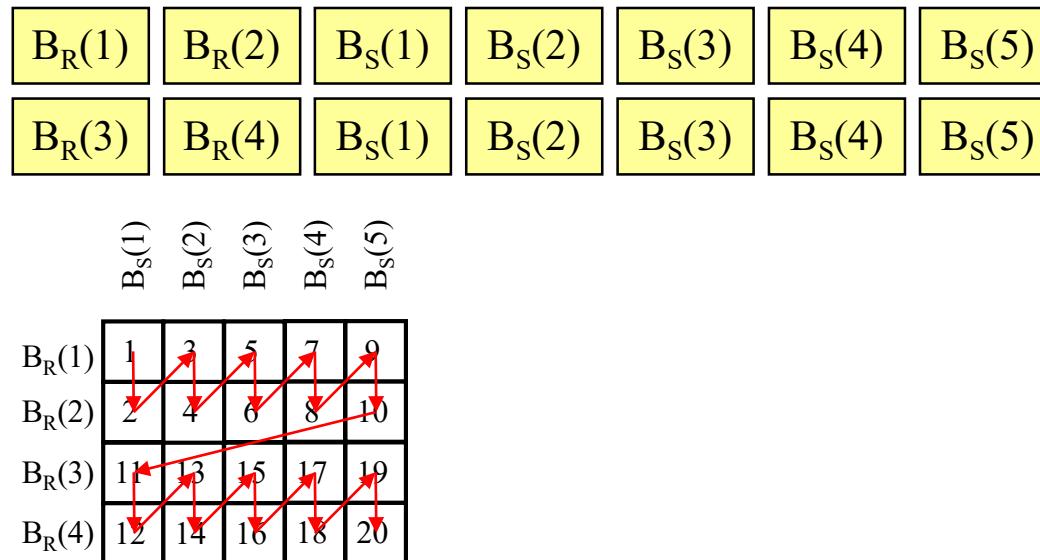


# Physische Anfrageoptimierung

## Cache Strategien für Nested-Block-Loop-Join

### Strategie 3

- $|C|-1$  Blöcke der äußeren Relation werden in den Cache eingelesen, zu jedem Block der inneren Relation werden diese Blöcke gejoint
- Blockzugriffe: 
$$B_R + B_S \cdot \left\lceil \frac{B_R}{|C|-1} \right\rceil$$
- Beispiel: 2 Seiten Cache für R, 1 Seite Cache für S





# Physische Anfrageoptimierung

## Cache Strategien für Nested-Block-Loop-Join

### Strategie 3 - Algorithmus

```
for  $i := 1$  to  $B_R$  step  $|C| - 1$  do
    lade Block  $B_R(i) \dots B_R(i + |C| - 2)$ 
    for each Block  $B_S \in S$  do
        lade Block  $B_S$ 
        for each Tupel  $r \in B_R(i) \dots B_R(i + |C| - 2)$  do
            for each Tupel  $s \in B_S$  do
                if  $r.A = s.B$  then
                     $result := result \cup (r \times s)$ 
```

- Leistung:
  - $|R|^*|S|$  Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
  - Effizienteste Ausführung von  $\theta$ -Joins mit  $\theta \neq '='$  (also allen Joins außer Equi-Joins)



## Blockgrößen-Optimierung NBL-Join

- Problem
  - Zu kleine Blockgröße:
    - Innere Relation wird in sehr kleinen Schritten eingelesen
    - Bei jedem I/O-Auftrag Latenzzeit des Plattenlaufwerks
  - Zu große Blockgröße (z.B.: Cache wird in 2-3 Blöcke geteilt):
    - Zu wenig Cache steht für die äußere Relation zur Verfügung
    - Innere Relation muss öfter gescanned werden
- Äquivalente Frage:  
Wie viel vom Cache für äußere/innere Relation?



## Blockgrößen-Optimierung NBL-Join (cont)

- Parameter
  - $f_R$  bzw.  $f_S$ : Größe der Relationen in Bytes
  - $c$ : Größe des Cache in Bytes
  - $t_{tr}$ : Transferzeit pro Byte
  - $t_{lat}$ : durchschnittliche Latenzzeit des Disk-Laufwerkes
  - $b$ : Blockgröße (Parameter, der optimiert wird)



# Physische Anfrageoptimierung

## Blockgrößen-Optimierung NBL-Join (cont)

- I/O-Kosten

|                       | <b>Äußere Relation R</b>  | <b>Innere Relation S</b>  |
|-----------------------|---|---|
| Anzahl Block-zugriffe | $B_R$   | $B_R + B_S \cdot \left\lceil \frac{B_R}{ C -1} \right\rceil$  |
|                       | Suchen zum aktuellen Block von R + Suchen zum Start von S   |   |
| $t_{NL-Join} \approx$ | $\left\lceil \frac{B_R}{ C -1} \right\rceil \cdot (2t_{seek} + t_{lat} + b \cdot ( C -1) \cdot t_{tr})$ | $+ B_S \cdot \left\lceil \frac{B_R}{ C -1} \right\rceil \cdot (t_{lat} + b \cdot t_{tr})$   |
|                       | in einer Leseoperation werden $ C -1$ Blöcke der äußeren Relation gelesen                               | Jeweils ein Block wird gelesen, aber nächster Block startet meist auf gleicher Spur   |
| $t_{NL-Join} \approx$ | <i>ignorieren, da nur 1x und in großen Blöcken</i>  | $\left( \left\lceil \frac{f_s}{b} \right\rceil \cdot \left\lceil \frac{\lceil f_R/b \rceil}{\lfloor c/b \rfloor - 1} \right\rceil \right) \cdot (t_{lat} + b \cdot t_{tr})$ |



# Physische Anfrageoptimierung

## Blockgrößen-Optimierung NBL-Join (cont)

- I/O-Kosten

$$t_{NL-Join} \approx \left( \left\lceil \frac{f_s}{b} \right\rceil \cdot \left\lceil \frac{\lceil f_R / b \rceil}{\lfloor c / b \rfloor - 1} \right\rceil \right) \cdot (t_{lat} + b \cdot t_{tr})$$

- Weglassen der Rundungsfunktion (unproblematisch für  $f_R, f_s \gg b$ , d.h. relativer Fehler ist vernachlässigbar) ergibt stückweise differenzierbaren Term

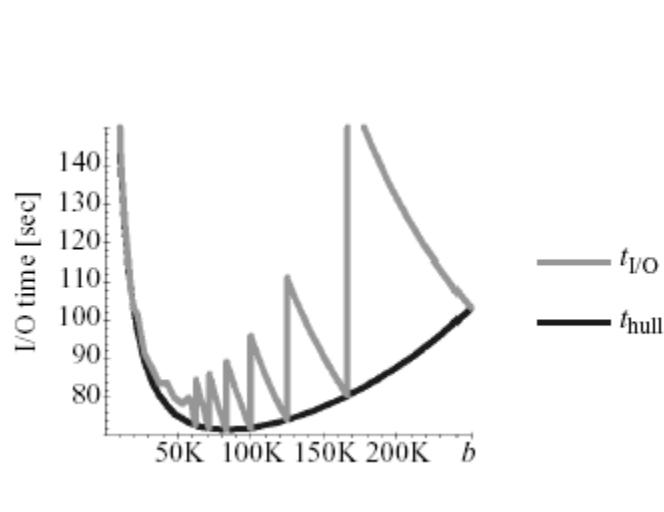
$$t_{NL-Join} \approx \left( \frac{f_s \cdot f_R}{b^2 \cdot (\lfloor c / b \rfloor - 1)} \right) \cdot (t_{lat} + b \cdot t_{tr})$$

# Physische Anfrageoptimierung

## Blockgrößen-Optimierung NBL-Join (cont)

$$t_{NL-Join} \approx \left( \frac{f_S \cdot f_R}{b^2 \cdot (\lfloor c/b \rfloor - 1)} \right) \cdot (t_{lat} + b \cdot t_{tr})$$

- Optimierung der Hüllfunktion



$$t_{hull} \approx \left( \frac{f_S \cdot f_R}{b^2 \cdot ((c/b) - 1)} \right) \cdot (t_{lat} + b \cdot t_{tr})$$

Joinkosten bei

- $f_R = f_S = 10\text{MByte}$
- $c = 500\text{ KByte}$
- $t_{lat} = 5\text{ ms}$
- $t_{tr} = 0,25\text{ s /MByte}$
- $b_{opt} = 85\text{ KByte}$



## Blockgrößen-Optimierung NBL-Join (cont)

- Optimierung durch Differenzieren
  - Gleichsetzen der 1. Ableitung mit 0
  - 2 Lösungen, von denen nur eine positiv ist

$$0 = \frac{\partial}{\partial b} t_{hull} \Rightarrow b_{opt} = \frac{\sqrt{t_{lat}^2 + t_{tr} \cdot t_{lat} \cdot c} - t_{lat}}{t_{tr}}$$

- Lösung ist Minimum (aus 2. Ableitung erkennbar)
- An den Stellen, an denen  $\lfloor c/b \rfloor$  konstant ist, ist  $t_{NLJoin}$  streng monoton fallend (negative Ableitung)
- Deshalb kann das Minimum von  $t_{NLJoin}$  nur an der ersten Sprungstelle links oder rechts vom Minimum von  $t_{hull}$  sein:

$$b_1 = c / \left\lfloor \frac{c}{b_{opt}} \right\rfloor, \quad b_2 = c / \left\lceil \frac{c}{b_{opt}} \right\rceil$$



# Physische Anfrageoptimierung

## Blockgrößen-Optimierung NBL-Join (cont)

- Was sind denn eigentlich die CPU-Kosten?
  - Im wesentlichen müssen  $|S| * |R|$  Vergleiche durchgeführt werden
- Vergleich: I/O- versus CPU-Kosten
  - Beispiel:  $|S| = |R| = 100.000$  Tupel und  $f_S$  und  $f_R$  jeweils ca. 10 MB
  - Bearbeitungszeit eines Vergleichs durchschnittlich 0,1  $\mu$ s
  - I/O-Zeit bei optimaler Seitengröße (85 KB) ca. 75 sec
  - CPU-Zeit: ca. 1000 sec

**=> Der NBL-Join ist CPU-bound!!!**

- Daher im Folgenden:  
Maßnahmen zur Senkung des CPU-Aufwands



## Sort-Merge-Join

- Zweistufiger Algorithmus

- 1. Schritt:

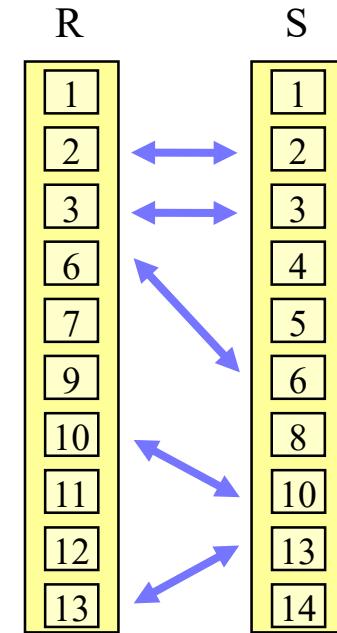
```
sortiere R bzgl. Attribut A
sortiere S bzgl. Attribut B
```

- 2. Schritt:

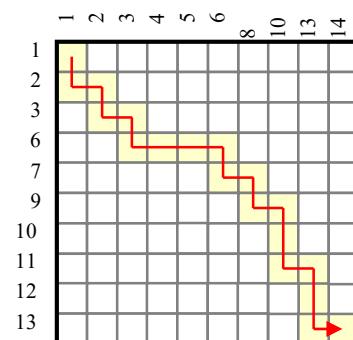
```
j = 1;
s = erstes Tupel von S;
for i = 1 to |R| do
    r = i - tes Tupel von R;
    while s.B < r.A
        j = j + 1;
        s = j - tes Tupel von S;
        if r.A = s.B then
            result := result ∪ ((r - r.A) × s);
```

Achtung: Dieser Algorithmus funktioniert nur, falls R und S auf dem Joinattribut keine Duplikate enthalten.

Wie muss der Algorithmus erweitert werden um Duplikate zu erfassen?



## Matrixnotation





## Sort-Merge-Join (cont.)

- Leistung
  - Jede Relation wird genau einmal durchlaufen:  $O(|R| + |S|)$  Vergleiche
  - Sortieren der Relation kostet  $O(|R| \cdot \log |R| + |S| \cdot \log |S|)$
  - Sortieren ist nicht notwendig, wenn bereits ein Index existiert
  - Verfahren versagt, wenn in beiden Relationen sehr viele Duplikate (d.h. mehr als in den Puffer passen) auftreten. In diesem Fall muss auf Nested-Loop-Join umgeschaltet werden



## Einfacher Hash-Join

- Reduktion des CPU-Aufwandes bei der Join-Berechnung
  - Der Join-Partner eines S-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, anstatt das S-Tupel sequentiell mit jedem Tupel der Relation R zu vergleichen.
  - Zu diesem Zweck wird die Relation R gehasht, d.h. es wird zu allen Tupeln der Hash-Key bestimmt und die Tupel in einer Tabelle unter diesem Key eingetragen.
  - Nicht alle R-Tupel, die den passenden Hash-Key haben, sind Join-Partner eines S-Tupels, aber alle Join-Partner haben denselben Hash-Key.
  - Im Idealfall soll der Join im Hauptspeicher ablaufen: die Hashtabelle soll für die kleinere Relation erzeugt werden.
  - Hash-Join Verfahren können nur für Equi-Join und Natürlichen Join effizient genutzt werden.



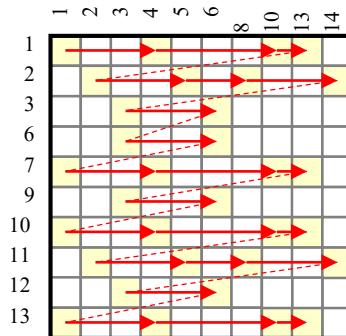
# Physische Anfrageoptimierung

## Einfacher Hash-Join (cont.)

### – Algorithmus

```
for each Tupel  $r \in R$  do
    berechne  $adr = \text{hash}(r)$ ;
    speichere  $r$  in  $HT[adr]$  ab;
for each Tupel  $s \in S$  do //prüfe in der Hashtabelle  $HT$ 
    berechne  $adr = \text{hash}(s)$ ;
    for each Tupel  $r \in HT[adr]$  do
        if  $r.A = s.B$  then
             $result := result \cup ((r - r.A) \times s)$ 
```

### Matrixnotation



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

### – Leistung

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



## Hashed-Loop-Join

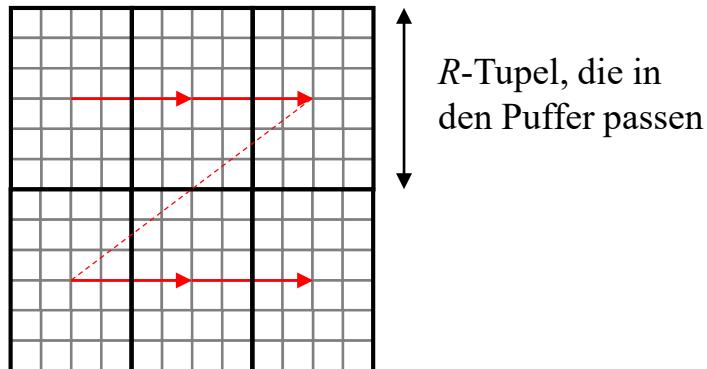
- Kombination aus dem Nested-Loop-Join und dem einfachen Hash-Join
- Relation R wird in große Blöcke eingeteilt, deren Hashtabellen in den Puffer passen
- Für jeden dieser Blöcke wird die Relation S gescannt und ein einfacher Hash-Join durchgeführt
- Algorithmus

```
repeat
    lese soviel Tupel von R in Hauptspeicher bis der Platz aufgebraucht ist;
    erzeuge für diese Tupel eine Hashtabelle HT;
    for each Tupel s ∈ S do
        berechne adr = hash(s);
        for each Tupel r ∈ HT[adr] do
            if r.A = s.B then
                result := result ∪ ((r - r.A) × s)
        until alle Tupel der Relation R sind eingelesen;
```

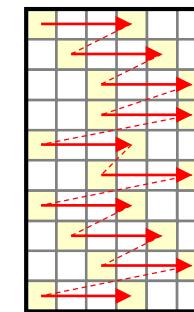


## Hashed-Loop-Join (cont.)

### Matrixnotation

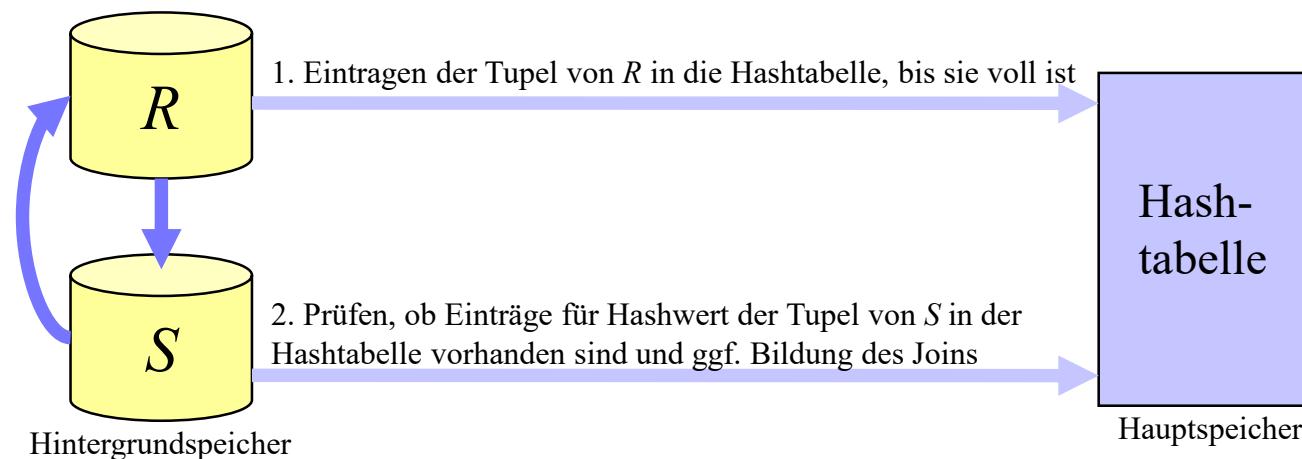


auf den einzelnen Blöcken: Hash-Join



### Ablauf

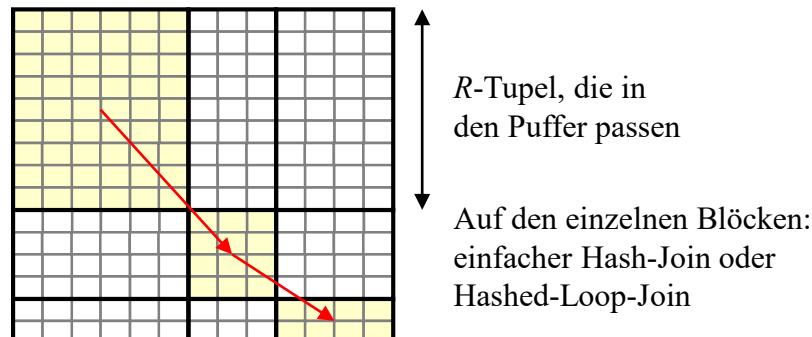
Schritt A:



Schritt B: Wiederhole Schritt A für die restlichen Tupel von  $R$

## Hash-Partitioned-Join (a.k.a. GRACE)

- Hashed-Loop-Join zerlegt Relationen willkürlich in Blöcke, jeder Block von R muss mit jedem Block von S kombiniert werden
- Idee: Zerlege R und S mit einer Hashfunktion in Partitionen, so dass nur Partitionen mit demselben Hash-Key kombiniert werden müssen
- Zweistufiges Verfahren
  - Partitioniere die Relationen R und S in  $R_1, \dots, R_N$  und  $S_1, \dots, S_N$
  - Berechne den Join der einzelnen Partitionen  $R_i$  und  $S_i$  mit einem beliebigen Join Verfahren
- Matrixnotation



