

# DBS Prüfung Zusammenfassung

## Vorwort

---

Diese Stoffsammlung/Zusammenfassung enthält den Stoff, der in der DBS Vorlesung der TU Wien im Sommersemester 2025 vorgetragen wurde, der auch in den jeweiligen Slides zu finden ist. Die Struktur dieser Zusammenfassung basiert demnach auch auf den Slides.

### Disclaimer

Vieles der Zusammenfassung wurde mit AI generiert, basiert allerdings nur auf Inhalten der Unterlagen. Die Stellen die mit AI generiert wurden, wurden von mir überprüft und mit den Unterlagen verglichen, aber auch ich kann Fehler machen.

Demnach, falls sich irgendwo Fehler befinden oder es Verbesserungsvorschläge gibt, bitte an [@xmozz](#) auf Discord wenden.

## Legende:

---

 Definitionen / Wichtiges / etc.

 Beispiele

 Aufzählungen / etc.

 Info

Die restlichen sind Situationsabhängig

## ⓘ Inhalt

- 0. Testinformationen
- 1. Relationale Algebra
- 2. ER-Diagramme
- 3. Normalization
- 4. SQL
- 5. Transaktionen
- 6. Physischer Datenbankentwurf
- 7. Anfrageoptimierung

# 0. Testinformationen

## Allgemeine Information (slide)

---

### Termine

- SQL-Prüfung Wiederholungstermin  
05.06.2025 (Anmeldung endet am 3. Juni um 23:59 Uhr)
- Schriftliche Prüfung  
30.06.2025 18:00-20:00
- Schriftliche Prüfung Wiederholungstermin  
22.09.2025 18:00-20:00

Teilnahme an einem Wiederholungstermin bedeutet

- Das Ergebnis beim Wiederholungstermin ersetzt die vorherige Punktezahl
- **Achtung: es ist möglich, sich zu verschlechtern**

## Schriftliche Prüfung (slide)

---

### Prüfungsmodus:

- 60 Minuten Zeit
- Multiple Choice-Fragen
- Praktische Aufgaben und theoretische Fragen

### Prüfungsstoff

- Alle Themen, die in der Vorlesung durchgenommen wurden
- SQL: alles, bis auf den Stoff vom SQL Exam  
d.h. keine SQL-Anfragen, aber sehr wohl Theorie und z.B. CREATE-Statements

## Ausstellen der Zeugnisse (slide)

---

Nach dem Wiederholungstermin der schriftlichen Prüfung

- Default

Am Ende dieses Semesters

- Alle mit „sehr gut“ am Ende des Semesters  
(d.h. unter Berücksichtigung von SQL-Exam, schriftlicher Prüfung im Juni, Punkte von Quizzes und Übungen - inklusive nachgebrachter Übungen)
- Auf expliziten Wunsch durch Eintragen in einer Liste

- Dann aber keine Teilnahme am Wiederholungstermin der schriftlichen Prüfung im September

# 1. Relationale Algebra

## Relationenmodell

---

### Domänen

Seien  $D_1, D_2, \dots, D_n$  Domänen (=Wertebereiche).

### Relationen

$R \subseteq D_1 \times \dots \times D_n$ .

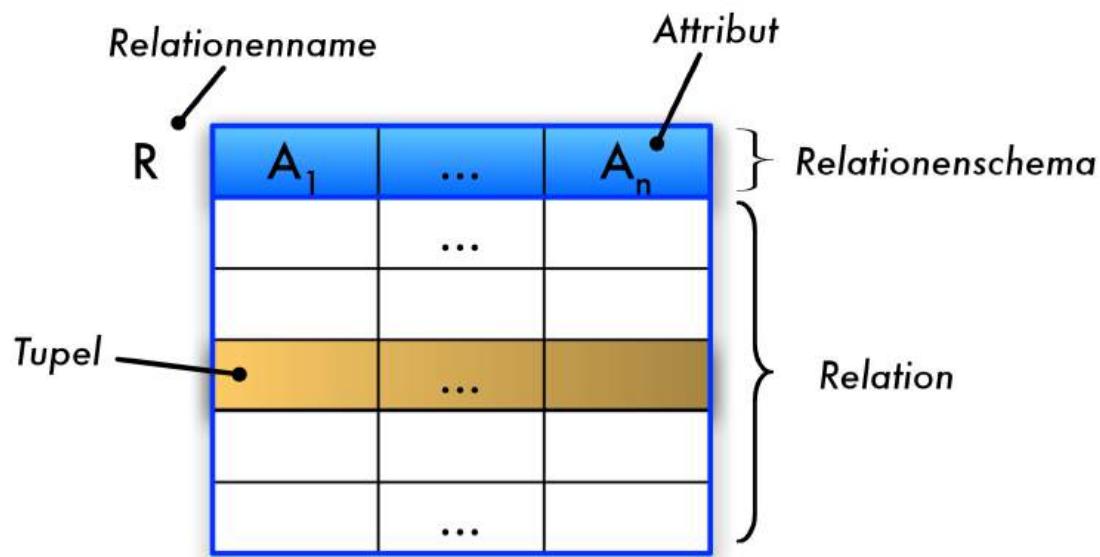
- Beispiel:  $\text{telephoneBook} \subseteq \text{string} \times \text{string} \times \text{integer}$
- Wertebereiche dürfen identisch sein:  $D_i = D_j$  für  $i \neq j$ .
- Basierend auf Mengen.