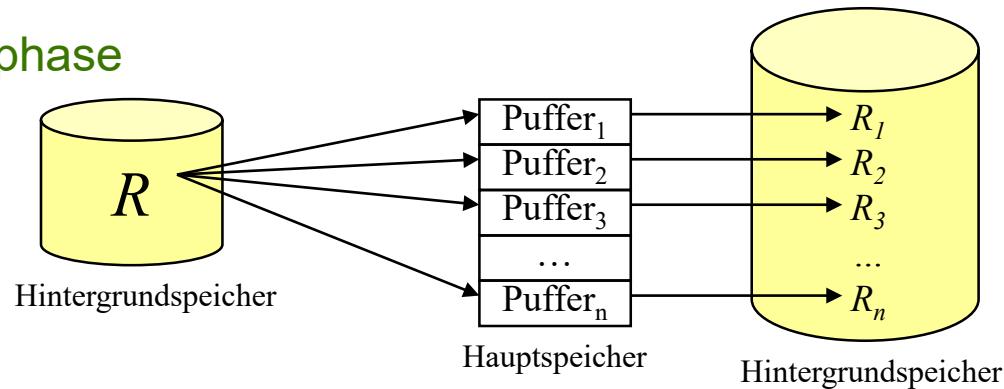


# Physische Anfrageoptimierung

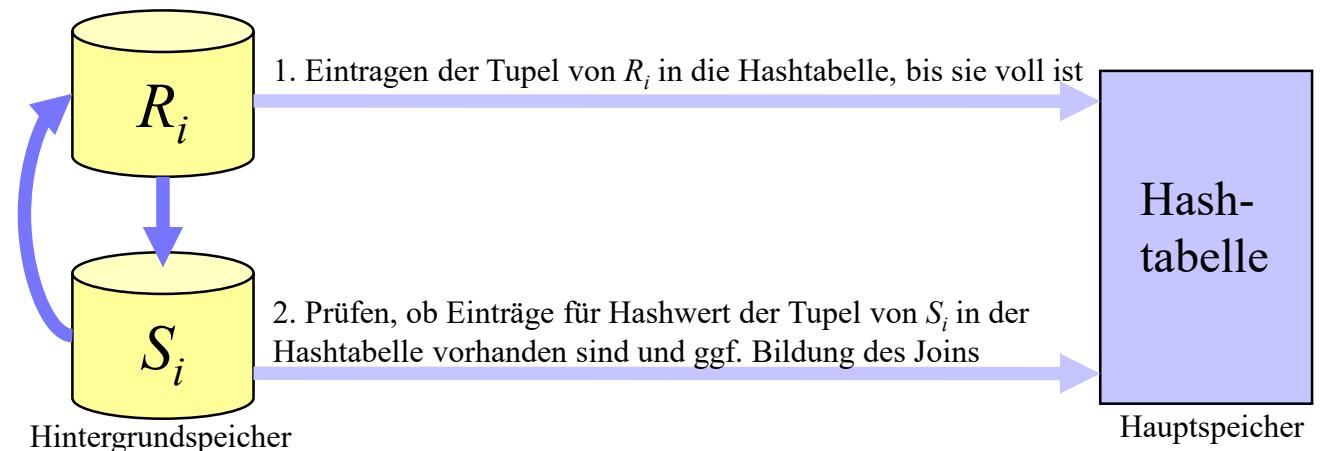
## Hash-Partitioned-Join (a.k.a. GRACE) (cont.)

### – Ablauf

- Partitionierungsphase



- Join-Phase



Skript zur Vorlesung:  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 10a

# Transaktionen - Synchronisation

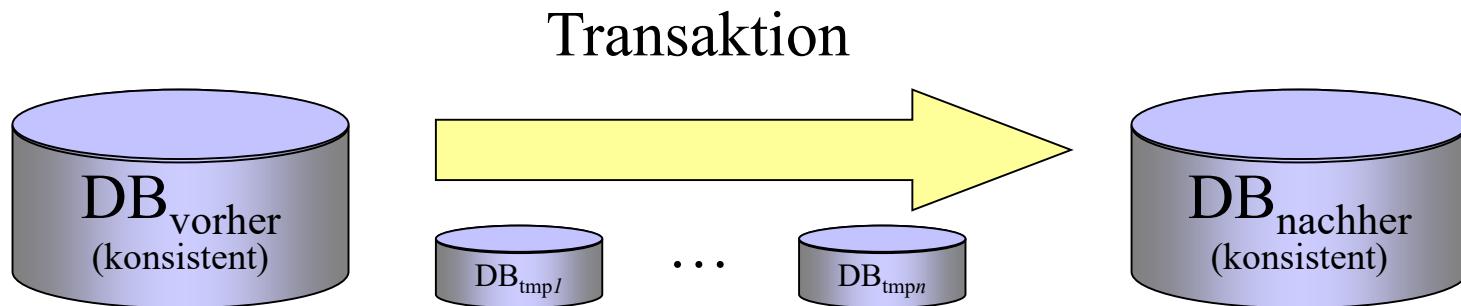
Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani





# 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 um 200 €
- Möglicher Ablauf

| Konto | Kunde | Stand   |     | Konto | Kunde | Stand   |                    |
|-------|-------|---------|-----|-------|-------|---------|--------------------|
|       | Meier | 1.000 € | (1) |       | Meier | 1.000 € |                    |
|       | Huber | 1.500 € |     |       | Huber | 1.300 € | (2) System-absturz |