### Relationenschema

- Legt die Struktur der gespeicherten Daten fest.
- Wird mit  $\text{sch}(R)$  oder  $R$  bezeichnet.
- Notation:  $R(A_1 : D_1, A_2 : D_2, \dots)$  mit  $A_i$  für Attribute.
- Beispiel:  $\text{telephoneBook}(\text{name} : \text{string}, \text{street} : \text{string}, \text{phoneNumber} : \text{integer})$

#### Veranschaulichung der Grundbegriffe

- Fettgeschriebene Kopfzeile: Relationenschema
- Spaltenüberschrift: Attribut
- Weitere Einträge in der Tabelle: Relation
- Eine Zeile der Tabelle: Tupel
- Ein Eintrag: Attributwert
- Unterstrichenes Attribut: Primärschlüssel



*Theorie vs. Realität:* In der Realität meist keine Unterscheidung zwischen Ausprägung (Instanz) und Schema einer Relation.

### ⌚ Fremdschlüssel

- Eine Relation kann die Primärschlüsselattribute einer anderen Tabelle beinhalten.
- Werte der Fremdschlüsselattribute müssen im Primärschlüssel der referenzierten Tabelle vorkommen. (Dies stellt die referentielle Integrität sicher, d.h., dass Verweise auf nicht existierende Einträge vermieden werden.)

## Eigenschaften des Relationalen Modells

### ⌚ Tupelreihenfolge

Tupel einer Relation haben keine **Reihenfolge**.

name	age
Pat	1
Fred	2
Sue	3
Pam	4

name	age
Sue	3
Pam	4
Fred	2
Pat	1

Diese Relationen beinhalten dieselben Informationen. (Die Reihenfolge, in der Zeilen in einer Datenbanktabelle gespeichert oder abgerufen werden, ist irrelevant.)

## ⓘ Attributreihenfolge

Die **mathematische Definition** von Tupeln sieht eine **bestimmte Reihenfolge** der Attribute des Tupels/der Relation vor.

name	age
Pat	1
Fred	2
Sue	3
Pam	4

age	name
1	Pat
2	Fred
3	Sue
4	Pam

Aber...

- Die Reihenfolge der Attribute ist in den meisten Anwendungen bedeutungslos.
- Das Verwenden von Attributnamen statt einer bestimmten Reihenfolge ist praktischer. (Man spricht Attribute über ihren Namen an, nicht über ihre Position.)
- Das kartesische Produkt wird kommutativ. ( $A \times B$  ist in der Praxis dasselbe wie  $B \times A$  bei Verwendung von Attributnamen.)

## ⓘ Atomare Werte

- Werte eines Tupels sind **atomar** (unteilbar).
- Ein Wert kann kein zusammengesetzter Datentyp (Liste, Array, ...) oder eine Relation sein.

name	age
Pat	1
Fred	2
Sue	3
Pam	4

Alle Werte sind atomar

name			age
Pat	Jensen		1
Fred	D.	Roosevelt	2
Sue	H.M.I.	Knuth	3
Pam	C.	Anderson	4

name ist nicht atomar

*Alle Werte sind atomar* vs. *name ist nicht atomar* (da "H.M. Knuth" oder "C. Anderson" aus mehreren Teilen bestehen könnten, was in der mathematischen Definition nicht atomar wäre.)

## ⓘ Null Werte

Ein spezieller **Null** Wert wird verwendet, um unbekannte bzw. für gewisse Tupel unanwendbare Werte zu repräsentieren.

name	age
Pat	1
Fred	2
Sue	null
Pam	null

name	age
Pat	1
Fred	2
Sue	$\perp$
Pam	$\perp$

Alternative Notation

### ⓘ Duplikate

Eine Relation folgt der **mathematischen Definition einer Menge**.

name	age
Pat	1
Fred	2
Sue	3
Pam	4

zulässig

name	age
Pat	1
Fred	2
Sue	3
Sue	3

unzulässig

Keine zwei Tupel einer Relation dürfen identische Werte in allen Attributen beinhalten.  
(Jedes Tupel muss einzigartig sein.)

## Relationale Algebra

### 🔎 Basisoperatoren

- Projektion  $\pi$
- Selektion  $\sigma$
- Umbenennung  $\rho$
- Kreuzprodukt  $\times$
- Vereinigung  $\cup$
- Differenz –

Jede Anfrage in relationaler Algebra kann ausschließlich mit Basisoperatoren ausgedrückt werden.

Das Weglassen eines Basisoperators verringert die Ausdrucksstärke. (Man kann dann nicht mehr alle möglichen Anfragen formulieren.)