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



- **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



- **begin of transaction (BOT)**
  - markiert den Anfang einer Transaktion
  - In SQL werden Transaktionen 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;
```

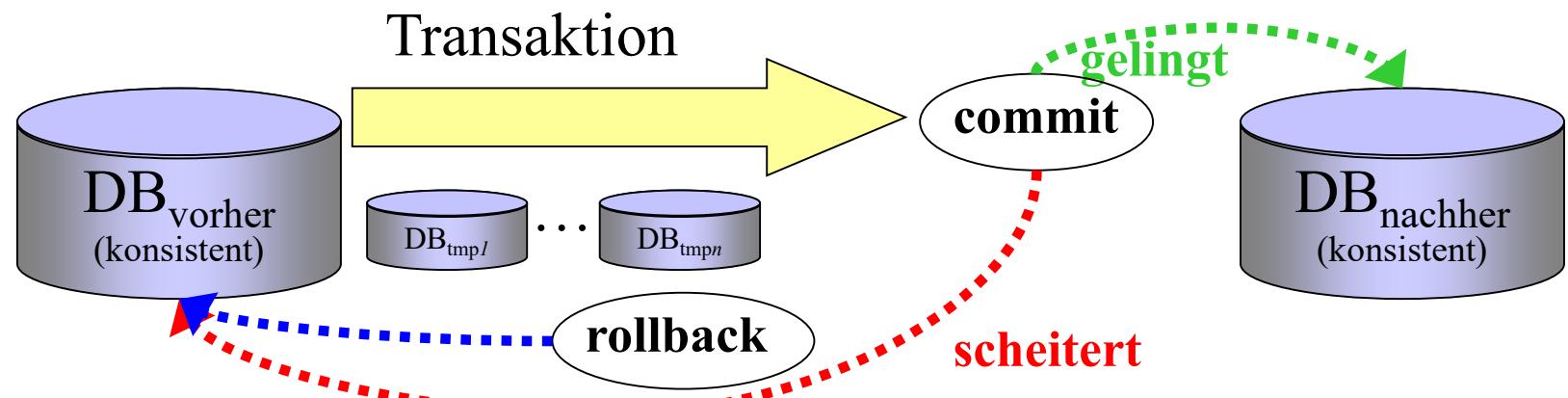


Unterstützung langer Transaktionen durch

- **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. Synchronisation (Concurrency Control)

Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

## 2. Datensicherheit (Recovery)

Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)

## 3. Integrität (Integrity)

Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer



## Synchronisation (Concurrency Control)

- Serielle Ausführung von Transaktionen
  - unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist
  - Folgen: 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



# Einleitung

## Anomalien im Mehrbenutzerbetrieb

Wir unterscheiden u.a. folgende Grundmuster von Anomalien:

- Verloren gegangene Änderungen (Lost Updates)
  - Zugriff auf „schmutzige“ (nicht dauerhaft gültige) Daten (Dirty Read / 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 TA können durch Änderungen anderer TA 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`;



# Einleitung

- Möglicher Ablauf

| T1   | T2  |
|--|---|
| <pre>read(Passagiere.Gepäck, x1);<br/><br/>x1 := x1+3;<br/>write(Passagiere.Gepäck, x1);</pre> | <pre>read(Passagiere.Gepäck, x2);<br/>x2 := x2 + 5;<br/>write(Passagiere.Gepäck, x2);</pre> |

- 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
- Beispiel:
  - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
  - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen



# Einleitung

- Möglicher Ablauf:

| T1   | T2   |
|--|--|
| UPDATE Passagiere<br>SET Gepäck = Gepäck+3;<br><br>ROLLBACK; | UPDATE Passagiere<br>SET Gepäck = Gepäck+5;<br>COMMIT; |

- Durch Abbruch von T1 werden die geänderten Werte ungültig, die T2 gelesen hat (Dirty Read). T2 setzt weitere Änderungen darauf auf (Dirty Write)

→ Verstoß gegen

- **Consistency:** Ablauf verursacht inkonsistenten DB-Zustand oder
- **Durability:** T2 muss zurückgesetzt werden



# Einleitung

## Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Beispiel:
  - 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



# Einleitung

- Möglicher Ablauf:

| T1   | T2  |
|--|---|
| <pre>SELECT Gepäck FROM Passagiere<br/>WHERE FlugNr = „BA932“;</pre> |   |
| <pre>SELECT Gepäck FROM Passagiere<br/>WHERE FlugNr = „BA932“;</pre> | <pre>INSERT INTO Passagiere<br/>VALUES (BA932, Meier, 3F, 5);<br/>COMMIT;</pre> |

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl T1 den DB-Zustand nicht geändert hat

→ Verstoß gegen *Isolation*



## Phantomproblem

- Spezialfall des nicht-reproduzierbaren Lesens, bei der neu generierte Daten, sowie meist bei der 2. TA Aggregationsfunktionen 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



# Einleitung

- Möglicher Ablauf

| T1  | T2  |
|---|---|
| <pre>SELECT * FROM Passagiere WHERE FlugNr = „LH745“;  SELECT COUNT(*) FROM Passagiere WHERE FlugNr = „ LH745“;</pre> | <pre>INSERT INTO Passagiere VALUES (LH745, Phantomas, 7D, 2); COMMIT;</pre> |

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



## Motivation

- Bearbeitung von Transaktionen
  - Nebenläufigkeit vor den Benutzern verbergen
  - Transparent für den Benutzer, als ob  
**TAs (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden**  
und NICHT als ob  
TAs ineinander verzahnt ablaufen und sich dadurch (unbeabsichtigt) beeinflussen



## Schedules

- Allgemeiner Schedule:  
Ein Schedule („Historie“) für eine Menge  $\{T_1, \dots, T_n\}$  von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der Transaktionen  $T_i$  entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.
- Allgemeine Schedules bieten offenbar eine beliebige Verzahnung ***und sind daher aus Performanz-Gründen erwünscht***
- Frage: Warum darf die Reihenfolge der Aktionen innerhalb einer TA nicht verändert werden?



# Serialisierbarkeit von Transaktionen

- Serieller Schedule:  
Ein serieller Schedule ist ein Schedule S von  $\{T_1, \dots, T_n\}$ , in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden.
- Aus Sicht des Isolation-Prinzips **sind serielle Schedules erwünscht**

**Kompromiss zwischen Performanz und Isolation (bzw. allgem. und seriellen Schedules):**

## **Serialisierbarer Schedule:**

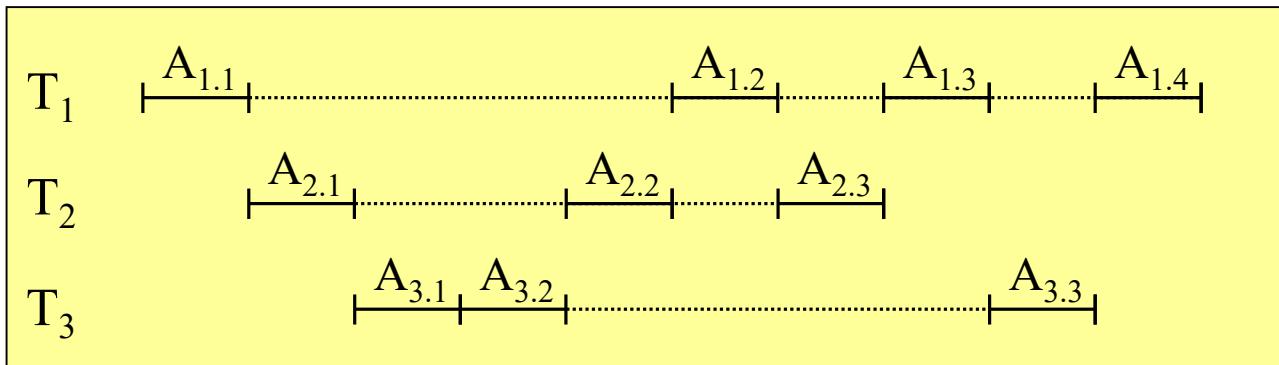
Ein (allgemeiner) Schedule S von  $\{T_1, \dots, T_n\}$  ist serialisierbar, wenn er dieselbe Wirkung hat wie ein beliebiger serieller Schedule von  $\{T_1, \dots, T_n\}$ .

- Nur serialisierbare Schedules dürfen zugelassen werden!

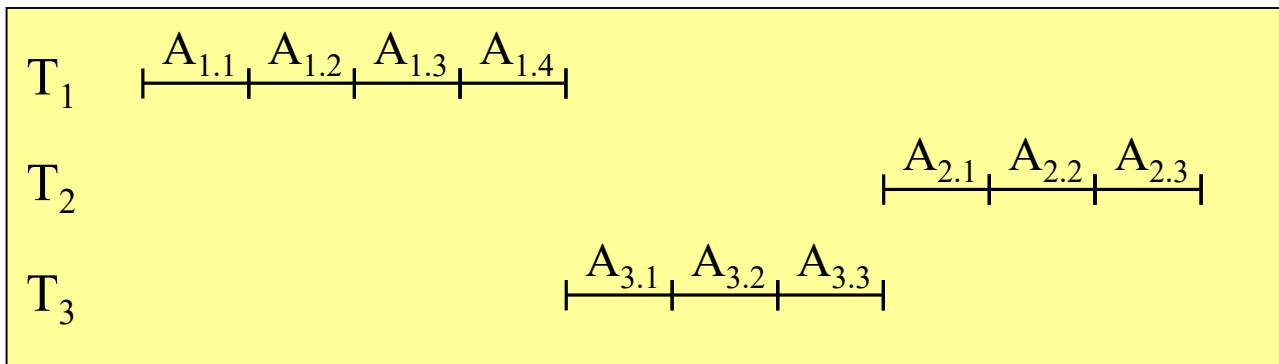


# Serialisierbarkeit von Transaktionen

- Beispiele
  - Beliebiger Schedule:



- Serieller Schedule:





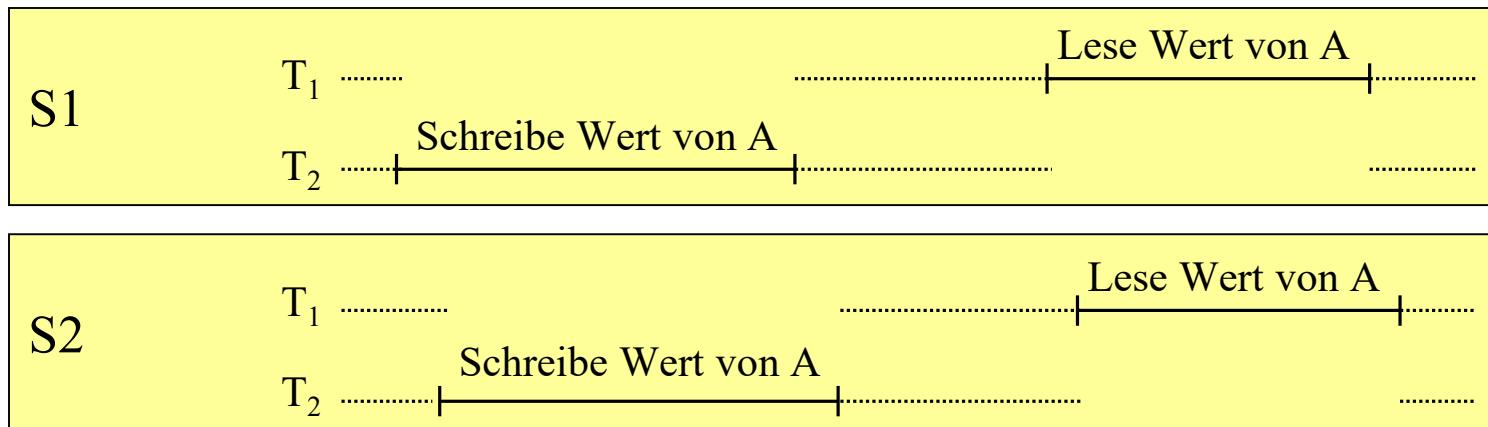
## Wirkung von Schedules

- Frage: Wann haben zwei Schedules S1 und S2 die gleiche Wirkung auf den Datenbank-Inhalt?
- Achtung:
  - Gleiches Ergebnis kann u.a. Ergebnis eines Zufalls sein
  - Dies könnte aber nur durch nachträgliches Überprüfen der Datenbank-Zustände nach S1 und S2 festgestellt werden.



# Serialisierbarkeit von Transaktionen

- Wir benötigen ein objektivierbares Kriterium:  
**Konflikt-Äquivalenz**
- Idee: Wenn in S1 eine Transaktion  $T_1$  z.B. einen Wert liest, den  $T_2$  geschrieben hat, dann muss das auch in S2 so sein.



- Wir sprechen hier von einer Schreib-Lese-Abhängigkeit (bzw. Konflikt) zwischen  $T_2$  und  $T_1$  (in Schedule S1 und S2)



## Abhängigkeiten

Sei  $S$  ein Schedule. Wir sprechen von einer

- Schreib-Lese-Abhängigkeit von  $T_i \rightarrow T_j$ 
  - Es existiert Objekt  $x$ , so dass in  $S$   $w_i(x)$  vor  $r_j(x)$  kommt
  - Abkürzung:  $wr_{i,j}(x)$
- Lese-Schreib-Abhängigkeit von  $T_i \rightarrow T_j$ 
  - Es existiert Objekt  $x$ , so dass in  $S$   $r_i(x)$  vor  $w_j(x)$  kommt
  - Abkürzung:  $rw_{i,j}(x)$
- Schreib-Schreib-Abhängigkeit von  $T_i \rightarrow T_j$ 
  - Es existiert Objekt  $x$ , so dass in  $S$   $w_i(x)$  vor  $w_j(x)$  kommt
  - Abkürzung:  $ww_{i,j}(x)$
- ***Warum keine Lese-Lese-Abhängigkeiten?***



# Serialisierbarkeit von Transaktionen

## Konfliktäquivalenz von Schedules

- Zwei Schedules  $S_1$  und  $S_2$  heißen konfliktäquivalent, wenn
  - $S_1$  und  $S_2$  die gleichen Transaktions- und Aktionsmengen besitzen, d.h. wenn beide Schedules dieselben Operationen ausführen.
  - $S_1$  und  $S_2$  die gleichen Abhängigkeitsmengen besitzen, d.h. wenn in der Abhängigkeitsmenge von  $S_1$  z.B. die Schreib-Lese-Abhängigkeit " $w_i(x)$  vor  $r_j(x)$ " vorkommt (für ein Objekt  $x$ ), dann muss diese auch in der Abhängigkeitsmenge von  $S_2$  vorkommen.
- Zwei konflikt-äquivalente Schedules haben die gleiche Wirkung auf den Datenbank-Inhalt. (Gilt die Umkehrung?)



# Serialisierbarkeit von Transaktionen

- Beispiel:

$$S_1 = (r_1(x), r_1(y), r_2(x), w_2(x), w_1(x), w_1(y))$$

$$S_2 = (r_2(x), r_1(x), r_1(y), w_2(x), w_1(x), w_1(y))$$

$$S_3 = (r_1(x), r_1(y), r_2(x), w_1(x), w_2(x), w_1(y))$$

$$S_4 = (r_2(x), r_1(y), r_1(x), w_2(x), w_1(y), w_1(x))$$

$r_i(x) = T_i$  liest  $x$

$w_i(x) = T_i$  schreibt  $x$

- Aktionsmengen von S1, S2 und S3 sind identisch
- Abhängigkeitsmengen:

$$A_{S1} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{2,1}(x)\}$$

$$A_{S2} = \{rw_{2,1}(x), rw_{1,2}(x), ww_{2,1}(x)\}$$

$$A_{S3} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{1,2}(x)\}$$

- Schedule S1 und S2 sind konfliktäquivalent
- Schedule S1 und S3, bzw. S2 und S3 sind nicht konfliktäquivalent
- Schedule S4 ist kein Schedule derselben Transaktionen, da die Aktionen transaktionsintern vertauscht sind.



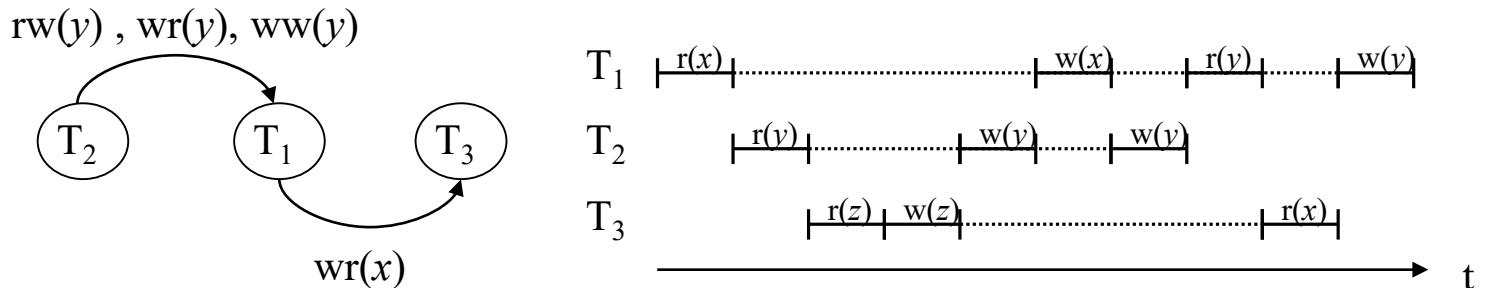
## Serialisierungs-Graph

- Überprüfung, ob ein Schedule von  $\{T_1, \dots, T_n\}$  serialisierbar ist (d.h. ob ein konflikt-äquivalenter serieller Schedule existiert)
- Die beteiligten Transaktionen  $\{T_1, \dots, T_n\}$  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  $ww(x)$

Die Kanten werden mit der Abhängigkeit beschriftet.

# Serialisierbarkeit von Transaktionen

- Es gilt:
  - Ein Schedule ist serialisierbar, falls der Serialisierungs-Graph **zyklenfrei** ist
  - Einen zugehörigen konfliktäquivalenten seriellen Schedule erhält man durch topologisches Sortieren des Graphen (**Serialisierungsreihenfolge**)
  - Es kann i.A. mehrere serielle Schedules geben.
  - Beispiel:  $S = (\text{r}_1(x), \text{r}_2(y), \text{r}_3(z), \text{w}_3(z), \text{w}_2(y), \text{w}_1(x), \text{w}_2(y), \text{r}_1(y), \text{r}_3(x), \text{w}_1(y))$



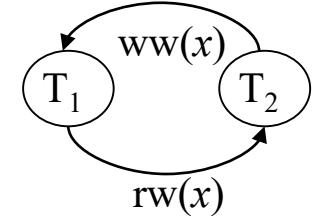
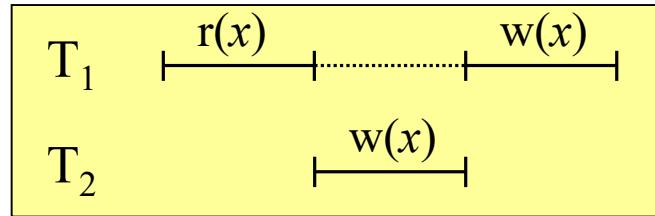
Serialisierungsreihenfolge: (T<sub>2</sub>, T<sub>1</sub>, T<sub>3</sub>)



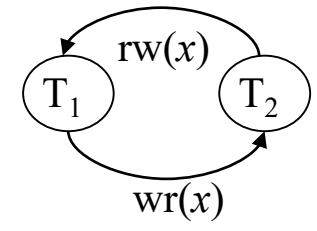
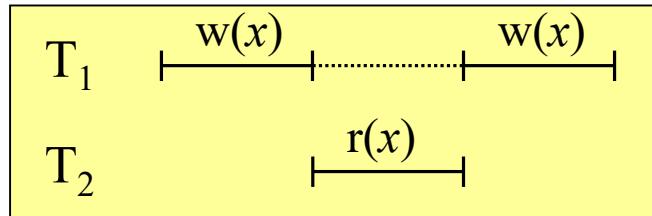
# Serialisierbarkeit von Transaktionen

## Beispiele für nicht-serialisierbare Schedules

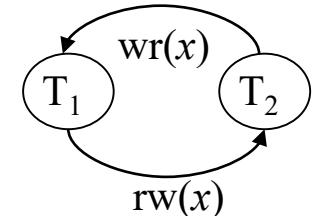
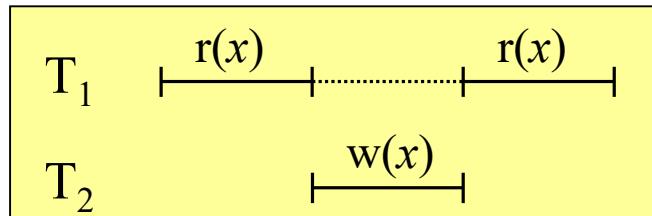
Lost Update:  $S=(r_1(x), w_2(x), w_1(x))$



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



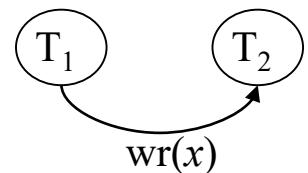
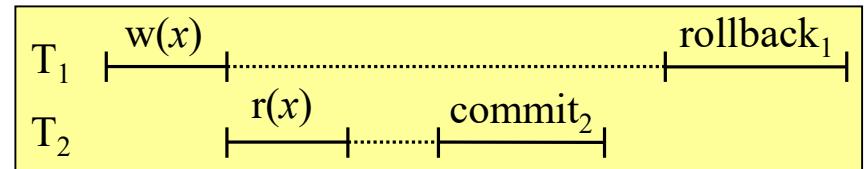
Non-repeatable Read:  $S=(r_1(x), w_2(x), r_1(x))$



# Serialisierbarkeit von Transaktionen

## Rücksetzbare Schedules

- Bisher: Serialisierbarkeit
- Frage: was passiert, wenn eine Transaktion (z.B. auf eigenen Wunsch) zurückgesetzt wird?
- Beispiel:
  - $T_1$  schreibt Datensatz  $x$
  - $T_2$  liest Datensatz  $x$
  - $T_2$  führt **COMMIT** aus
  - Schedule ist serialisierbar,  
der Serialisierungs-Graph ist zyklenfrei
- ABER
  - $T_1$  wird zurückgesetzt (d.h. Datensatz  $x$  wird wieder auf den Ursprungswert zurückgesetzt)
  - $T_2$  müsste eigentlich auch zurückgesetzt werden, hat aber schon **COMMIT** ausgeführt





# Serialisierbarkeit von Transaktionen

- Also: Serialisierbarkeit alleine reicht leider nicht aus, wenn TAs zurückgesetzt werden können
- ***Rücksetzbarer Schedule:***  
Eine Transaktion  $T_i$  darf erst dann ihr COMMIT durchführen, wenn alle Transaktionen  $T_j$ , von denen sie Daten gelesen hat, beendet sind.
- Andernfalls Problem: Falls ein  $T_j$  noch zurückgesetzt wird, müsste auch  $T_i$  zurückgesetzt werden, was nach  $COMMIT(T_i)$  nicht mehr möglich wäre



# Serialisierbarkeit von Transaktionen

- Noch schlimmer:  
Rücksetzbare Schedules können eine Lawine weiterer Rollbacks in Gang setzen

| Schritt | $T_1$              | $T_2$    | $T_3$    | $T_4$    | $T_5$    |
|---------|--------------------|----------|----------|----------|----------|
| 1.      | $w_1(A)$           |          |          |          |          |
| 2.      |                    | $r_2(A)$ |          |          |          |
| 3.      |                    | $w_2(B)$ |          |          |          |
| 4.      |                    |          | $r_3(B)$ |          |          |
| 5.      |                    |          | $w_3(C)$ |          |          |
| 6.      |                    |          |          | $r_4(C)$ |          |
| 7.      |                    |          |          | $w_5(D)$ |          |
| 8.      |                    |          |          |          | $r_5(D)$ |
| 9.      | abort <sub>1</sub> |          |          |          |          |

- ***Schedule ohne kaskadierendes Rücksetzen:***  
Änderungen werden erst nach dem *COMMIT* für andere Transaktionen zum Lesen freigegeben

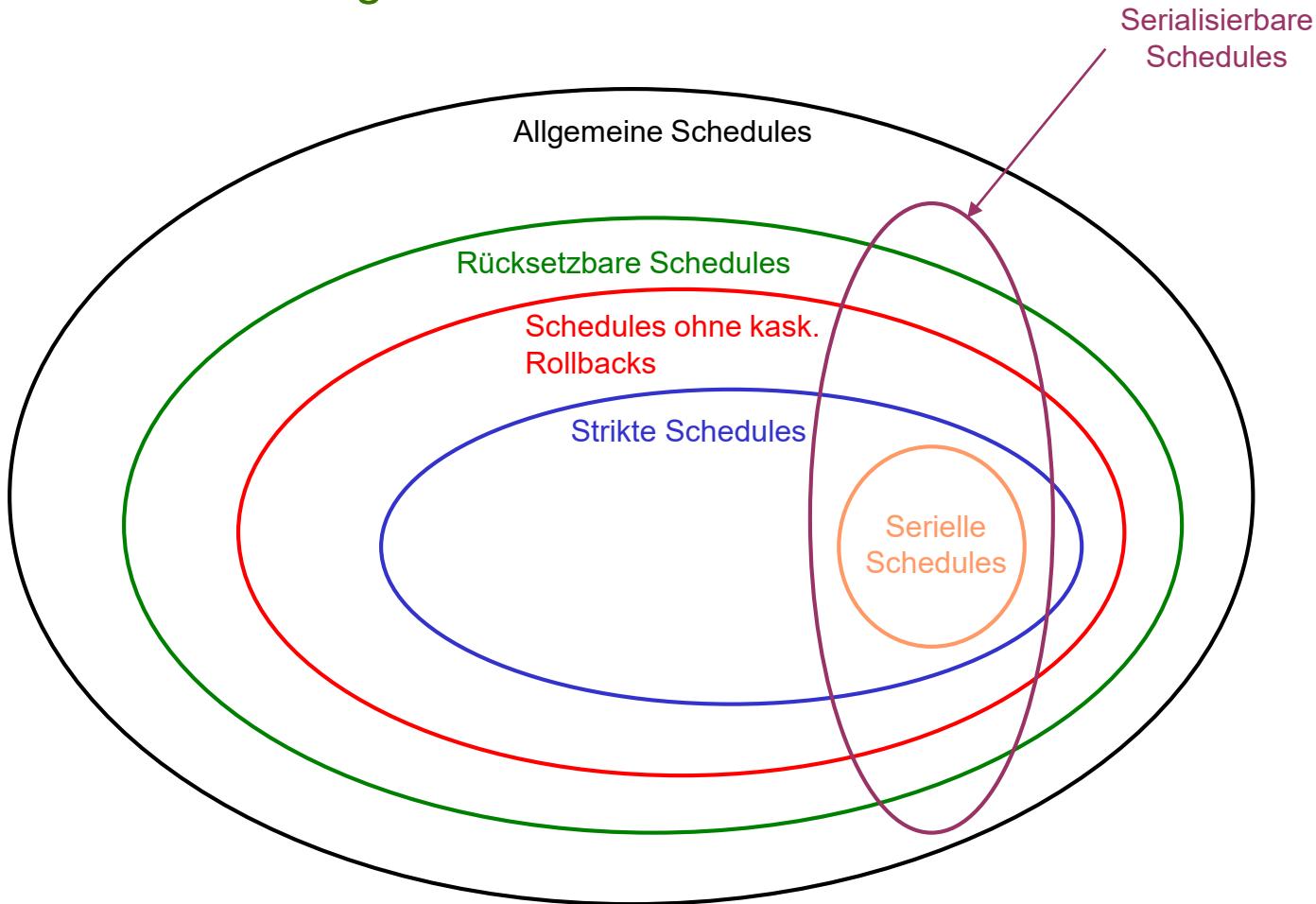


## Überblick: Scheduleklassen

- Serieller S.
  - TAs in einzelnen Blöcken, phys. Einbenutzerbetrieb
- Serialisierbarer S.
  - Konfliktäquivalent zu einem seriellen S.
- Rücksetzbarer S.
  - TA darf erst committen, wenn alle TAs von denen sie Daten gelesen hat committed haben
- S. ohne kaskadierendes Rollback
  - Veränderte Daten einer noch laufenden TA dürfen nicht gelesen werden
- Strikter S.
  - Zusätzlich dürfen veränderte Daten einer noch laufenden TA nicht überschrieben werden

# Serialisierbarkeit von Transaktionen

- Überblick: Beziehungen zwischen Scheduleklassen





## Techniken zur Synchronisation

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



## Allgemeines:

- Sperrverfahren sind DAS Standardverfahren zur Synchronisation in relationalen DBMS

## Sperre (Lock)

- Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
- Anforderung einer Sperre durch *LOCK*, z.B. *L(x)* für *LOCK* auf Objekt x
- Freigabe durch *UNLOCK*, z.B. *U(x)* für *UNLOCK* von Objekt x
- *LOCK / UNLOCK* erfolgt atomar (also nicht unterbrechbar!)
- Sperrgranularität (Objekte, auf denen Sperren gesetzt werden):  
Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
- Sperrenverwalter führt Tabelle für aktuell gewährte Sperren



## Legale Schedules

- Vor jedem Zugriff auf ein Objekt wird eine geeignete Sperre gesetzt.
- Keine Transaktion fordert eine Sperre an, die sie schon besitzt.
- Spätestens bei Transaktionsende werden alle Sperren zurückgegeben.
- Sperren werden respektiert, d.h. eine mit gesetzten Sperren unverträgliche Sperranforderung (z.B. exklusiver Zugriff auf Objekt x) muss warten.

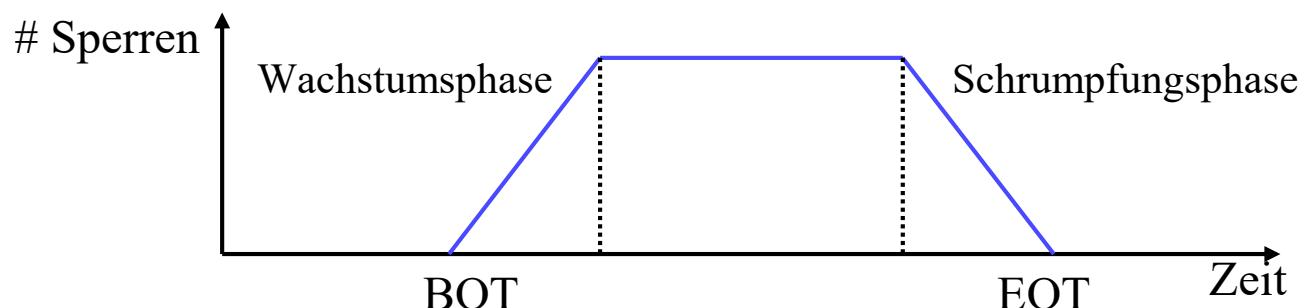
## Bemerkungen

- Anfordern und Freigeben von Sperren sollte das DBMS implizit selbst vornehmen.
- Die Verwendung legaler Schedules garantiert noch nicht die Serialisierbarkeit.

# Sperrverfahren (Locking)

## Zwei-Phasen-Sperrprotokoll (2PL)

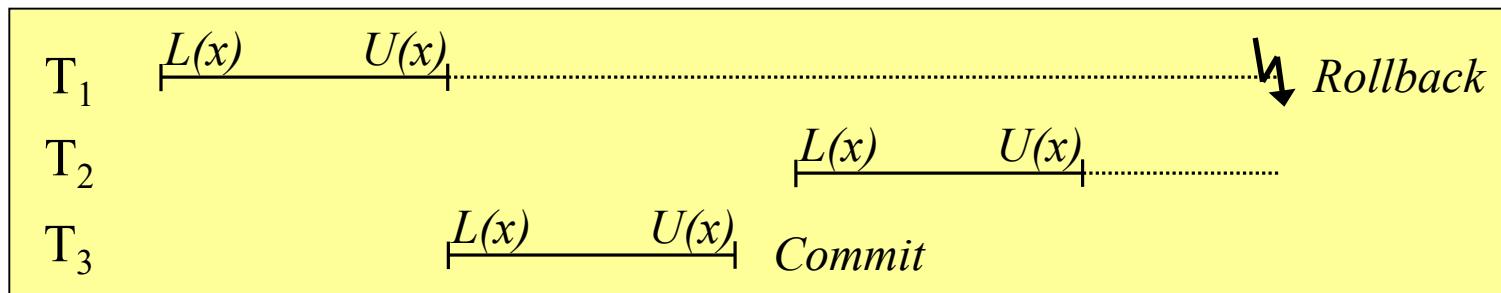
- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- Merkmal: keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
  - Wachstumsphase: Anforderungen der Sperren
  - Schrumpfungsphase: Freigabe der Sperren



# Sperrverfahren (Locking)

## Zwei-Phasen-Sperrprotokoll (2PL)

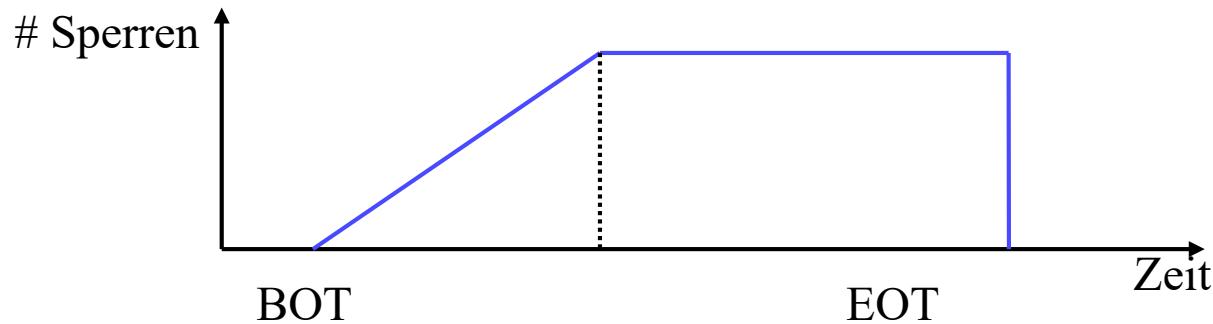
- Serialisierbarkeit ist gewährleistet, da Serialisierungsgraphen keine Zyklen enthalten können ☺
- Problem : Gefahr des kaskadierenden Rücksetzens im Fehlerfall (bzw. sogar **nicht-rücksetzbar**) ☹



- Transaktion T<sub>1</sub> wird nach  $U(x)$  zurückgesetzt
- T<sub>2</sub> hat “schmutzig” gelesen und muss zurückgesetzt werden
- Sogar T<sub>3</sub> muss zurückgesetzt werden  
→ Verstoß gegen die Dauerhaftigkeit (ACID) des COMMIT!

## Striktes Zwei-Phasen-Sperrprotokoll

- Abhilfe durch striktes (oder strenges) Zwei-Phasen-Sperrprotokoll:
  - Alle Sperren werden bis zum *COMMIT* gehalten
  - *COMMIT* wird atomar (d.h. nicht unterbrechbar) ausgeführt





# Sperrverfahren (Locking)

## Erhöhung des Parallelisierungsgrads

- Striktes 2PL erzwingt serialisierbare, rücksetzbare Schedules
- ABER: Parallelität der TAs wird dadurch stark eingeschränkt
  - Objekt ist entweder gesperrt (und dann bis zum Commit der entspr. TA) oder zur Bearbeitung frei  
=> kein paralleles Lesen oder Schreiben möglich
- Beobachtung: Parallelität unter Lesern könnte man eigentlich erlauben, da hier die Isoliertheit der beteiligten TAs nicht verletzt wird
- Daher statt 1 nun 2 Arten von Sperren
  - Lesesperren oder R-Sperren (read locks)
  - Schreibsperren oder X-Sperren (exclusive locks)



## RX-Sperrverfahren

- R- und X-Sperren
- Parallelität unter Lesern erlaubt
- Verträglichkeit der Sperrentypen  
(siehe Tabelle rechts)

|                     |   | bestehende Sperre |   |
|---------------------|---|-------------------|---|
|                     |   | R                 | X |
| angeforderte Sperre | R | +                 | - |
|                     | X | -                 | - |

## Serialisierungsreihenfolge bei RX

- RX-Sperrverfahren meist in Verbindung mit striktem 2PL um nur kaskadenfreie rücksetzbare Schedules zu erhalten
- Zur Erinnerung: Die Reihenfolge der Transaktionen im „äquivalenten seriellen Schedule“ ist die Serialisierungsreihenfolge.
- Bei RX-Sperrverfahren (in Verbindung mit striktem 2PL) wird die Serialisierungsreihenfolge durch die erste auftretende Konfliktoperation festgelegt.



# Sperrverfahren (Locking)

- Beispiel (Serialisierungsreihenfolge bei RX):
  - Situation:
    - $T_1$  schreibt ein Objekt  $x$
    - Danach möchte  $T_2$  Objekt  $x$  lesen
  - Folge:
    - $T_2$  muss auf das *COMMIT* von  $T_1$  warten, d.h. der serielle Schedule enthält  $T_1$  vor  $T_2$ .
    - Da  $T_2$  wartet, kommen auch alle weiteren Operationen erst nach dem *COMMIT* von  $T_1$ .
  - Achtung:  
Grundsätzlich sind zwar auch Abhängigkeiten von  $T_2$  nach  $T_1$  denkbar (z.B. auf einem Objekt  $y$ ), diese würden aber zu einer **Verklemmung (Deadlock)**, gegenseitiges Warten führen.

Skript zur Vorlesung:  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 10b

# Transaktionen - Datensicherheit

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani





# Aufgaben eines DBMS

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

## 1. Synchronisation (Concurrency Control)

Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

## 2. Datensicherheit (Recovery)

Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)

## 3. Integrität (Integrity)

Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer



## Fehler- und Recovery-Arten

- Transaktions-Recovery
  - **Transaktionsfehler:** Lokaler Fehler einer noch nicht festgeschriebenen TA, z.B. durch
    - Fehler im Anwendungsprogramm
    - Expliziter Abbruch der TA durch den Benutzer (`ROLLBACK`)
    - Verletzung von Integritätsbedingungen oder Zugriffsrechten
    - Rücksetzung aufgrund von Synchronisationskonflikten
  - Behandlung durch **Rücksetzen**
    - *Lokales UNDO:* der ursprüngliche DB-Zustand wie zu BOT wird wiederhergestellt, d.h. Rücksetzen aller Aktionen, die diese TA ausgeführt hat
    - Transaktionsfehler treten relativ häufig auf  
→ Behebung innerhalb von Millisekunden notwendig



## Fehler- und Recovery-Arten (cont.)

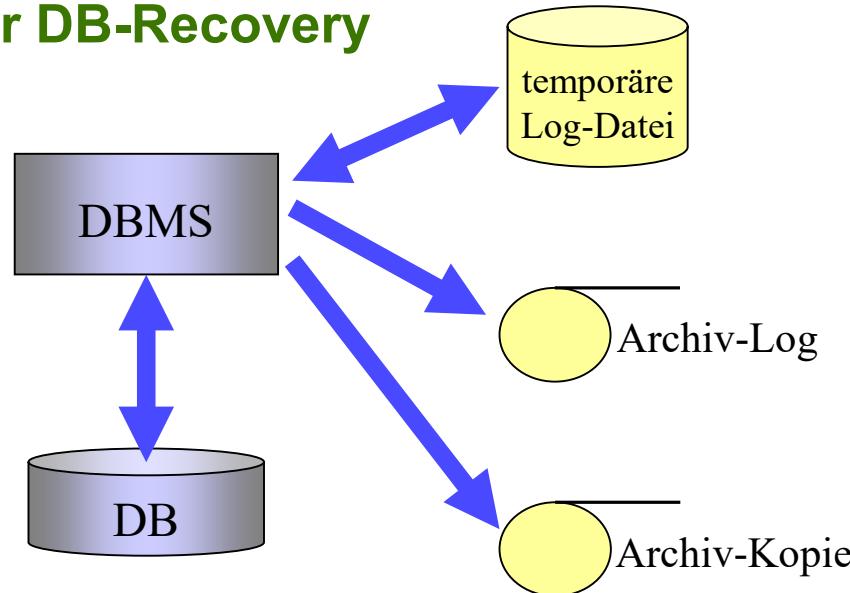
- Crash Recovery
  - **Systemfehler:** Fehler mit Hauptspeicherverlust, d.h. permanente Speicher sind *nicht* betroffen, z.B. durch
    - Stromausfall
    - Ausfall der CPU
    - Absturz des Betriebssystems, ...
  - Behandlung durch **Crash Recovery** (Warmstart)
    - *Globales UNDO*: Rücksetzen aller noch nicht abgeschlossenen TAs, die **bereits** in die DB eingebracht wurden
    - *Globales REDO*: Nachführen aller bereits abgeschlossenen TAs, die **noch nicht** in die DB eingebracht wurden
    - Systemfehler treten i.d.R. im Intervall von Tagen auf  
→ Recoverydauer einige Minuten



## Fehler- und Recovery-Arten (cont.)

- Geräte-Recovery
  - **Medienfehler:** Fehler mit Hintergrundspeicherverlust, d.h. Verlust von permanenten Daten, z.B. durch
    - Plattencrash
    - Brand, Wasserschaden, ...
    - Fehler in Systemprogrammen, die zu einem Datenverlust führen
  - Behandlung durch **Geräte-Recovery** (Kaltstart)
    - Aufsetzen auf einem früheren, gesicherten DB-Zustand (Archivkopie)
    - *Globales REDO:* Nachführen aller TAs, 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

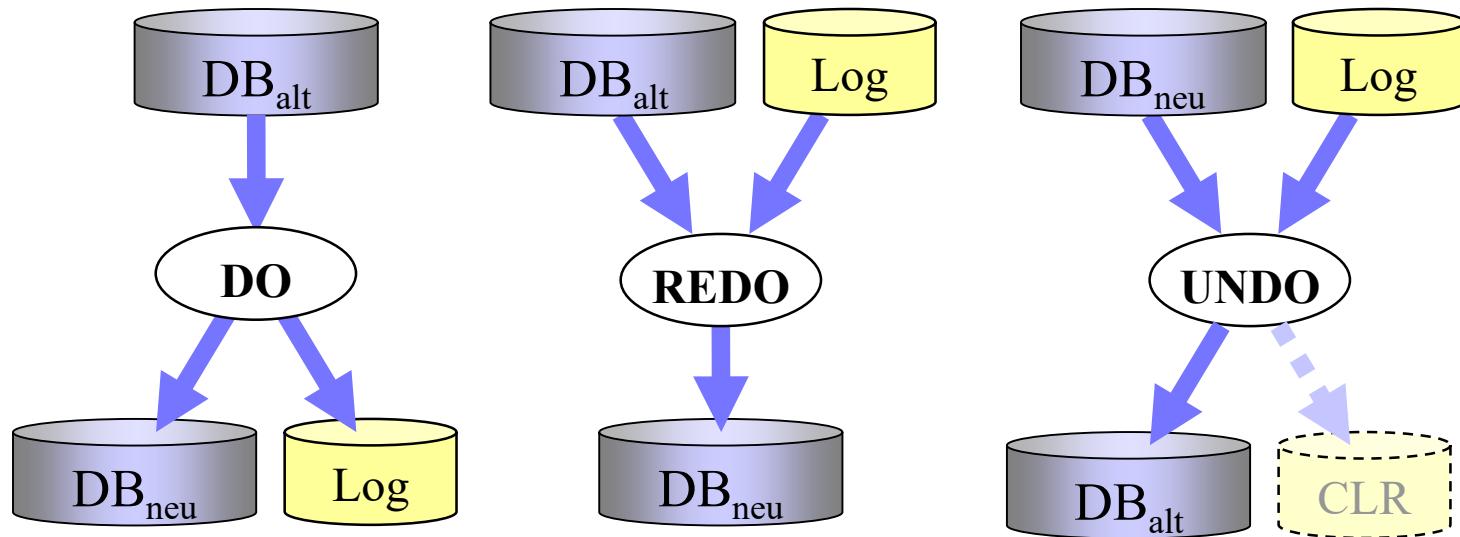
## Systemkomponenten der DB-Recovery



- Behandlung von Transaktions- und Systemfehlern  
$$\text{DB} + \text{temporäre Log-Datei} \rightarrow \text{DB}$$
- Behandlung von Medienfehlern  
$$\text{Archiv-Kopie} + \text{Archiv-Log} \rightarrow \text{DB}$$

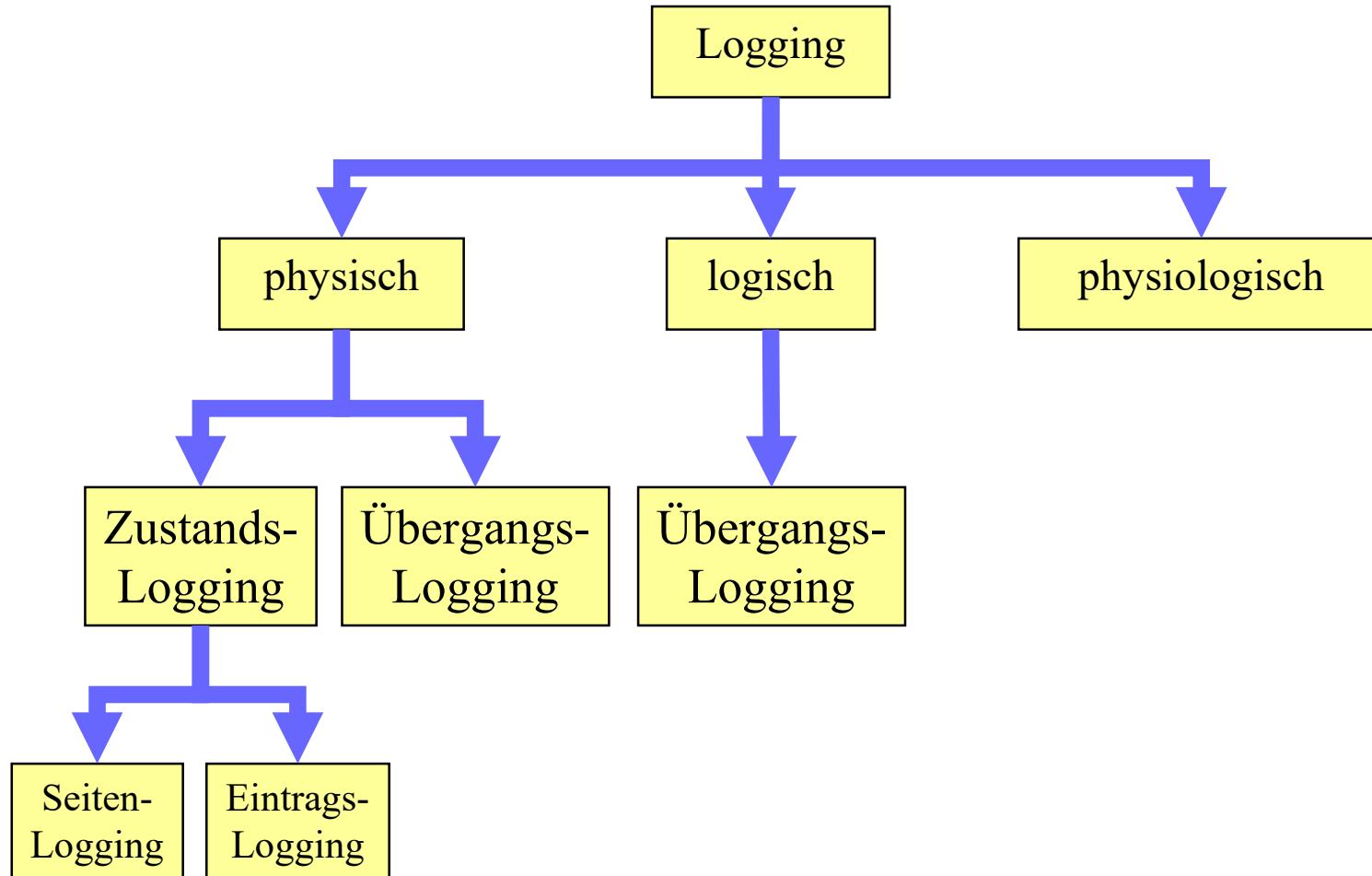
## Aufgaben des Logging

- Für jede Änderungsoperation auf der Datenbank im Normalbetrieb (**DO**) benötigt man Protokolleinträge für
  - **REDO**: Information zum Nachvollziehen der Änderungen erfolgreicher TAs
  - **UNDO**: Information zum Zurücknehmen der Änderungen unvollständiger TAs



CLR = Compensation Log Record (zur Behandlung von Fehlern während der Recovery)

## Klassifikation von Logging-Verfahren





## Physisches Logging

- Protokoll auf der Ebene der physischen Objekte (Seiten, Datensätze, Indexeinträge)
- Zustandslogging
  - Protokollierung der Werte vor und nach jeder Änderung: Alte Zustände *BFIM* (Before-Images) und neue Zustände *AFIM* (After-Images) der geänderten Objekte werden in die Log-Datei geschrieben
- Übergangslogging
  - Protokollierung der Zustandsdifferenz zwischen BFIM und AFIM



# Logging-Techniken

- Zustandslogging auf Seitenebene
  - vollständige Kopien von Seiten werden protokolliert
  - Recovery sehr einfach und schnell (Seiten einfach zurückkopieren)
  - sehr großer Logumfang und hohe I/O-Kosten auch bei nur kleinen Änderungen
  - Seitenlogging impliziert Seitensperren → hohe Konfliktrate bei Synchronisation
- Zustandslogging auf Eintragsebene
  - statt ganzer Seiten werden nur tatsächlich geänderte Einträge protokolliert
  - kleinere Sperrgranulate als Seiten möglich
  - Protokollgröße reduziert sich typischerweise um mind. 1 Größenordnung
  - Log-Einträge werden in Puffer gesammelt → wesentlich weniger Plattenzugriffe
  - Recovery ist aufwändiger: zu ändernde Datenbankseiten müssen vollständig in den Hauptspeicher geladen werden, um die Log-Einträge anwenden zu können



# Logging-Techniken

- Übergangslogging
  - Protokollierung der Zustandsdifferenz zwischen  $BFIM$  und  $AFIM$
  - Aus  $BFIM$  muss  $AFIM$  berechenbar sein (u.u.)
  - Realisierbar durch XOR-Operation  $\oplus$  (eXclusive-OR)<sup>1</sup>:

|   | Zustands-Logging  | Übergangs-Logging                                   |
|---|---|---|
| <b>DO</b><br>Änderung $A_{alt} \rightarrow A_{neu}$ | Protokollierung von<br>$BFIM = A_{alt}, AFIM = A_{neu}$ | Protokollierung von<br>$D = A_{alt} \oplus A_{neu}$ |
| <b>REDO</b> (in DB liegt $A_{alt}$ )                | Überschreibe $A_{alt}$ mit $AFIM$                       | $A_{neu} = A_{alt} \oplus D$                        |
| <b>UNDO</b> (in DB liegt $A_{neu}$ )                | Überschreibe $A_{neu}$ mit $BFIM$                       | $A_{alt} = A_{neu} \oplus D$                        |

<sup>1</sup> XOR-Operation:

$$\begin{aligned}XOR: \\ 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0\end{aligned}$$



## Logisches Logging

- Spezielle Form des Übergangs-Logging: Protokollierung von Änderungsoperationen mit ihren aktuellen Parametern
- Protokoll auf hoher Abstraktionsebene ermöglicht kurze Log-Einträge
- Probleme für **REDO**: Änderungen umfassen typischerweise mehrere Seiten (Tabelle, Indexe)
  - Atomares Einbringen der Mehrfachänderungen schwierig
  - Logische Änderungen sind aufwändiger durchzuführen als physische Änderungen
- Probleme für **UNDO**: Mengenorientierte Änderungen können sehr aufwändige Protokolleinträge verursachen:
  - Bsp.: `DELETE FROM Products WHERE Group = 'G1'`  
=> **UNDO** erfordert viele Einfügungen, falls Produktgruppe G1 umfangreich ist
  - Bsp.: `UPDATE Products SET Group = 'G2' WHERE Group = 'G1'`  
=> **UNDO** muss alte und neue Produkte der Gruppe G2 unterscheiden



## Physiologisches Logging

- Kombination von physischem und logischem Logging:  
Protokollierung von elementaren Operationen innerhalb einer Seite
  - Physical-to-a-page
    - Protokollierungseinheiten sind geänderte Seiten
    - gut verträglich mit Pufferverwaltung und direktem (atomarem) Einbringen
  - Logical-within-a-page
    - logische Protokollierung der Änderungen auf einer Seite
- Bewertung
  - Log-Einträge beziehen sich nicht auf mehrere Seiten wie bei logischem Logging
  - Dadurch einfachere Recovery als bei logischem Logging
  - Log-Datei ist länger als bei logischem Logging aber kürzer als bei physischem Logging