### ⓘ Weitere Operatoren

- Schnitt  $\cap$
- Join (Verbund)  $\bowtie$
- Linker äußerer Join  $\bowtie_l$
- Rechter äußerer Join  $\bowtie_r$
- Äußerer Join  $\bowtie_e$
- Semi-Join (linker)  $\ltimes$
- Semi-Join (rechter)  $\ltimes_r$
- Gruppierung  $\gamma$
- Division  $\div$

## Projektion

$$\pi_{name, dep\_name}(instructor)$$

Das Resultat ist eine Relation mit  $n$  Spalten, die durch das Weglassen der nicht angegebenen Spalten entsteht.

(In [SQL](#) ist das das `SELECT`)

---

## Selektion

- Symbol:  $\sigma_F$
- Selektionsprädikat  $F$  besteht aus:

*Logischen Operatoren:  $\vee$  (oder),  $\wedge$  (und),  $\neg$  (nicht)*

*Arithmetischen Vergleichsoperatoren:  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$ ,  $>$*

\* ...und natürlich aus Attributnamen der Argumentrelation oder Konstanten als Operanden.

Auswahl von Zeilen einer Tabelle anhand eines Selektionsprädikats.

$$\sigma_{salary > 80000}(instructor)$$

(In [SQL](#) ist das das `WHERE`)

---

## Umbenennung ( $\rho$ )

### Umbenennung einer Relation

- **Syntax:**  $\rho_S(R)$
- **Erklärung:** Die Relation  $R$  erhält den neuen Namen  $S$ .
  - **Beispiel:** Eine Relation namens `Kunden` könnte in `Besteller` umbenannt werden, um in einem bestimmten Kontext verständlicher zu sein.

### Umbenennung von Spalten (Attributen)

- **Syntax:**  $\rho_{A \leftarrow B}(R)$
  - **Erklärung:** Das Attribut  $B$  in der Relation  $R$  wird in  $A$  umbenannt.
    - **Beispiel:** In einer Relation `Produkte` könnte das Attribut `ProdNr` in `Produktnummer` umbenannt werden.
- 

## Kartesisches Produkt (Kreuzprodukt) ( $\times$ )

Das Kartesische Produkt **kombiniert alle möglichen Tupel (Zeilen) aus zwei Relationen miteinander**. Es erzeugt eine neue Relation, die alle Tupelpaare aus den ursprünglichen Relationen enthält.

- **Syntax:**  $(R \times S)$
- 

## Mengenoperationen

Damit Mengenoperationen auf zwei Relationen  $R$  und  $S$  angewendet werden können, müssen diese **vereinigungskompatibel** sein.

### ⌚ Definition - Vereinigungskompatibel

- **Gleiche Anzahl an Attributen:** Beide Relationen müssen die gleiche Anzahl von Spalten (Attributen) besitzen.
- **Gleicher Wertebereich in Spaltenreihenfolge:** Für jedes Attribut in der jeweiligen Spaltenreihenfolge müssen die Wertebereiche (Datentypen) gleich sein. Das heißt, wenn das erste Attribut von  $R$  ein `Integer` ist, muss das erste Attribut von  $S$  ebenfalls ein `Integer` sein und so weiter für alle Attribute.

### Vereinigung ( $\cup$ )

Die Vereinigung von zwei Relationen  $R$  und  $S$  sammelt alle Tupel (Zeilen) aus beiden Relationen und entfernt dabei Duplikate. Das Ergebnis ist eine neue Relation, die alle einzigartigen Tupel enthält, die entweder in  $R$  oder in  $S$  (oder in beiden) vorkommen.

- **Syntax:**  $(R \cup S)$
- **Beispiel:** Eine Vereinigung aller Abteilungsnamen von Instruktoren und Studenten.

$$\pi_{name, dep\_name}(instructor) \cup \pi_{name, dep\_name}(student)$$

## Differenz (Ohne, Minus) ( $-$ oder $\setminus$ )

Die Differenz von zwei Relationen  $R$  und  $S$  eliminiert alle Tupel aus der ersten Relation ( $R$ ), die auch in der zweiten Relation ( $S$ ) vorkommen. Das Ergebnis ist eine neue Relation, die nur die Tupel enthält, die exklusiv in  $R$  sind und nicht in  $S$  vorkommen.

- **Syntax:**  $(R - S)$  bzw.  $(R \setminus S)$

## Schnitt (Durchschnitt) ( $\cap$ )

Der Schnitt von zwei Relationen  $R$  und  $S$  besteht aus der Menge aller Tupel, die in beiden Relationen gemeinsam vorkommen. Im Gegensatz zur Vereinigung und Differenz ist der **Schnitt kein Basisoperator** in der relationalen Algebra, da er sich aus den Basisoperatoren (Differenz) ableiten lässt.

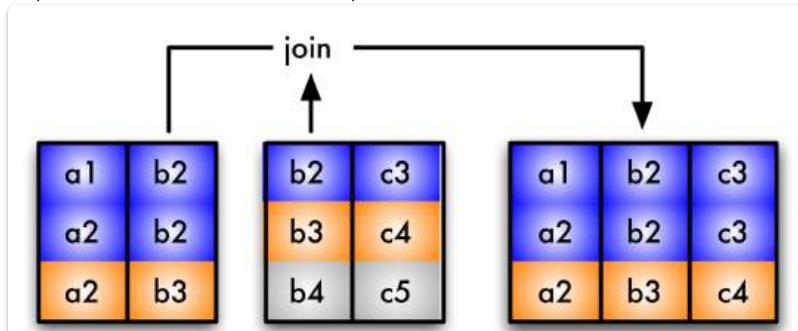
- **Syntax:**  $(R \cap S)$
- **Erklärung:** Der Schnitt liefert alle Tupel, die sowohl in Relation  $R$  als auch in Relation  $S$  enthalten sind.