  - Flexibler als physisches Logging wegen variabler Objektpositionen auf Seiten.



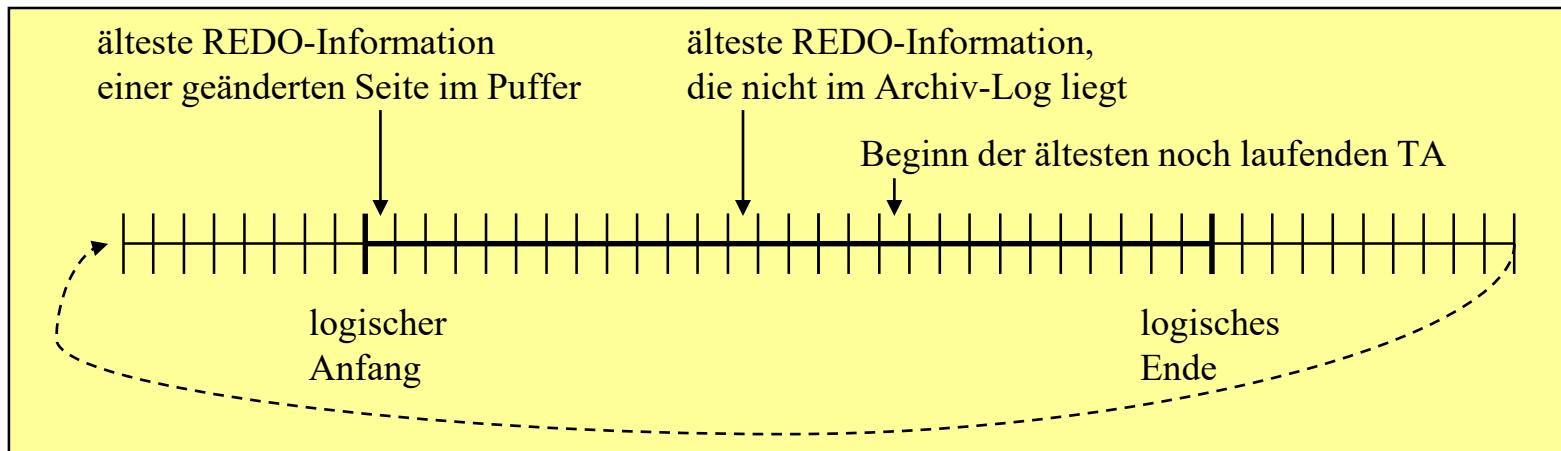
## Die Log-Datei

- Art der Protokolleinträge
  - Beginn, Commit und Rollback von Transaktionen
  - Änderungen des DB-Zustandes durch Transaktionen
  - Sicherungspunkte (Checkpoints)
- Komponenten von Änderungseinträgen: (LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)
  - LSN (Log Sequence Number): eindeutige Kennung des Log-Eintrags in chronologischer Reihenfolge
  - TA-Id: eindeutige Kennung der TA, die die Änderung durchgeführt hat
  - Page-Id : Kennung der Seite auf der die Änderungsoperation vollzogen wurde (ein Eintrag pro geänderter Seite)
  - REDO: gibt an, wie die Änderung nachvollzogen werden kann
  - UNDO: beschreibt, wie die Änderung rückgängig gemacht werden kann
  - PrevLSN: Zeiger auf vorhergehenden Log-Eintrag der jeweiligen TA (Effizienzgründe)



# Logging-Techniken

- Die Log-Datei ist eine **sequentielle** Datei: Schreiben neuer Protokolldaten an das aktuelle Dateiende  
=> Ringpuffer-Organisation
  - Log-Daten sind für **Crash-Recovery** nur begrenzte Zeit relevant:
    - UNDO-Sätze für erfolgreich beendete TAs werden nicht mehr benötigt
    - Nach Einbringen der Seite in die DB wird REDO-Information nicht mehr benötigt



- REDO-Information für Geräte-Recovery** ist im Archiv-Log zu sammeln



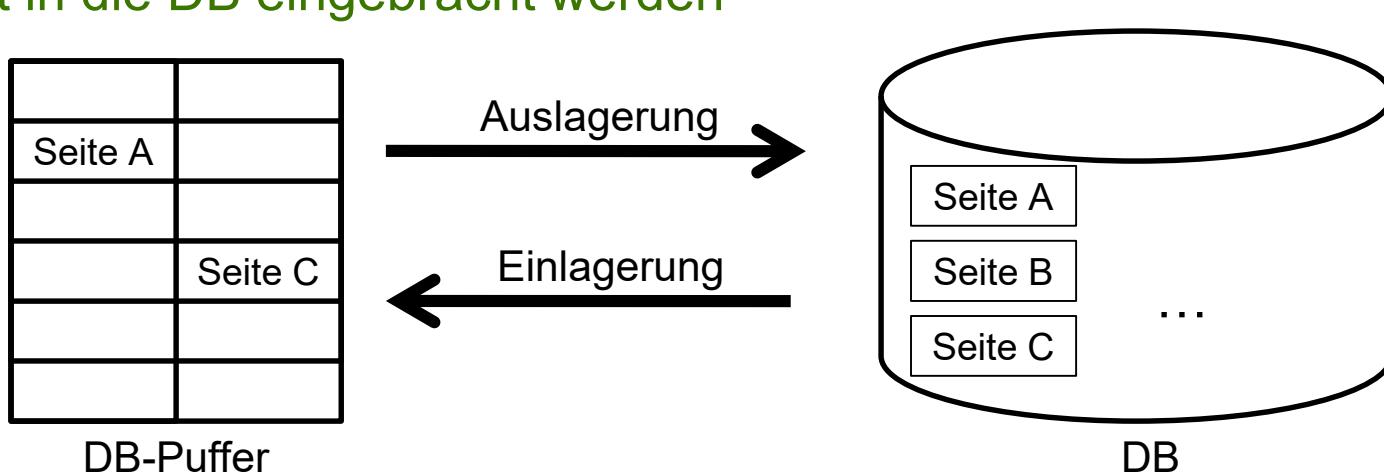
- Beispiel

| Ablauf T <sub>1</sub>   | Ablauf T <sub>2</sub>  | Log-Eintrag<br>(LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)   |
|---|--|---|
| <b>begin</b><br>read(A, a <sub>1</sub> )  |  | (#1, T <sub>1</sub> , begin, 0)   |
| a <sub>1</sub> := a <sub>1</sub> - 50<br><b>write</b> (A, a <sub>1</sub> )                                  | <b>begin</b><br>read(C, c <sub>2</sub> ) //80  | (#2, T <sub>2</sub> , begin, 0)   |
|   | c <sub>2</sub> := 100<br><b>write</b> (C, c <sub>2</sub> )   | (#3, T <sub>1</sub> , p <sub>A</sub> , A-=50, A+=50, #1)<br>(#4, T <sub>2</sub> , p <sub>C</sub> , C=100, C=80, #2) |
| read(B, b <sub>1</sub> ) //70<br>b <sub>1</sub> := 50<br><b>write</b> (B, b <sub>1</sub> )<br><b>commit</b> |  | (#5, T <sub>1</sub> , p <sub>B</sub> , B=50, B=70, #3)<br>(#6, T <sub>1</sub> , commit, #5)                         |
|   | read(A, a <sub>2</sub> )<br>a <sub>2</sub> := a <sub>2</sub> - 100<br><b>write</b> (A, a <sub>2</sub> )<br><b>commit</b> | (#7, T <sub>2</sub> , p <sub>A</sub> , A-=100, A+=100, #4)<br>(#8, T <sub>2</sub> , commit, #7)                     |

(hier: logisches Logging)

## Die Speicherhierarchie

- i.d.R. besteht die Speicherhierarchie bei DBMS aus zwei Ebenen
  - DBMS-Puffer (Hauptspeicher) [kurz: DB-Puffer]
  - DB (Hintergrundspeicher)
- Im laufenden Betrieb werden die Operationen der einzelnen TAs im DB-Puffer ausgeführt
- Die DB muss gemäß dem ACID-Prinzip transaktionskonsistent gehalten werden, d.h. Änderungen durch TAs müssen nach dem Commit in die DB eingebracht werden



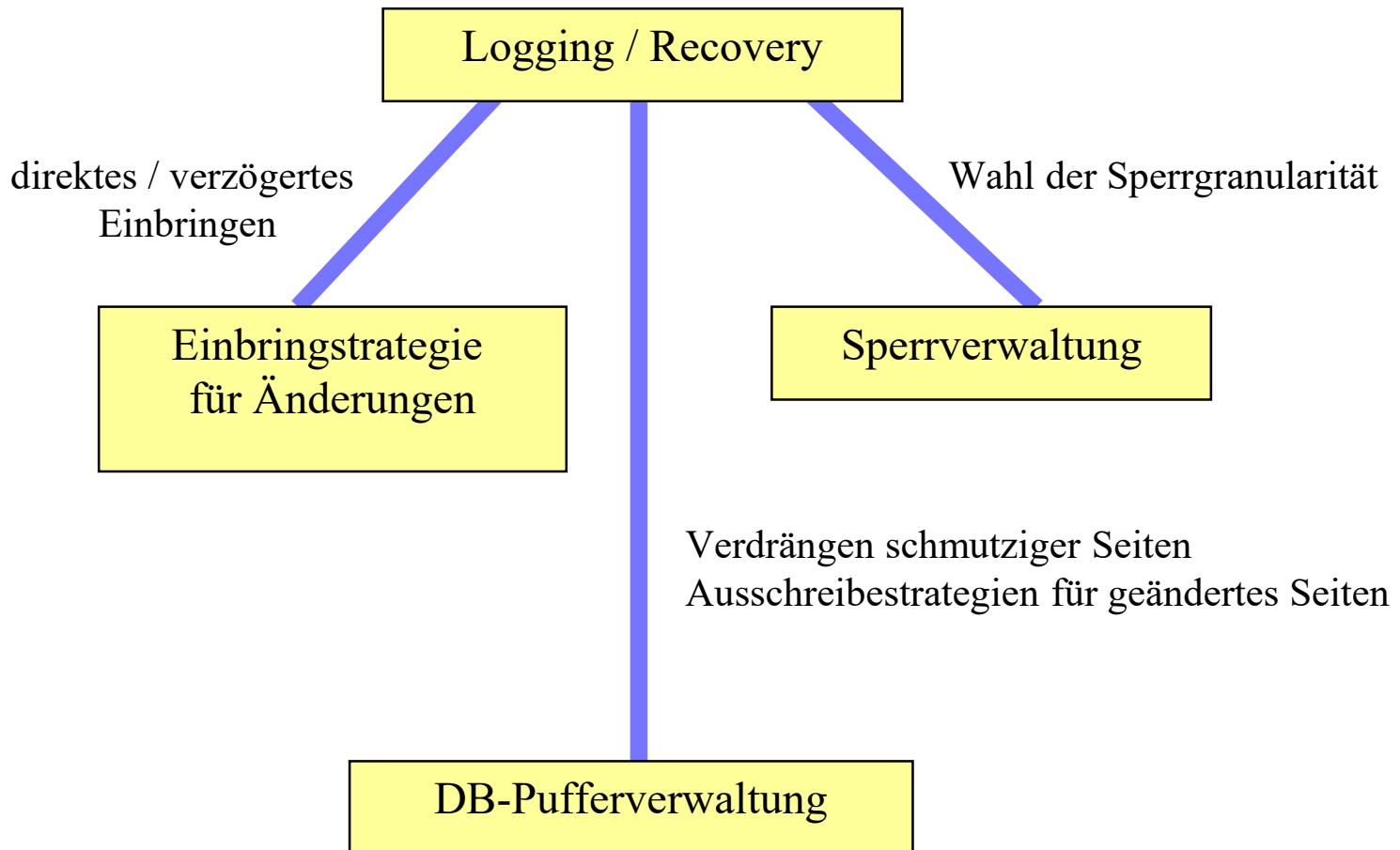


## Abhangigkeiten

- Aus der zweistufigen Speicherhierarchie ergeben sich insbesondere Abhangigkeiten der Recovery/des Loggings zur Speicherverwaltung
  - DB-Puffer ist begrenzt => was passiert, wenn Puffer voll?  
=> Pufferverwaltung (**Verdrangungsstrategien**)
  - Wann schreibe ich anderungen in die Datenbank?  
=> Pufferverwaltung (**Ausschreibestrategien**)
  - Wie schreibe ich die anderungen aus?  
=> HGS-Verwaltung (**Einbringungsstrategien**)  
=> Abhangig davon: wann schreibe ich Log-Datei auf Platte?
- Wie wir sehen werden, besteht zudem eine Abhangigkeit der Recovery/des Loggings zur Sperrverwaltung (bei pessimistischer Synchronisation)



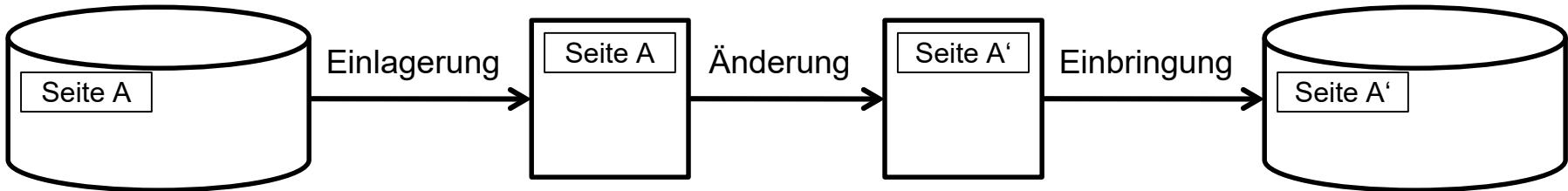
## Schematischer Überblick der Abhängigkeiten





## Einbringungstrategie: Direktes Einbringen (NonAtomic, Update-in-Place)

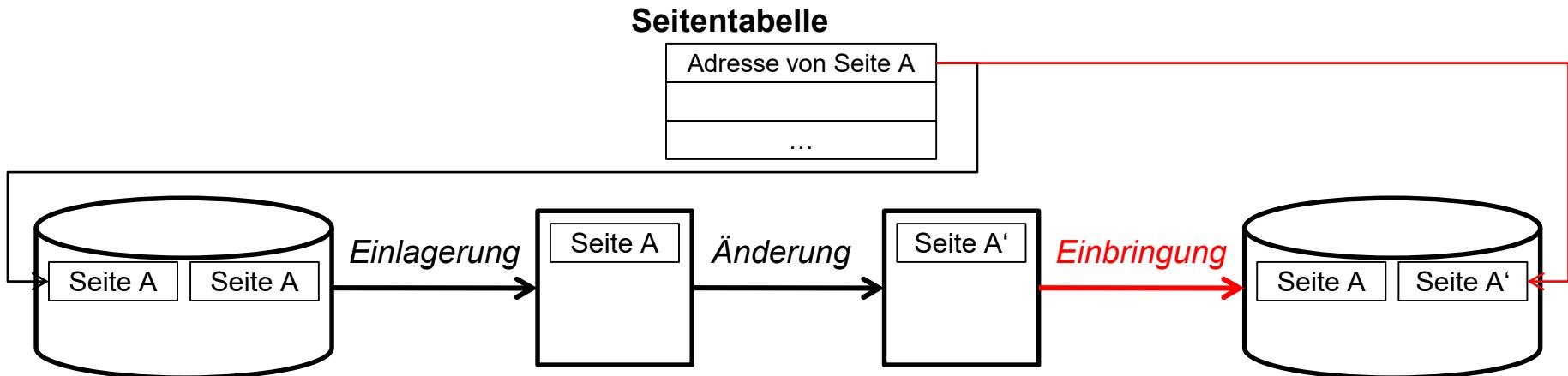
- Jede Seite hat eine Speicheradresse auf der Platte
- Geänderte Seiten werden immer auf ihren Block zurück geschrieben, d.h. der alte Inhalt der Seite in der DB wird dabei überschrieben
- Ausschreiben einer Seite ist dadurch gleichzeitig Einbringen in die permanente DB (es gibt keinen Zwischenspeicher für Seiten)
- Es ist nicht möglich mehrere Seiten atomar einzubringen, d.h. Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (daher: **NonAtomic**)
- Dies ist die gängigste Methode in heutigen DBMS
- **UNDO**-Informationen müssen explizit gespeichert werden





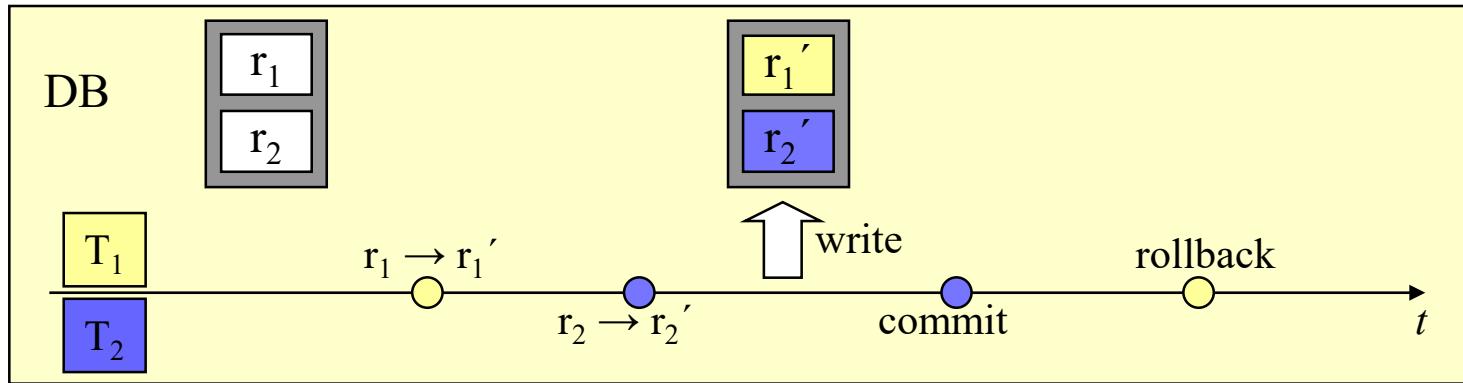
## Einbringungstrategie: Indirektes Einbringen (Atomic)

- Geänderte Seite wird in separaten Block auf Platte geschrieben
  - Twin-Block-Verfahren: jede Seite hat zwei Blöcke auf der Platte
  - Schattenspeichertechnik: nur modifizierte Seiten haben zwei Blöcke
- Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich (daher: **Atomic**)
- Alte Versionen der Objekte bleiben erhalten, d.h. es muss keine **UNDO**-Information explizit gespeichert werden



## Einfluss der Sperrgranularität

- Log-Granularität muss kleiner oder gleich der Sperrgranularität sein, sonst Lost Updates möglich
- D.h. Satzsperren erzwingen feine Log-Granulate
- Beispiel für Problem bei “Satzsperren mit Seitenlogging”



- $T_1, T_2$  ändern die Datensätze  $r_1, r_2$ , die auf derselben DB-Seite liegen
- Die Seite wird in die DB zurück geschrieben,  $T_2$  endet mit COMMIT
- Falls  $T_1$  zurückgesetzt wird, geht auch die Änderung  $r_2 \rightarrow r_2'$  verloren
- Lost Update, d.h. Verstoß gegen die Dauerhaftigkeit des COMMIT



## Pufferverwaltung: Verdrängungsstrategien

- Ersetzung schmutziger Seiten im Puffer (Wohin werden Seiten verdrängt? Warum nur schmutzige Seiten?)  
Seite ist schmutzig wenn: SeitePuffer  $\neq$  SeiteDB
  - **No-Steal**
    - Schmutzige Seiten dürfen nicht aus dem Puffer entfernt werden
    - DB enthält keine Änderungen nicht-erfolgreicher TAs
    - **UNDO-Recovery** ist nicht erforderlich
    - Probleme bei langen Änderungs-TAs, da große Teile des Puffers blockiert werden => Einschränkung der Parallelität
  - **Steal**
    - Schmutzige Seiten dürfen jederzeit ersetzt und in die DB eingebracht werden
    - DB kann unbestätigte Änderungen enthalten
    - **UNDO-Recovery** ist erforderlich
    - effektivere Puffernutzung bei langen TAs mit vielen Änderungen



## Pufferverwaltung: Ausschreibestrategien (EOT-Behandlung)

- Wann werden Änderungen in die DB eingebracht?
  - **Force**
    - Alle geänderte Seiten werden spätestens bei EOT (vor COMMIT) in die DB geschrieben
    - keine **REDO**-Recovery erforderlich bei Systemfehler
    - hoher I/O-Aufwand, da Änderungen jeder TA einzeln geschrieben werden
    - Vielzahl an Schreibvorgängen führt zu schlechteren Antwortzeiten, länger gehaltenen Sperren und damit zu mehr Sperrkonflikten
    - Große DB-Puffer werden schlecht genutzt
  - **No-Force**
    - Änderungen können auch erst nach dem COMMIT in die DB geschrieben werden
    - Beim COMMIT werden lediglich **REDO**-Informationen in die Log-Datei geschrieben
    - **REDO**-Recovery erforderlich bei Systemfehler
    - Änderungen auf einer Seite von mehreren TAs können gesammelt werden



## Kombination:

|          | No-Steal   | Steal                          |
|----------|--|--------------------------------|
| Force    | kein <i>UNDO</i> – kein <i>REDO</i><br>(nicht für <i>Update-in-Place</i> ) | <i>UNDO</i> – kein <i>REDO</i> |
| No-Force | kein <i>UNDO-REDO</i>  | <i>UNDO-REDO</i>               |

- Bewertung **Steal / No-Force**
  - erfordert zwar **UNDO** als auch **REDO**, ist aber allgemeinste Lösung
  - beste Leistungsmerkmale im Normalbetrieb
- Bewertung **No-Steal / Force**
  - optimiert den Fehlerfall auf Kosten des Normalfalls (sehr teures COMMIT)
  - für *Update-in-Place* nicht durchführbar:
    - wegen **No-Steal** dürfen Änderungen erst nach COMMIT in die DB gelangen, was jedoch **Force** widerspricht (**No-Steal** → **No-Force**)
    - wegen Force müssten Änderungen vor dem COMMIT in der DB stehen, was bei *Update-in-Place* unterbrochen werden kann, **UNDO** wäre nötig (**Force** → **Steal**)



## WAL-Prinzip und COMMIT-Regel

- WAL-Prinzip (**Write-Ahead-Log**)
  - UNDO-Information (z.B. BFIM) muss vor Änderung der DB im Protokoll stehen
  - Wichtig, um schmutzige Änderungen rückgängig zu machen
  - Nur relevant für *Steal*
  - Wichtig bei direktem Einbringen
- COMMIT-Regel (**Force-Log-at-Commit**)
  - REDO-Information (z.B. AFIM) muss vor dem COMMIT im Protokoll stehen
  - Voraussetzung für Crash-Recovery bei *No-Force*
  - Erforderlich für Geräte-Recovery (auch bei *Force*)
  - Gilt für direkte und indirekte Einbringstrategien gleichermaßen
- Bemerkung: Um die chronologische Reihenfolge im Ringpuffer zu wahren, werden alle Log-Einträge bis zum letzten notwendigen ausgeschrieben, d.h. es werden keine Log-Einträge übergangen



## COMMIT-Verarbeitung

- **Standard Zwei-Phasen-Commit**

- *Phase 1: Logging*

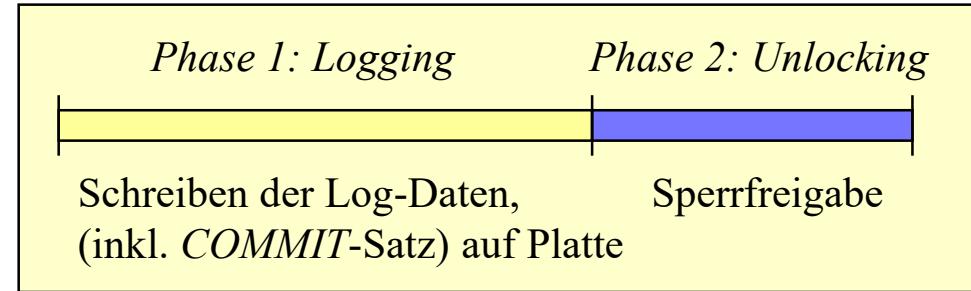
- Überprüfen der verzögerten Integritätsbedingungen
    - Logging der **REDO**-Informationen incl. COMMIT-Satzes

- *Phase 2: Unlocking*

- Freigabe der Sperren (Sichtbarmachen der Änderungen)
    - Bestätigung des COMMIT an das Anwendungsprogramm

- Problem

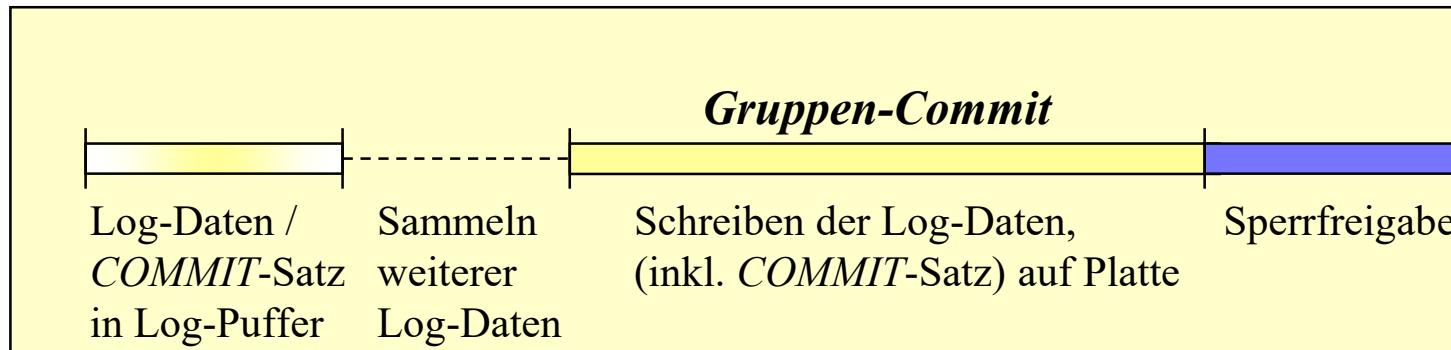
- COMMIT-Regel verlangt Ausschreiben des Log-Puffers bei jedem COMMIT
    - Beeinträchtigung für kurze TAs, deren Log-Daten weniger als eine Seite umfassen
    - Durchsatz an TAs ist eingeschränkt





- ***Gruppen-Commit***

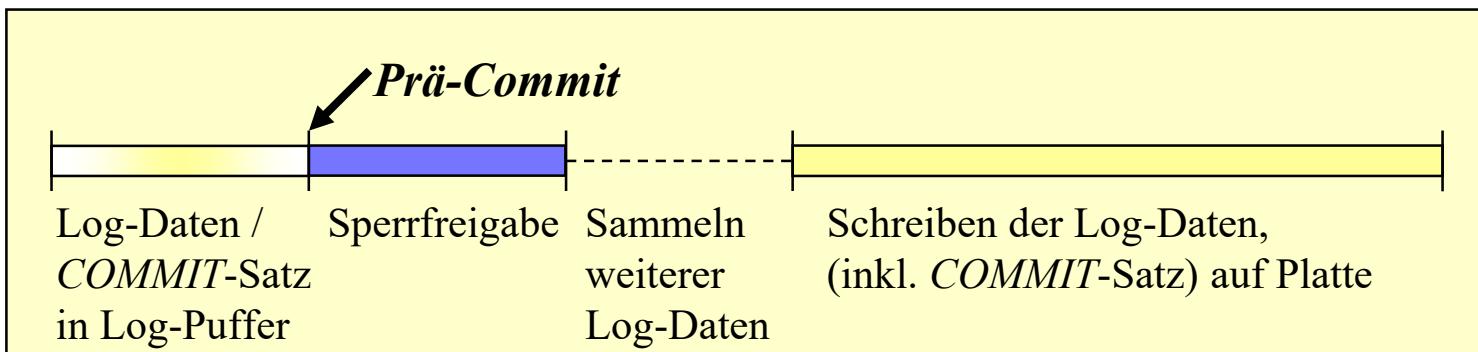
- Log-Daten mehrerer TAs werden im Puffer gesammelt
- Log-Puffer wird auf Platte geschrieben, sobald Puffer gefüllt ist oder nach Timeout
- Vorteil: Reduktion der Plattenzugriffe und höhere Transaktions-raten möglich
- Nachteil: längere Sperrdauer führt zu längeren Antwortzeiten
- In der Praxis: wird von zahlreichen DBS unterstützt





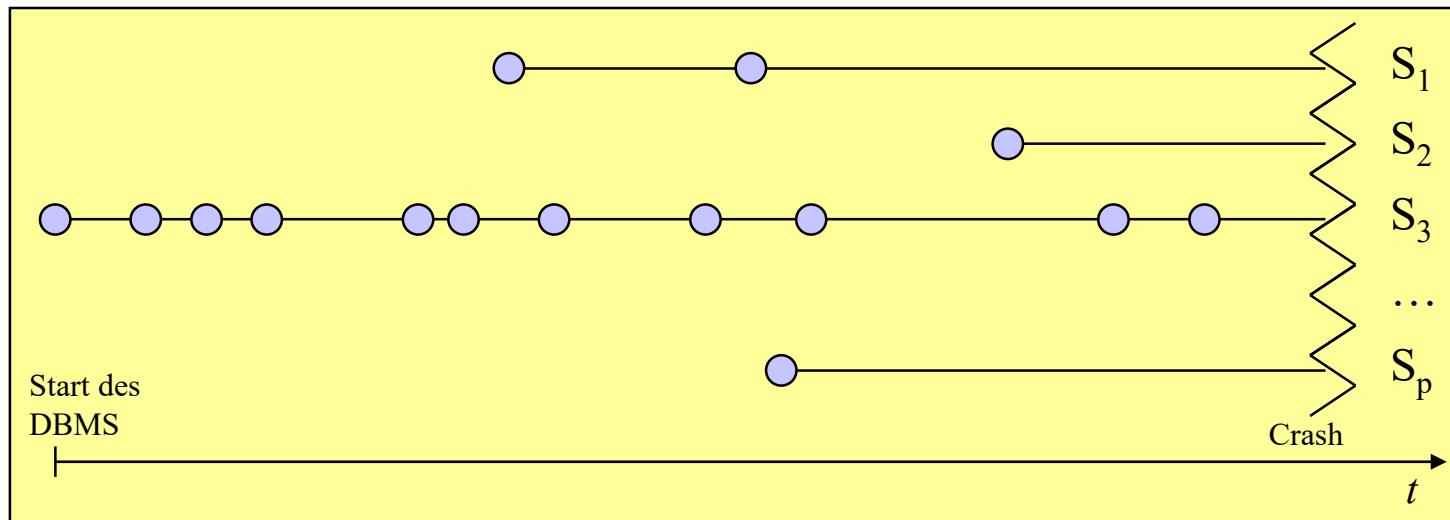
- **Prä-Commit**

- Vermeidung der langen Sperrzeiten des Gruppen-Commit indem Sperren bereits freigegeben werden, wenn COMMIT-Satz im Log-Puffer steht
- Ist Prä-Commit zulässig?
- Normalfall: ändernde TA kommt erfolgreich zu Ende, Änderungen sind gültig
- Fehlerfall: Abbruch der TA nur noch durch Systemfehler möglich; bei Systemfehler werden auch die anderen laufenden TAs abgebrochen, "schmutziges Lesen" kann sich also nicht auf DB auswirken



# Sicherungspunkte

- Sicherungspunkte sind eine Maßnahme zur Begrenzung des **REDO**-Aufwands nach Systemfehlern
- Ohne Sicherungspunkte müssten potentiell alle Änderungen seit Start des DBMS wiederholt werden
- Besonders kritisch: Hot-Spot-Seiten, die (fast) nie aus dem Puffer verdrängt werden





# Sicherungspunkte

- Durchführung von Sicherungspunkten
  - Spezielle Log-Einträge:  
    BEGIN\_CHKPT  
    Info über laufende TAs  
    END\_CHKPT
  - *LSN* des letzten vollständig ausgeführten Sicherungspunktes wird in Restart-Datei geführt
- Häufigkeit von Sicherungspunkten
  - **zu selten**: hoher **REDO**-Aufwand
  - **zu oft**: hoher Overhead im Normalbetrieb
  - z.B. Sicherungspunkte nach bestimmter Anzahl von Log-Sätzen einfügen



## Direkte Sicherungspunkte

- Charakterisierung
  - Alle geänderten Seiten werden am Sicherungspunkt in die persistente DB (Platte) geschrieben
  - Zeitbedarf steigt mit dem zeitlichen Abstand der Sicherungspunkte
  - Multi-Page-Access hilft, Schreibkopf-Positionierungen zu minimieren
  - **REDO-Recovery** kann beim letzten vollständig ausgeführten Checkpoint beginnen
- 3 Arten
  - Transaktions-orientierte Sicherungspunkte (**TOC**)
  - Transaktions-konsistente Sicherungspunkte (**TCC**)
  - Aktions-konsistente Sicherungspunkte (**ACC**)

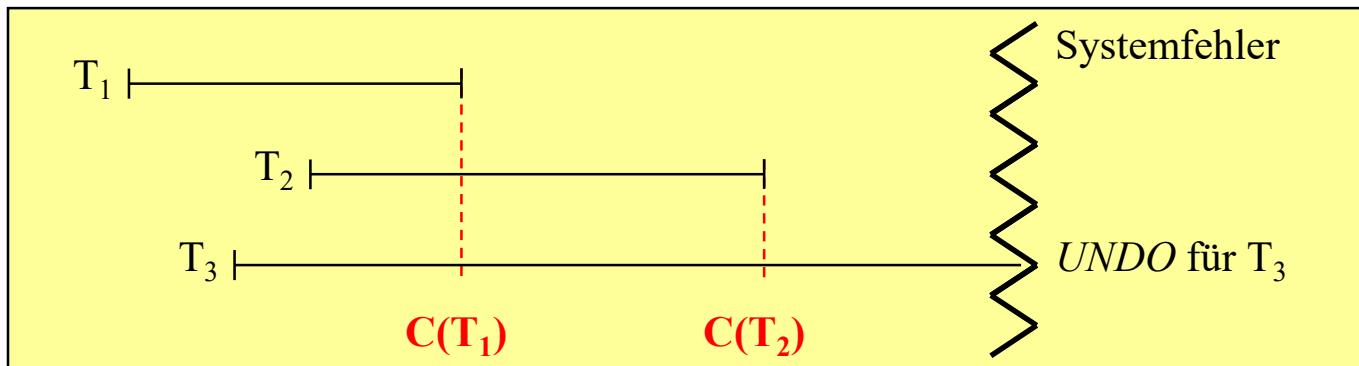