## Natural Join ( $\bowtie$ )

Er verknüpft Tabellen (Relationen) über **gleichbenannte Spalten** und verschmilzt jeweils zwei Tupel, wenn sie dort **gleiche Werte** aufweisen.

- **Gegeben zwei Relationen (+ Schemata):**

- $R(A_1, \dots, A_m, B_1, \dots, B_k)$
- $S(B_1, \dots, B_k, C_1, \dots, C_n)$



- Natural Join ist Kommutativ
- 

## Weitere Joins

### Theta-Join ( $\theta$ -Join oder allgemeiner Join)

Der Theta-Join ist eine sehr flexible Join-Operation, die zwei Relationen basierend auf einem beliebigen Prädikat (Bedingung) verknüpft, das die beteiligten Attribute betrifft.

Also beispielsweise:

$$\pi_{buch.autor}((gelesen \bowtie_{gelesen.titel=buch.titel} buch))$$

### Equi-Join

Der Equi-Join ist ein Spezialfall des Theta-Joins, bei dem das Join-Prädikat  $\theta$  nur Gleichheit (=) prüft.

### Outer Joins

- left-outer-joins nehmen auch Partnerlose Tupel der linken relation mit
- right... von rechts
- full-outer-joins nehmen alles kommt mit

Symbol:



### Semijoins

Semi-Joins sind spezielle Join-Varianten, die dazu dienen, Tupel aus einer Relation zu finden, die einen Joinpartner in einer anderen Relation haben, aber ohne die Attribute der Joinpartner-Relation dem Ergebnis hinzuzufügen. Sie sind nützlich, wenn man nur wissen möchte, welche Tupel aus der Ausgangsrelation "passen", ohne die kombinierten Informationen.

Das gibt's wieder links und rechts

Links:

- **Definition:** Finde alle Tupel der **linken Relation**, die Joinpartner in der **rechten Relation** haben.

- **Syntax:**  $L \times R = \pi_L(L \bowtie R)$

Rechts:

- **Definition:** Finde alle Tupel der **rechten Relation**, die Joinpartner in der **linken Relation** haben.
- **Syntax:**  $L \times R = \pi_R(L \bowtie R)$

## Gruppierung und Aggregation

### ⌚ Gruppierung

Tupel mit gleichen Attributwerten (für eine angegebene Liste von Attributen) werden **gruppiert**. Das bedeutet, die Relation wird in Gruppen unterteilt, wobei alle Tupel innerhalb einer Gruppe dieselben Werte für die angegebenen Gruppierungsattribute haben.

### ⌚ Aggregation

Auf jede dieser gebildeten Gruppen wird anschließend eine **Aggregatfunktion** angewendet. Diese Funktionen berechnen einen einzelnen Wert für jede Gruppe, der die zusammengefassten Daten dieser Gruppe repräsentiert.

## Typische Aggregatfunktionen

- **count:** Zählt die Anzahl der Elemente (Tupel) pro Gruppe.
- **sum:** Berechnet die Summe der Werte eines bestimmten Attributs pro Gruppe.
- **min, max, avg:** Berechnen das Minimum, Maximum bzw. den Durchschnittswert eines bestimmten Attributs pro Gruppe.

## Rationale Division ( $\div$ )

- Wichtig für "für alle" Abfragen
- **Formale Definition:**

$$R \div S = \pi_{R-S}(R) - \pi_{R-S}((\pi_{R-S}(R) \times S) - R)$$

- 1. Schritt 1: Projektion von R auf die Nicht-S-Attribute** ( $\pi_{R-S}(R)$ ): Dies sind die möglichen "Kandidaten" für das Ergebnis der Division (im Beispiel: die `studIDs` ).
- 2. Schritt 2: Kreuzprodukt der Kandidaten mit S** ( $\pi_{R-S}(R) \times S$ ): Für jeden Kandidaten wird eine "erwartete" Menge an Tupeln generiert, die alle Kombinationen des Kandidaten mit jedem Element aus S enthält. Dies repräsentiert, welche Kombinationen jeder Kandidat *haben sollte*, um alle Elemente von S zu "erfüllen".

3. Schritt 3: Differenz  $((\pi_{R-S}(R) \times S) - R)$ : Von diesen erwarteten Tupeln wird die ursprüngliche Relation  $R$  abgezogen. Das Ergebnis sind die Tupel, die ein Kandidat *nicht* hat, obwohl er sie haben *sollte*, um alle Elemente von  $S$  zu erfüllen.
4. Schritt 4: Letzte Differenz  $(\pi_{R-S}(R) - \dots)$ : Von den ursprünglichen Kandidaten (aus Schritt 1) werden diejenigen abgezogen, die in Schritt 3 als "nicht vollständig" identifiziert wurden. Was übrig bleibt, sind genau die Kandidaten, die *alle* erforderlichen Verbindungen zu  $S$  hatten.

## 2. ER-Diagramme

### Schritte des Datenbankentwurfs

#### 1) Anforderungsanalyse

##### ⌚ Definition

Die Anforderungsanalyse ist der Prozess des Sammelns und Analysierens von Informationen über die Bedürfnisse und Erwartungen der Benutzer an das zukünftige Datenbanksystem.

##### ☰ Beispiele für Anforderungen

- Studierende nehmen an Vorlesungen teil.
- Professor:innen bieten Vorlesungen an.
- Studierende werden durch die Matrikelnummer eindeutig identifiziert.

#### 2) Erstellung ER-Modell

- **Ziel:** Erstellung eines ER-Modells (Entity-Relationship-Modell) basierend auf den identifizierten Anforderungen.

#### 3) Abbildung auf relationale Modell

- **Ziel:** Überführung des ER-Modells in ein relationales Schema, das aus Tabellen (Relationen), Attributen und Schlüsseln besteht.

#### 4) Praktische Umsetzung und Implementierung

- **Ziel:** Die tatsächliche Erstellung der Datenbank und ihrer Tabellen in einem Datenbanksystem (DBMS) und das Einfügen von Daten.

### Grundkonzepte des ER-Modells

#### Entität und Entitätstypen

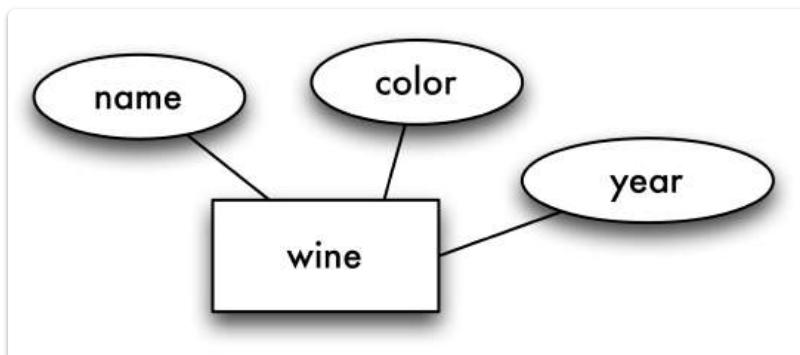
- **Entitäten** sind Objekte der realen Welt, über die wir Informationen abspeichern wollen.
  - Nur **Eigenschaften** der Entitäten können in einer Datenbank gespeichert werden (Beschreibung), nicht die Entitäten selbst.
- Entitäten werden in **Entitätstypen** eingeteilt.

# wine

- Eine **Entitymenge (entity set)** ist eine konkrete Menge von Entitäten des gleichen Entitätstyps.
  - Die zwei Begriffe Entitymenge (entity set) und Entitätstypen (entity type) werden oft als Synonyme verwendet.

## Attribute

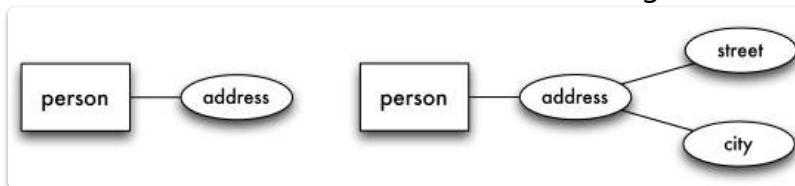
- **Attribute** modellieren Eigenschaften von Entitäten oder auch Beziehungen.
  - Alle Entitäten eines Entitätstyps haben dieselben Arten von Eigenschaften.
  - Attribute werden für Entitätstypen deklariert.
  - Attribute haben eine **Domäne** bzw. **Wertemenge** (eine definierte Menge von erlaubten Werten, z.B. für "Alter" nur positive ganze Zahlen).



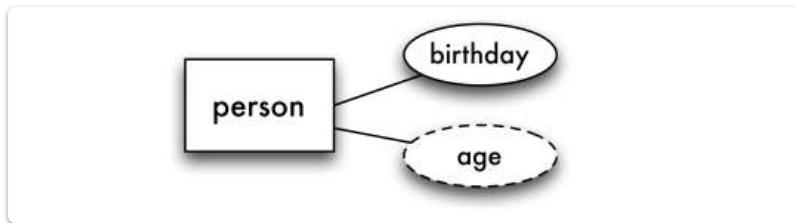
Attribute können einwertig oder mehrwertig sein. Also entweder nur eine Value haben oder auch mehrere. Wenn es mehrere sind, dann ist es doppelt umrandet:



Außerdem können Attribute auch zusammengesetzt sein:



Wenn man von einem Attribut auf ein anderes schließt macht man man's gestrichelt:



## Schlüssel

- Ein **(Super-)Schlüssel** besteht aus einer Untermenge von Attributen eines Entitätstyps  $E(A_1, \dots, A_m)$ .
  - Die Menge der Schlüsselattribute ist  $\{S_1, \dots, S_k\} \subseteq \{A_1, \dots, A_m\}$ .
  - Die Attribute  $S_1, \dots, S_k$  eines Schlüssels werden **Schlüsselattribute** genannt.
- Die Werte der Schlüsselattribute identifizieren zusammen eindeutig ein bestimmtes Entität. (Das bedeutet, dass es keine zwei Entitäten geben kann, die die gleichen Werte für alle Schlüsselattribute haben.)
- Ein **Schlüsselkandidat** ist ein **minimaler Schlüssel**. (Minimal bedeutet, dass kein Attribut aus dem Schlüsselkandidaten entfernt werden kann, ohne dass die Eindeutigkeit verloren geht.)
- Gibt es mehrere Schlüsselkandidaten, so wird einer als **Primärschlüssel** ausgewählt.
  - Primärschlüssel werden **unterstrichen**

## Beziehung und Beziehungstyp

- Eine **Beziehung** beschreibt die Verbindung zwischen Entitäten.
- Beziehungen zwischen Entitäten werden zu **Beziehungstypen** zusammengefasst.



## Eigenschaften von Beziehungstypen

### Merkmale von Beziehungstypen

- **Stelligkeit bzw. Grad**
  - Beschreibt die Anzahl der beteiligten Entitätstypen an einer Beziehung.
  - Häufig: **binär** (zwei Entitätstypen beteiligt)
  - Weniger häufig: **ternär** (drei Entitätstypen beteiligt)
  - Allgemein: **n-stellig** ( $n$  Entitätstypen beteiligt)

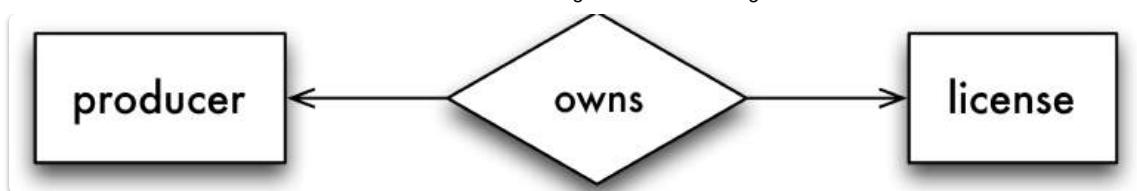
### Notation funktionaler Beziehungstypen

- **1:N Beziehungstyp:**



- Hier kann ein Produzent viele Weine produzieren, aber ein Wein wird nur von einem Produzenten produziert.

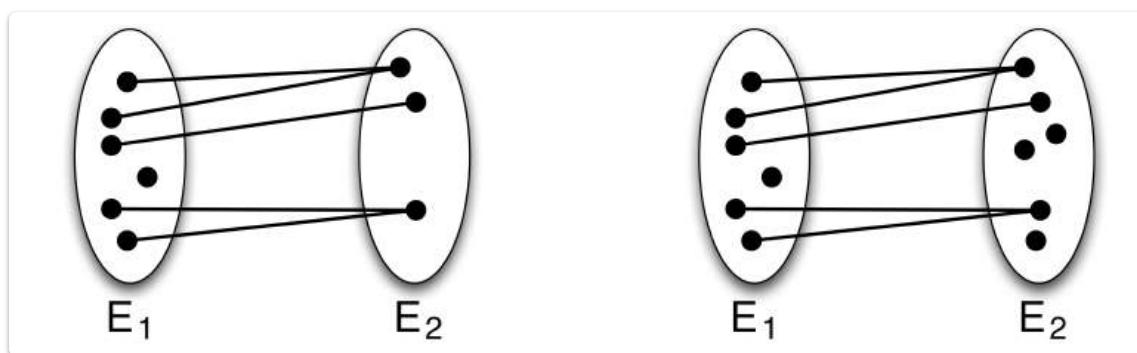
- **1:1 Beziehungstyp:**



- Hier besitzt ein Produzent genau eine Lizenz und eine Lizenz gehört genau einem Produzenten.

## Participation Constraints

- **Total (totale Partizipation):**
  - Jedes Entity eines Entitätstyps **muss** an einer Beziehung teilnehmen. Es kann nicht existieren, ohne zu partizipieren.
  - Dies wird oft durch eine **Doppellinie** dargestellt.
- **Partiell (partielle Partizipation):**
  - Jedes Entity eines Entitätstyps **kann** an einer Beziehung teilnehmen. Es kann existieren, ohne zu partizipieren.
  - Dies wird oft durch eine **Einzellinie** dargestellt.