## TOC: TA-orientierte Sicherungspunkte

- TOC entspricht *Force*, d.h. Ausschreiben aller Änderungen beim COMMIT oder anders gesagt: jedes COMMIT definiert einen Sicherungspunkt
- Nicht alle Seiten im Puffer werden geschrieben, sondern nur Änderungen der jeweiligen TA
- Sicherungspunkt bezieht sich immer auf genau eine TA (die den Sicherungspunkt mit COMMIT ausgelöst hat)
- **UNDO-Recovery**  
Bei *Update-in-Place* ist **UNDO** nötig (*Force* → *Steal*),  
**UNDO** beginnt dann beim letzten Sicherungspunkt
- **REDO-Recovery** garnicht nötig (*Force!*)

# Sicherungspunkte

- Vorteile:
  - keine **REDO** nötig
  - Implementierung ist einfach in Kombination mit Seitensperren
- Nachteil: (sehr) aufwändiger Normalbetrieb, insbesondere für Hot-Spot-Seiten
- Beispiel: Sicherungspunkte bei COMMIT von T1 und T2, deshalb kein **REDO** nötig





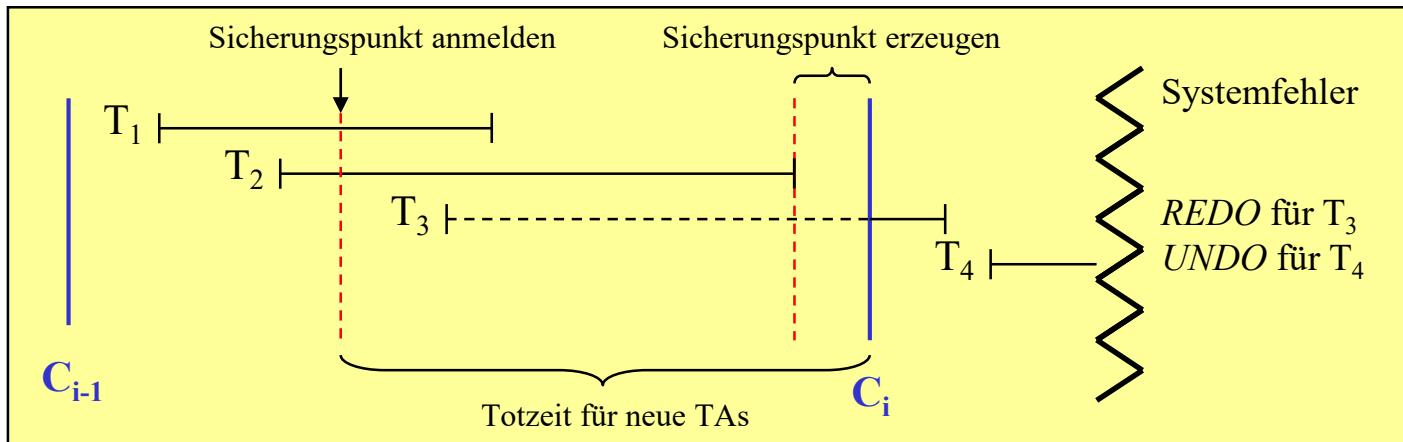
# Sicherungspunkte

## TCC: TA-konsistente Sicherungspunkte

- DB wird in TA-konsistenten Zustand gebracht, d.h. keine schmutzigen Änderungen
- Während Sicherung keine aktiven Änderungs-TAs
- Sicherungspunkt bezieht sich immer auf alle TAs
- **UNDO-** und **REDO-Recovery** sind durch letzten Sicherungspunkt begrenzt
- Ablauf:
  - Anmeldung des Sicherungspunktes
  - Warten, bis alle Änderungs-TAs abgeschlossen sind
  - Erzeugen des Sicherungspunktes
  - Verzögerung neuer Änderungs-TAs bis zum Abschluss der Sicherung

# Sicherungspunkte

- Vorteil: **UNDO-** und **REDO-Recovery** beginnen beim letzten Sicherungspunkt (im Beispiel:  $C_i$ ), d.h. es sind nur TAs betroffen, die nach der letzten Sicherung gestartet wurden (hier:  $T_3, T_4$ )
- Nachteil: lange Wartezeiten (“Totzeiten”) im System
- Beispiel:



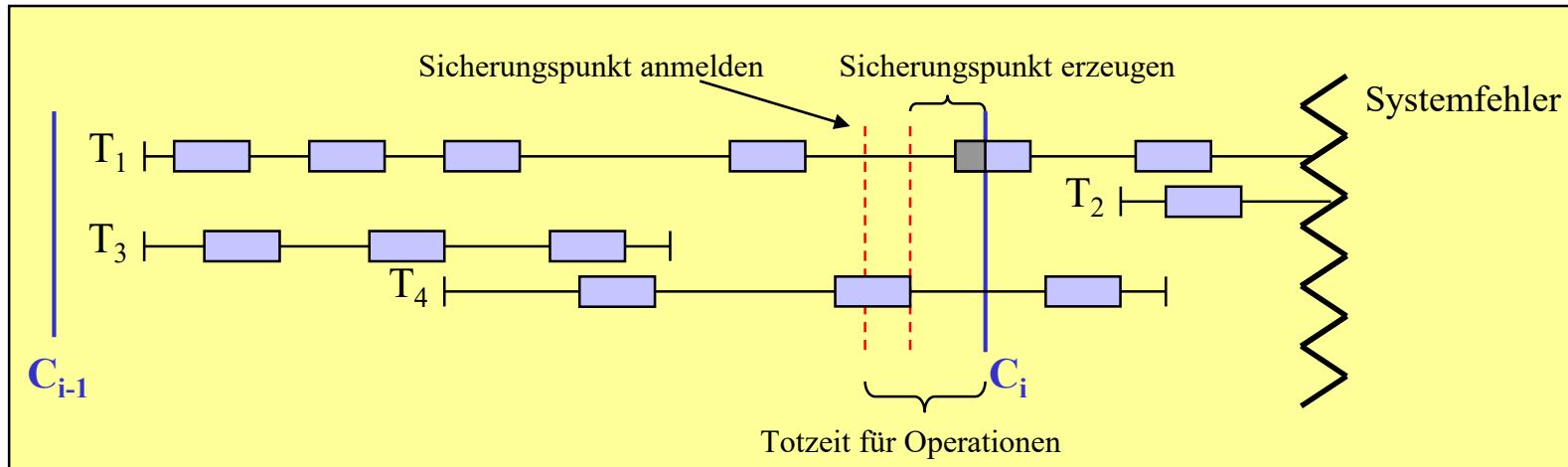


## ACC: Aktions-konsistente Sicherungspunkte

- Blockierung nur auf Operationenebene, nicht mehr für ganze TAs
- Keine Änderungsoperationen während der Sicherung
- **UNDO-Recovery** beginnt bei  $\text{MinLSN}$  = kleinste  $LSN$  aller noch aktiven TAs des letzten Sicherungspunktes
- **REDO-Recovery** durch letzten SP begrenzt
- Ablauf:
  - Anmelden des Sicherungspunktes
  - Beendigung aller laufenden Änderungsoperationen abwarten
  - Erzeugen des Sicherungspunktes
  - Verzögerung neuer Änderungsoperationen bis zum Abschluss der Sicherung

# Sicherungspunkte

- Vorteil: Totzeit des Systems für Änderungen deutlich reduziert
- Nachteil: Geringere Qualität der Sicherungspunkte
  - schmutzige Änderungen können in die Datenbank gelangen
  - zwar **REDO**-, nicht jedoch **UNDO**-Recovery durch letzten Sicherungspunkt begrenzt
- Beispiel:





## Indirekte Sicherungspunkte

- Charakterisierung
  - Direkte Sicherungspunkte: hoher Aufwand bei großen DB-Puffern nicht akzeptabel
  - Indirekte Sicherungspunkte: Änderungen werden nicht vollständig ausgeschrieben
  - DB hat keinen Aktions- oder TA-konsistenten Zustand, sondern unscharfen (fuzzy) Zustand
- Erzeugung eines indirekten Sicherungspunktes
  - im wesentlichen Logging des Status von laufenden TAs und geänderten Seiten
  - minimaler Schreibaufwand, keine nennenswerte Unterbrechung des Betriebs

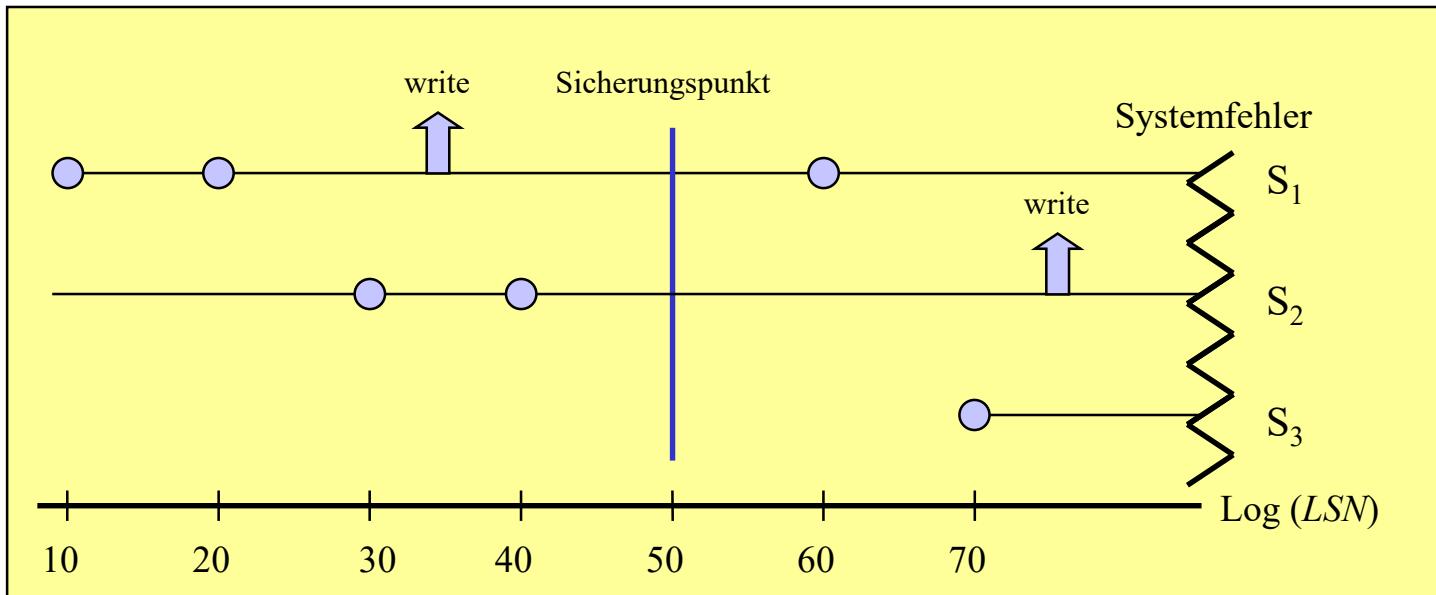


# Sicherungspunkte

- Ausschreiben von DB-Änderungen
  - außerhalb der Sicherungspunkte, asynchron zur laufenden TA-Verarbeitung
  - länger nicht mehr referenzierte Seiten werden vorausschauend ausgeschrieben
  - Sonderbehandlung für Hot-Spot-Seiten nötig:
    - zwangsweises Ausschreiben bei bestimmtem Log-Umfang
    - Anlegen einer Kopie, um keine Verzögerung für neue Änderungen zu verursachen
- **UNDO-Recovery** beginnt bei *MinLSN* (siehe ACC)
- **REDO-Recovery**
  - Startpunkt ist nicht mehr durch letzten Sicherungspunkt gegeben, auch weiter zurückliegende Änderungen müssen ggf. wiederholt werden
  - Zu jeder geänderten Seite wird *StartLSN* vermerkt (*LSN* der 1. Änderung seit Einlesen von Platte)
  - **REDO** beginnt bei *MinDirtyPageLSN* = min (*StartLSN*)

# Sicherungspunkte

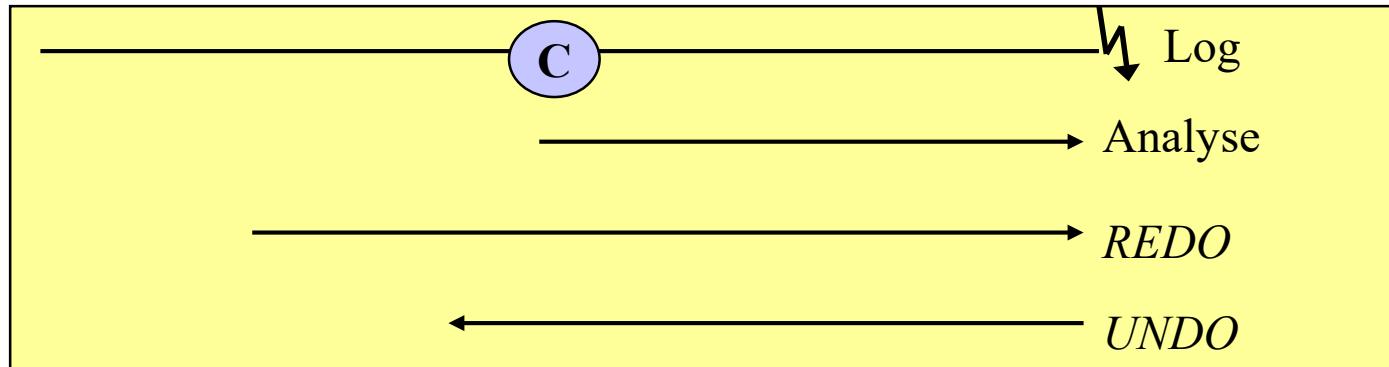
- Beispiel:



- beim Sicherungspunkt stehen S<sub>1</sub> und S<sub>2</sub> geändert im Puffer
- älteste noch nicht ausgeschriebene Änderung ist auf Seite S<sub>2</sub>
- *MinDirtyPageLSN* hat also den Wert 30, dort muss **REDO-Recovery** beginnen

# Sicherungspunkte

## Allgemeine Prozedur der Crash-Recovery

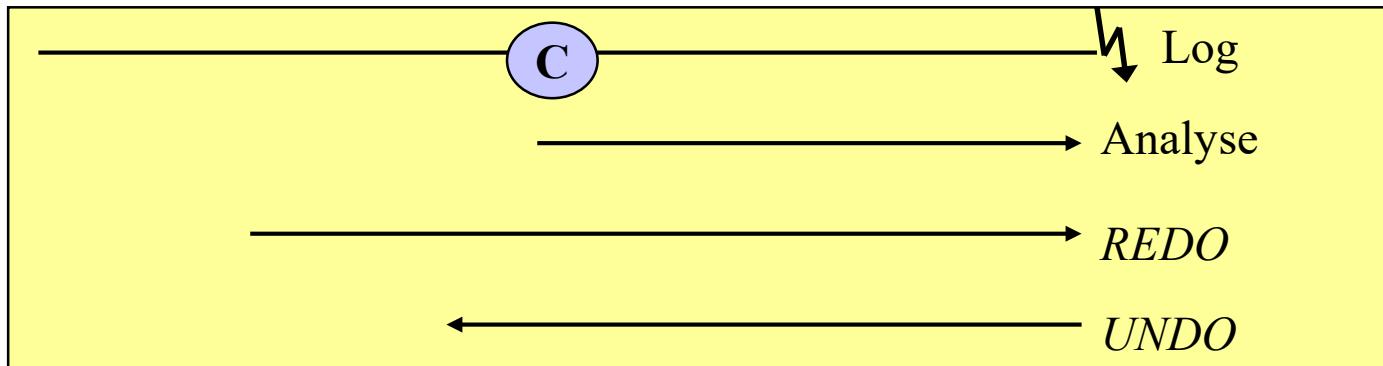


### 1. Analyse-Phase

- Lies Log-Datei vom letzten Sicherungspunkt bis zum Ende
- Bestimmung von Gewinner- und Verlierer-TAs, sowie der Seiten, die von ihnen geändert wurden
  - Gewinner: TAs, für die ein COMMIT-Satz im Log vorliegt
  - Verlierer: TAs, für die ein ROLLBACK-Satz bzw. kein COMMIT-Satz vorliegt
- Ermittle alle weiteren Seiten, die nach dem Checkpoint geändert wurden



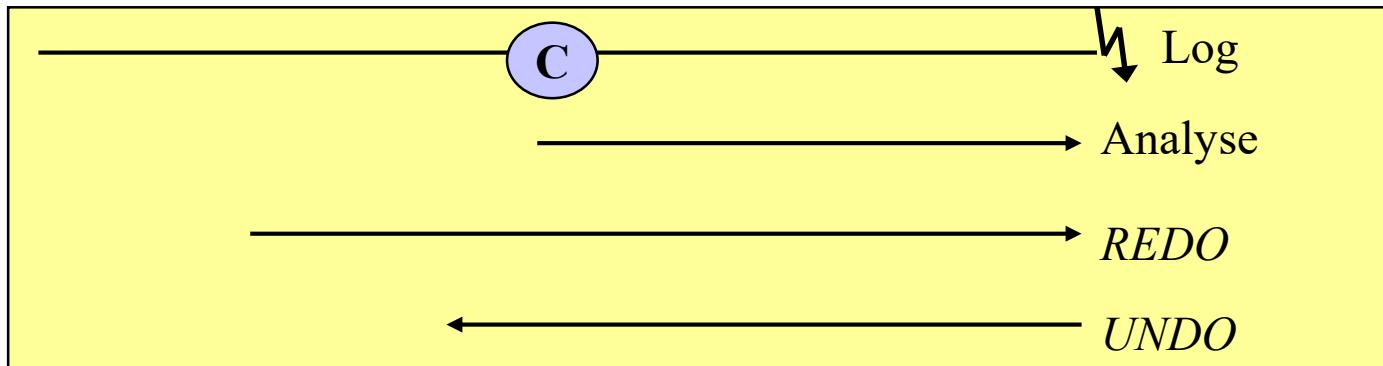
# Sicherungspunkte



## 2. REDO-Phase

- Vorwärtslesen der Log-Datei: Startpunkt ist abhängig vom Sicherungspunktyp
- Aufgabe: Wiederholen der Änderungen, die noch nicht in der DB vorliegen
- zwei Ansätze:
  - vollständiges **REDO** (*redo all*): Alle Änderungen werden wiederholt
  - selektives **REDO**: Nur die Änderungen der Gewinner-TAs werden wiederholt

# Sicherungspunkte



## 3. UNDO-Phase

- Rückwärtslesen der Log-Datei bis BOT der ältesten Verlierer TA
- Aufgabe: Zurücksetzen der Verlierer-TAs
- Fertig wenn Beginn der ältesten TA erreicht ist, die bei letztem Checkpoint aktiv war
- abhängig von REDO-Vorgehen:
  - vollständiges **REDO**: nur zum Fehlerzeitpunkt noch laufende TAs zurücksetzen
  - selektives **REDO**: **alle** Verlierer-TAs zurücksetzen

## 4. Abschluß der Recovery durch einen Sicherungspunkt



## Recovery der Recovery

- Nach einem Fehler während der Recovery beginnt die Recovery der Recovery wieder von vorne (Analysephase, **REDO**, **UNDO**, Sicherungspunkt)
- **REDO**- und **UNDO**-Phasen müssen *idempotent* sein
- Auch bei mehrfacher Ausführung müssen **REDO/UNDO** immer wieder dasselbe Ergebnis liefern
- D.h. zu jeder Änderungsaktion A muss gelten

$$\text{UNDO}(\text{UNDO}(\dots \text{UNDO}(A) \dots)) = \text{UNDO}(A)$$

$$\text{REDO}(\text{REDO}(\dots \text{REDO}(A) \dots)) = \text{REDO}(A)$$



## Idempotenz der REDO-Phase

- Für jeden Log-Eintrag  $E$ , für den ein **REDO** (tatsächlich) durchgeführt wurde, wird die  $LSN$  von  $E$  in die (betroffene) Seite eingetragen
- D.h. zu jeder Seite wird protokolliert, welche **REDO**-Operation als letztes ausgeführt wurde
- Verhindert, dass nach einem Absturz während der **REDO**-Phase, das erneute **REDO** nicht versehentlich auf dem AFIM aufsetzt



# Sicherungspunkte

## Idempotenz der UNDO-Phase

- Für jede ausgeführte **UNDO**-Operation wird **ein Compensation Log Record (CLR)** angelegt, der folgende Informationen enthalten muss:
  - Eindeutige Log Sequence Number (*LSN*)
  - ID der beteiligten TA
  - ID(s) der geänderten Seit(en)
  - **REDO**-Information: entspricht der **UNDO**-Operation, die ausgeführt wurde
  - Nach einem Fehler während einer **UNDO**-Operation wird diese Operation dann in der **REDO**-Phase ausgeführt und in der nachfolgenden **UNDO**-Phase übersprungen; dazu enthält jeder CLR einen Pointer zur *LSN* der zu dieser TA gehörenden Änderung, die der kompensierten Operation vorausging (relativ einfach aus *PrevLSN*-Einträgen zu ermitteln)

Skript zur Vorlesung:  
**Datenbanksysteme**  
Wintersemester 2023/2024

# Kapitel 10c

# Transaktionen – Integrität

Vorlesung: Prof. Dr. Thomas Seidl  
Übungen: Collin Leiber, Walid Durani





# Aufgaben eines DBMS

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

## 1. Synchronisation (Concurrency Control)

Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

## 2. Datensicherheit (Recovery)

Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)

## 3. Integrität (Integrity)

Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer



# Integritätsbedingungen

- Integritätsbedingungen (Integrity Constraints)
  - Bedingungen, die von einer Datenbank zu jedem Zeitpunkt erfüllt sein müssen
  - Typen
    - Schlüssel-Integrität
    - Referentielle Integrität
    - Multiplizitäten Constraints
    - Allgemeine Constraints
  - Diese Constraints sind
    - statisch, d.h. sie definieren Einschränkungen der möglichen **DB-Zustände** (Ausprägungen der Relationen)
    - dynamisch, d.h. sie spezifizieren Einschränkungen der möglichen **Zustandsübergänge** (Update-Operationen)



# Integritätsbedingungen

- Beispiele
  - Eindeutigkeit von Schlüssel-Attributen (Schlüssel-Integrität)
  - Ein Fremdschlüssel, der in einer anderen Relation auf seine Basisrelation verweist, muss in dieser Basisrelation tatsächlich existieren (Referentielle Integrität)
  - Bei  $1:m$ -Beziehungen müssen die Kardinalitäten beachtet werden – funktioniert z.B. durch Umsetzung mittels Fremdschlüssel auf der  $m$ -Seite (Multiplizitäten Constraint)
  - Wertebereiche für Attribute müssen eingehalten werden (allgemeines Constraint)  
Achtung: das Typkonzept in relationalen DBMS ist typischerweise sehr einfach, daher können Attribute mit der selben Domain verglichen werden, obwohl es möglicherweise semantisch keinen Sinn macht (z.B. MatrNr und VorlesungsNr)
  - ...
- Von wem werden diese und andere Integritätsbedingungen überwacht...
  - ... vom DBMS?
  - ... vom Anwendungsprogramm?



# Integritätsbedingungen

- Integritätsbedingungen sind Teil des Datenmodells
  - Wünschenswert ist eine zentrale Überwachung im DBMS innerhalb des Transaktionsmanagements
  - Einhaltung wäre unabhängig von der jeweiligen Anwendung gewährleistet, es gelten dieselben Integritätsbedingungen für alle Benutzer
- Für eine Teilmenge von Integritätsbedingungen (primary key, unique, foreign key, not null, check) ist dies bei den meisten relationalen Datenbanken realisiert => **deklarative Constraints**
- Für anwendungsspezifische Integritätsbedingungen ist häufig eine Definition und Realisierung im Anwendungsprogramm notwendig
  - Problem: Nur bei Verwendung des jeweiligen Anwendungsprogrammes ist die Einhaltung der Integritätsbedingungen garantiert sowie Korrektheit etc.
  - Meist: einfache Integritätsbedingungen direkt in DDL (deklarativ), Unterstützung für komplexere Integritätsbedingungen durch Trigger-Mechanismus => **prozedurale Constraints**



# Deklarative Constraints

- Deklarative Constraints sind Teil der Schemadefinition (create table ...)
- Arten:
  - Schlüsseleigenschaft: primary key (einmal), unique (beliebig)
    - unique kennzeichnet Schlüsselkandidaten
    - primary key kennzeichnet den Primärschlüssel
  - keine Nullwerte: not null (implizit bei primary key)
  - Typintegrität: Datentyp
  - Wertebedingungen: check (<Bedingung>)
  - referenzielle Integrität: foreign key ... references ...  
(nur Schlüssel)



# Deklarative Constraints

- Constraints können
  - attributsbezogen (für jeweils ein Attribut)
  - tabellenbezogen (für mehrere Attribute)definiert werden.
- Beschreibung im Entwurf meist durch geschlossene logische Formeln der Prädikatenlogik 1. Stufe

## Beispiele

- *Es darf keine zwei Räume mit gleicher R\_ID geben.*  
$$IB_1 : \forall r_1 \in \text{Raum} \ (\forall r_2 \in \text{Raum} \ (r_1[R\_ID] = r_2[R\_ID] \Rightarrow r_1 = r_2))$$
- *Für jede Belegung muss ein entsprechender Raum existieren.*  
$$IB_2 : \forall b \in \text{Belegung} \ (\exists r \in \text{Raum} \ (b[R\_ID] = r[R\_ID]))$$



# Deklarative Constraints

- Umsetzung in SQL?
  - Bei  $IB_1$  handelt es sich um eine Eindeutigkeitsanforderung an die Attributwerte von  $R\_ID$  in der Relation *Raum* (Schlüsseleigenschaft).
  - $IB_2$  fordert die referenzielle Integrität der Attributwerte von  $R\_ID$  in der Relation *Belegung* als Fremdschlüssel aus der Relation *Raum*.