## Graphische Darstellung von Participation Constraints

- **1:N Beziehungstyp mit totaler Partizipation vom Entitätstyp "wine":**

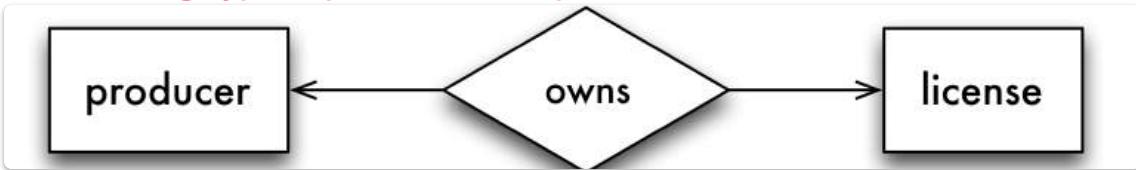


- Das bedeutet, jeder Wein muss von einem Produzenten produziert werden.
- **1:N Beziehungstyp mit totaler Partizipation von beiden Entitätstypen ("producer" und "wine"):**



- Das bedeutet, jeder Produzent muss Weine produzieren und jeder Wein muss von einem Produzenten produziert werden.

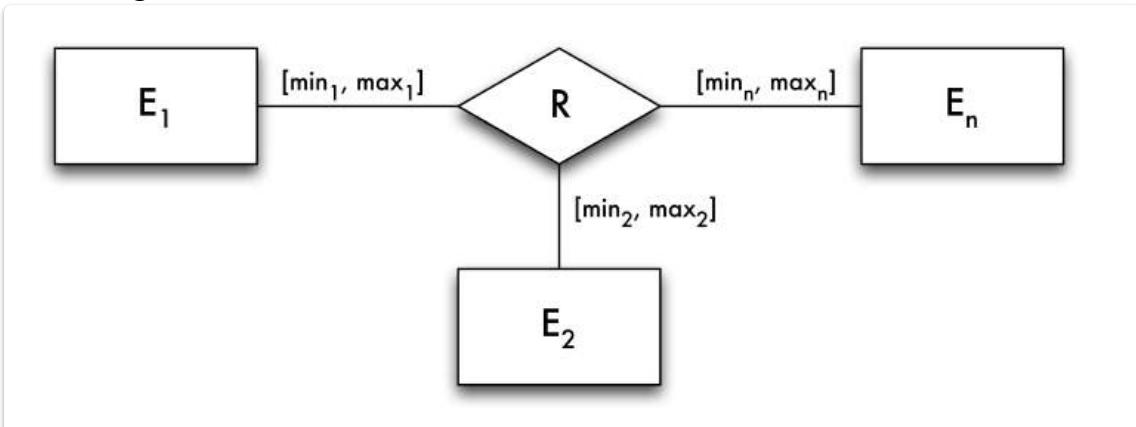
- **1:1 Beziehungstyp mit partieller Partizipation:**



- Das bedeutet, ein Produzent kann eine Lizenz besitzen (muss aber nicht) und eine Lizenz kann von einem Produzenten besessen werden (muss aber nicht).

## [min, max]-Notation (Kardinalität)

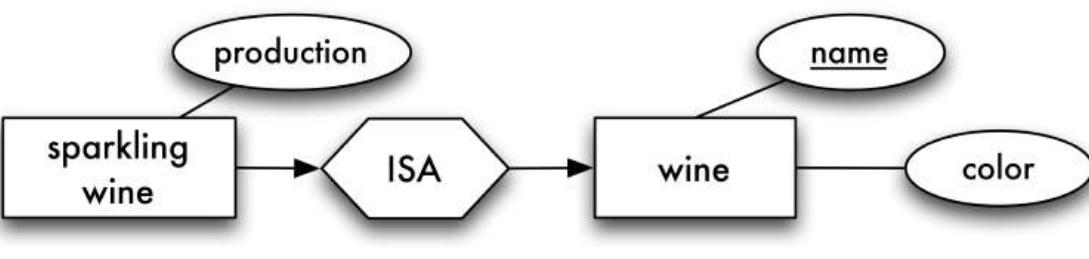
- Die  $[min, max]$ -Notation schränkt ein, wie oft ein Entity eines Entitätstyps an einer Beziehung teilnehmen kann.



- Für jedes  $e_i \in E_i$  gibt es:
  - **mindestens**  $min_i$  Instanzen des Beziehungstyps der Art  $(\dots, e_i, \dots)$  und
  - **höchstens**  $max_i$  viele Instanzen des Beziehungstyps der Art  $(\dots, e_i, \dots)$ .
- Die Kardinalitätsbedingung ist also:  $min_i \leq |\{r \in R \mid r \text{ enthält } e_i\}| \leq max_i$ .
  - Das bedeutet, die Anzahl der Beziehungen, an denen ein bestimmtes Entity  $e_i$  beteiligt ist, muss zwischen  $min_i$  und  $max_i$  liegen.
- Spezielle Wertangabe für  $min_i$ : 0
  - Bedeutet, dass ein Entity dieses Typs nicht an der Beziehung teilnehmen muss (partielle Partizipation).
- Spezielle Wertangabe für  $max_i$ : \* (oder N)
  - Bedeutet, dass ein Entity dieses Typs an beliebig vielen Beziehungen teilnehmen kann.
- Wenn  $min_i = 1$ , bedeutet das **totale Partizipation**.
- Wenn  $max_i = 1$ , bedeutet das eine "1"-Seite in der Chen-Notation.

## Der ISA-Beziehungstyp (Spezialisierung/Generalisierung)

- **Spezialisierung und Generalisierung** (Vererbung) wird durch den **ISA-Beziehungstyp** ausgedrückt.