```
CREATE TABLE raum (
    r_id          varchar2(10)      UNIQUE / PRIMARY KEY
  (IB1)           ...
);
```

```
CREATE TABLE belegung (
    b_id          number(10),
    r_id          varchar2(10)
    CONSTRAINT fk_belegung_raum      REFERENCES raum(r_id)
  (IB2)           ...
);
```



# Deklarative Constraints

- Überwachung von Integritätsbedingungen durch das DBMS
- *Definitionen:*
  - $S$  sei ein Datenbankschema
  - $IB$  sei eine Menge von Integritätsbedingungen  $\mathcal{I}$  über dem Schema  $S$
  - $DB$  sei Instanz von  $S$ , d.h. der aktuelle Datenbankzustand (über dem Schema  $S$ )
  - $U$  sei eine Update-Transaktion, d.h. eine Menge zusammengehöriger Einfüge-, Lösch- und Änderungsoperationen
  - $U(DB)$  sei der aktuelle Datenbankzustand nach Ausführen von  $U$  auf  $DB$
  - $Check(\mathcal{I}, DB)$  bezeichne den Test der Integritätsbedingung  $\mathcal{I} \in IB$  auf dem aktuellen Datenbankzustand  $DB$

$$Check(\mathcal{I}, DB) = \begin{cases} \text{true, falls } \mathcal{I} \text{ in } DB \text{ erfüllt ist} \\ \text{false, falls } \mathcal{I} \text{ in } DB \text{ nicht erfüllt ist} \end{cases}$$



# Deklarative Constraints

- *Wann sollen Integritätsbedingungen geprüft werden?*
  - 1. Ansatz: Periodisches Prüfen der Datenbank  $DB$  gegen **alle** Integritätsbedingungen:

```
for each U <seit letztem Check> do
  if ( $\forall I \in IB: \text{Check}(I, U(DB))$ ) then <ok>
  else <Rücksetzen auf letzten konsistenten Zustand>;
```

## Probleme:

- Rücksetzen auf letzten geprüften konsistenten Zustand ist aufwändig
- beim Rücksetzen gehen auch korrekte Updates verloren
- erfolgte lesende Zugriffe auf inkonsistente Daten sind nicht mehr rückgängig zu machen



# Deklarative Constraints

- 2. Ansatz: Inkrementelle Überprüfung bei jedem Update  $U$ 
  - Voraussetzung: Update erfolgt auf einem konsistenten Datenbankzustand
  - dazu folgende Erweiterung:

$$\text{Check}(I, U(DB)) = \begin{cases} \text{true, falls } I \text{ durch Update } U \text{ auf } DB \text{ nicht verletzt ist} \\ \text{false, falls } I \text{ durch Update } U \text{ auf } DB \text{ verletzt ist} \end{cases}$$

dann:

```
<führe U durch>;  
if ( $\forall I \in IB: \text{Check}(I, U(DB))$ ) then <ok>  
else <rollback U>;
```



# Deklarative Constraints

- Bei jedem Update  $\cup$  alle Integritätsbedingungen gegen die gesamte Datenbank zu testen ist immer noch zu teuer, daher Verbesserungen:
  1. Nur betroffene Integritätsbedingungen testen; z.B. kann die referenzielle Integritätsbedingung *Belegung*  $\rightarrow$  *Raum*, nicht durch
    - Änderungen an der Relation *Dozent* verletzt werden
    - Einfügen in die Relation *Raum* verletzt werden
    - Löschen aus der Relation *Belegung* verletzt werden(siehe nächste Folien)
  2. Abhängig von  $\cup$  nur vereinfachte Form der betroffenen Integritätsbedingungen testen; z.B. muss bei Einfügen einer *Belegung*  $x$  nicht die gesamte Bedingung  $IB_2$  getestet werden, sondern es genügt der Test von:
$$\exists r \in Raum (x[R\_ID] = r[R\_ID])$$



# Deklarative Constraints

- *Was muss eigentlich geprüft werden?*

Beispiel: Referentielle Integrität

- Gegeben:
  - Relation R mit Primärschlüssel  $\alpha$  (potentiell zusammengesetzt)
  - Relation S mit Fremdschlüssel  $\beta$  (potentiell zusammengesetzt) aus Relation R
  - Referentielle Integrität ist erfüllt, wenn für alle Tupel  $s \in S$  gilt
    1.  $s.\beta$  enthält nur **null**-Werte oder nur Werte ungleich **null**
    2. Enthält  $s.\beta$  keine **null**-Werte, existiert ein Tupel  $r \in R$  mit  $s.\beta = r.\alpha$
- D.h.
  - Der Fremdschlüssel  $\beta$  in S enthält genauso viele Attribute wie der Primärschlüssel  $\alpha$  in R
  - Die Attribute haben dieselbe Bedeutung, auch wenn sie umbenannt wurden
  - ***Es gibt keine Verweise auf ein undefiniertes Objekt (dangling reference)***  
Das Tupel s in S wird hier auch ***abhängiger Datensatz*** (vom entsprechenden r in R) genannt



# Deklarative Constraints

- Gewährleistung der Referentiellen Integrität
  - Es muss sichergestellt werden, dass keine dangling references eingebaut werden
  - D.h. für Relation R mit Primärschlüssel  $\underline{\alpha}$  und Relation S mit Fremdschlüssel  $\beta$  aus R muss folgende Bedingung gelten:

$$\pi_{\beta}(S) \subseteq \pi_{\underline{\alpha}}(R)$$

(also alle gültigen Werte in  $\beta$  in S müssen auch in R vorkommen)

- Erlaubte Änderungen sind also:
  1. Einfügen von Tupel s in S, wenn  $s.\beta \in \pi_{\underline{\alpha}}(R)$   
(Fremdschlüssel  $\beta$  verweist auf ein existierendes Tupel in R)
  2. Verändern eines Wertes  $w = s.\beta$  zu  $w'$ , wenn  $w' \in \pi_{\underline{\alpha}}(R)$   
(wie 1.)
  3. Verändern von  $r.\underline{\alpha}$  in R nur, wenn  $\sigma_{\beta=r.\underline{\alpha}}(S) = \emptyset$   
(es existieren keine Verweise in S auf Tupel r mit Schlüssel  $\underline{\alpha}$   
also keine abhängigen Tupel in S)
  4. Löschen von r in R nur, wenn  $\sigma_{\beta=r.\underline{\alpha}}(S) = \emptyset$   
(wie 3.)

Andernfalls: ROLLBACK der entspr. TA



# Deklarative Constraints

- Beim Löschen in R weitere Optionen:

| Option                     | Wirkung   |
|----------------------------|---|
| <b>ON DELETE NO ACTION</b> | Änderungsoperation wird zurückgewiesen, falls abhängiger Datensatz in S vorhanden                           |
| ON DELETE RESTRICT         |   |
| ON DELETE CASCADE          | Abhängige Datensätze in S werden automatisch gelöscht; kann sich über mehrstufige Abhängigkeiten fortsetzen |
| ON DELETE SET NULL         | Wert des abhängigen Fremdschlüssels in S wird auf null gesetzt  |
| ON DELETE SET DEFAULT      | Wert des abhängigen Fremdschlüssels in S wird auf den Default-Wert der Spalte gesetzt                       |



# Deklarative Constraints

- Wertebedingungen (statische Constraints nach **check**-Klauseln)
  - Dienen meist zur Einschränkung des Wertebereichs
  - Ermöglichen die Spezifikation von Aufzählungstypen,  
z.B. 

```
create table Professoren (
    ...
    Rang      character(2) check (Rang in ('W1', 'W2', 'W3')),
    ...
)
```
  - Ermöglicht auch, die referentielle Integrität bei zusammen gesetzten Fremdschlüsseln zu spezifizieren (alle Teile entweder null oder alle Teile nicht null)
  - Achtung: **check**-Constraints gelten auch dann als erfüllt, wenn die Formel zu **unknown** ausgewertet wird (kann durch **null**-Wert passieren!!!)  
(Übrigens im Ggs. zu **where**-Bedingungen)



# Deklarative Constraints

- Komplexere Integritätsbedingungen
  - In einer `check`-Bedingung können auch Unteranfragen stehen => IBs können sich auf mehrere Relationen beziehen (Verallgemeinerung der ref. Int.)
  - Beispiel:
    - Tabelle `pruefen` modelliert Relationship zwischen Student, Professor und Vorlesung
    - Das Constraint `VorherHoeren` garantiert, dass Studenten sich nur über Vorlesungen prüfen lassen können, die sie auch gehört haben

```
create table pruefen (
    MatrNr      integer references Studenten ...
    VorlNr      integer references Vorlesungen ...
    PersNr      integer references Professoren ...
    Note        numeric(2,1) check (Note between 1.0 and 5.0),
    primary key (MatrNr, VorlNr)
    constraint VorherHoeren
        check(      exists (  select * from hoeren h, pruefen p where
                                h.VorlNr = p.VorlNr and h.MatrNr = p.MatrNr
                           )
    )
)
```

- Diese IBs werden leider kaum unterstützt (Lösung: Trigger)

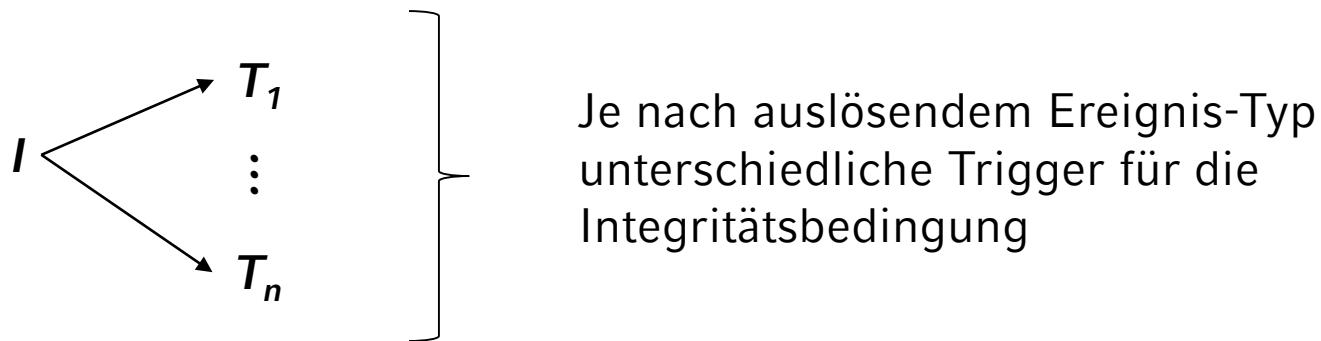


# Prozedurale Constraints (Trigger)

- Motivation: Komplexere Bedingungen als bei deklarativen Constraints und damit verbundene Aktionen wünschenswert.
- Trigger: Aktion (typischerweise PL/SQL-Programm), die einer Tabelle zugeordnet ist und durch ein bestimmtes Ereignis ausgelöst wird.
- Ein Trigger enthält Code, der die mögliche Verletzung einer Integritätsbedingung bei einem bestimmten Ereignis-Typ testet und daraufhin bestimmte Aktionen veranlasst.
- mögliche Ereignisse: `insert`, `update`, `delete`
- zwei Arten:
  - **Befehls-Trigger** (*statement trigger*): werden einmal pro auslösendem Befehl ausgeführt.
  - **Datensatz-Trigger** (*row trigger*): werden einmal pro geändertem/eingefügtem/gelöschenem Datensatz ausgeführt.
- mögliche Zeitpunkte: vor (`BEFORE`) oder nach (`AFTER`) dem auslösenden Befehl

# Prozedurale Constraints (Trigger)

- Datensatz-Trigger haben Zugriff auf zwei Instanzen eines Datensatzes: vor und nach dem Ereignis (Einfügen/Ändern/Löschen)  
=> Adressierung durch Präfix: `new.` bzw. `old.` (Syntax systemspezifisch)
- Befehlstrigger haben Zugriff auf die Änderungen durch die auslösenden Befehle (die typischerweise Tabellen verändern)  
=> Adressierung durch `newtable` bzw. `oldtable` (Syntax systemspezifisch)
- Zu einer Integritätsbedingung / gehören in der Regel mehrere Trigger  $T_i$





# Prozedurale Constraints (Trigger)

- Aufbau eines Trigger-Programms:

```
create or replace trigger <trig_name>
before/after/instead of -- Trigger vor/nach/alternativ zu Auslöser ausführen
insert or update of <attrib1>, <attrib2>, ... or delete -- Trigger-Ereignisse
on <tab_name>/<view_name>/          -- zugehörige Tabelle od. View (DML-Trigger)
    <schema_name>/<db_name>      -- Schema od. Datenbank (DDL-Trigger)
[for each row]                      -- Datensatz-Trigger
when <bedingung>                  -- zusätzliche Trigger-Restriktion
declare
...
begin
if inserting then <pl/sql Anweisungen>
end if;
if updating (<attrib1>) then <pl/sql Anweisungen>
end if;
if deleting then <pl/sql Anweisungen>
end if;
...
end;                                -- Code hier gilt für alle Ereignisse
```



# Prozedurale Constraints (Trigger)

- Beispiel
  - Ausgangspunkt: Relation *Period\_Belegung* mit regelmäßig stattfindenden Lehrveranstaltungen in einem Hörsaal
  - Hier sollen folgende Bedingungen gelten:
$$\forall p \in \text{Period\_Belegung} (0 \leq p[\text{Tag}] \leq 6 \wedge p[\text{Erster\_Termin}] \leq p[\text{Letzter\_Termin}] \wedge \text{Wochentag}(p[\text{Erster\_Termin}]) = p[\text{Tag}] \wedge \text{Wochentag}(p[\text{Letzter\_Termin}]) = p[\text{Tag}])$$

Formulierung als deklaratives Constraint:

```
ALTER TABLE Period_Belegung ADD CONSTRAINT check_day
CHECK (
    (Tag between 0 and 6) and
    (Erster_Termin <= Letzter_Termin) and
    (to_number (to_char (Erster_Termin, 'd')) = Tag) and
    (to_number (to_char (Letzter_Termin, 'd')) = Tag)
);
```



# Prozedurale Constraints (Trigger)

Formulierung als prozedurales Constraint (Trigger):

```
CREATE OR REPLACE TRIGGER check_day
    BEFORE
    INSERT OR UPDATE
    ON Period_Belegung
    FOR EACH ROW
    DECLARE
        tag number; et date; lt date;
    BEGIN
        tag := new.Tag;
        et := new.Erster_Termin; lt := new.Letzter_Termin;
        if (tag < 0) or (tag > 6) or (et > lt) or
            (to_number(to_char(et, 'd')) != tag) or
            (to_number(to_char(lt, 'd')) != tag)
        then
            raise_application_error(-20089, 'Falsche Tagesangabe');
        end if;
    END;
```



# Prozedurale Constraints (Trigger)

- Verwandtes Problem: Sequenzen für die Erstellung eindeutiger IDs

```
CREATE SEQUENCE <seq_name>
  [INCREMENT BY n]                      -- Default: 1
  [START WITH n]                        -- Default: 1
  [{MAXVALUE n | NOMAXVALUE}]          -- Maximalwert (10^27 bzw. -1)
  [{MINVALUE n | NOMINVALUE}]           -- Mindestwert (1 bzw. -10^26)
  [{CYCLE | NOCYCLE}]
  [{CACHE n | NOCACHE}];                -- Vorcachen, Default: 20
```

- Zugreifen über NEXTVAL (nächster Wert) und CURRVAL (aktueller Wert):

```
CREATE SEQUENCE seq_pers;
INSERT INTO Person (p_id, p_name, p_alter)
    VALUES (seq_pers.NEXTVAL, 'Ulf Mustermann', 28);
```



# Prozedurale Constraints (Trigger)

- Beispiel mit Trigger:

```
CREATE OR REPLACE TRIGGER pers_insert
BEFORE
INSERT ON Person
FOR EACH ROW
BEGIN
    SELECT seq_pers.NEXTVAL
    INTO new.p_id
    FROM dual;
END;

INSERT INTO Person (p_name, p_alter)
VALUES ('Ulf Mustermann', 28);
```

- Vorteil: Zuteilung der ID erfolgt transparent, d.h. kein expliziter Zugriff (über .NEXTVAL) in INSERT-Statement nötig!



# Prozedurale Constraints (Trigger)

- Allgemeines Schema der Trigger-Abarbeitung

Event  $e$  aktiviert während eines Statements  $S$  einer Transaktion eine Menge von Triggern  $T = (T_1, \dots, T_k)$

1. Füge alle neu aktivierten Trigger  $T_1, \dots, T_k$  in die **TriggerQueue**  $Q$  ein
2. Unterbreche die Bearbeitung von  $S$
3. Berechne `new` und `old` bzw. `newtable` und `oldtable`
4. Führe alle BEFORE-Trigger in  $T$  aus, deren Vorbedingung erfüllt ist
5. Führe die Updates aus, die in  $S$  spezifiziert sind
6. Führe die AFTER-Trigger in  $T$  aus wenn die Vorbedingung erfüllt ist
7. Wenn ein Trigger neue Trigger aktiviert, springe zu Schritt 1



# Prozedurale Constraints (Trigger)

- Achtung:  
Eine (nicht-terminierende) Kettenreaktion von Triggern ist grundsätzlich möglich
- Eine Menge von Triggern heißt **sicher (safe)**, wenn eine potentielle Kettenreaktion immer terminiert
  - Es gibt Bedingungen die hinreichend sind um Sicherheit zu garantieren (d.h. wenn sie erfüllt sind, ist die Trigger-Menge sicher, es gibt aber sichere Trigger-Mengen, die diese Bedingungen nicht erfüllen)
  - Typischerweise gibt es aber keine hinreichend und notwendigen Bedingungen, daher ist Sicherheit algorithmisch schwer zu testen.
- Eine Möglichkeit wäre wieder einen Abhängigkeits- (bzw. Aktivierungs-)graph
  - Knoten: Trigger
  - Kante von  $T_i$  nach  $T_j$  wenn die Ausführung von  $T_i$   $T_j$  aktivieren kann
  - Keine Zyklen implizieren Sicherheit (Zyklen implizieren nicht notwendigerweise Unsicherheit)
  - ABER: ineffizient und nicht einfach zu realisieren (automatische Erkennung wann  $T_i$   $T_j$  aktivieren kann?)



# Prozedurale Constraints (Trigger)

- Trigger können auch noch für andere Aufgaben verwendet werden
  - Implementierung von Integritätsbedingungen und Erzeugung eindeutiger IDs (siehe dieses Kapitel)
  - Implementierung von Geschäftsprozessen (z.B. wenn eine Buchung ausgeführt wird, soll eine Bestätigungs-Email versandt werden)
  - Monitoring von Einfügungen/Updates (im Prinzip eine Kopplung der ersten beiden: wenn ein neuer Wert eingefügt wird, kann abhängig davon ein entsprechendes Ereignis ausgelöst werden)
  - Verwaltung temporär gespeicherter oder dauerhaft materialisierter Daten (z.B. materialisierte Views)