- Hier ist "sparkling wine" eine Spezialisierung von "wine" (d.h. Sekt ist eine Art Wein).
- Attribute von "wine" werden an "sparkling wine" vererbt.
- "sparkling wine" kann zusätzliche, spezifische Attribute haben ("production").

## Relationen aus Grundkonzepten ableiten

### Entwurfsanmerkungen für ER-Modelle

- **Entitys** entsprechen Substantiven.
- **Beziehungen** entsprechen Verben.
- Jede Aussage in den Anforderungen sollte sich im ER-Schema widerspiegeln.
- Jedes ER-Diagramm (ERD) sollte sich in den Anforderungen wiederfinden.
- Ein konzeptueller Entwurf kann Inkonsistenzen und Mehrdeutigkeiten in den Anforderungen aufdecken, die zuerst geklärt werden müssen.

### Relationale Modellierung aus ER-Modell

#### Entitätstypen

- **student**: {{ studID: integer, name: string, semester: integer }}
- **course**: {{ courseID: integer, title: string, ects: integer }}
- **professor**: {{ empID: integer, name: string, rank: string, office: integer }}
- **assistant**: {{ empID: integer, name: string, department: string }}

#### Grundsätzliches Vorgehen für Entitätstypen

Für jeden Entitätstyp wird eine Relation erstellt:

- Name des Entitätstyps → Name der Relation
- Attribute des Entitätstyps → Attribute der Relation
- Schlüssel des Entitätstyps → Schlüssel der Relation

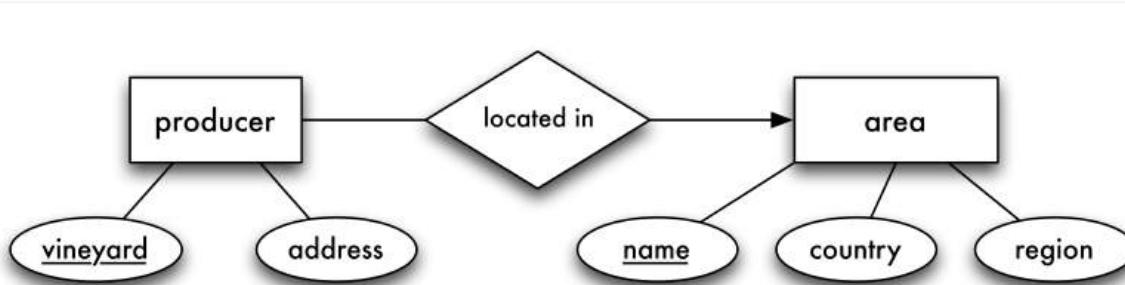
#### Notation von Relationenschemata

Beispiel:

- **student** (studID, name, semester)
- **student**: {{ studID, name, semester }}

Die Reihenfolge der Attribute ist in diesem Kontext egal. Die Domäne der Attribute ist im Moment auch nicht wichtig.

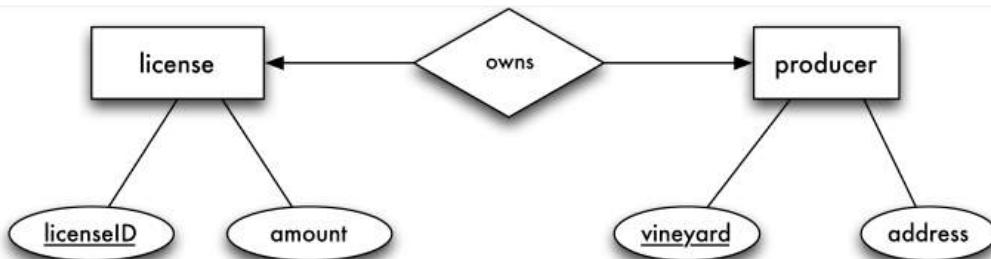
## N:1



Wie bildet man dieses ER-Diagramm auf Relationen ab?

- **producer:** {[ vineyard, address, locatedIn → area ]}
- **area:** {[ name, country, region ]}

## 1:1



- Neues Relationschema mit allen Attributen des Beziehungstyps
- Übernahme aller Primärschlüssel der beteiligen Entitytypen
- Irgendein Primärschlüssel der involvierten Entitytypen wird der Primärschlüssel im neuen Relationenschema

## Initialentwurf

- **license:** {[ licenseID, amount ]}
- **producer:** {[ vineyard, address ]}
- **owns:** {[ licenseID → license, vineyard → producer ]} oder  
**owns:** {[ licenseID → license, vineyard → producer ]}

Verbesserung durch Zusammenfassung

- **license:** {[ licenseID, amount, ownedBy → producer ]}
- **producer:** {[ vineyard, address ]}

oder

- **license:** {[ licenseID, amount ]}
- **producer:** {[ vineyard, address, ownsLicense → license ]}

## Überblick der Schritte

- **Regulärer Entitätstyp:** Relation erstellen, spezielle Attributtypen beachten.
- **Schwacher Entitätstyp:** Relation erstellen, Primärschlüssel aus partiellem Schlüssel und Fremdschlüssel des starken Entitätstyps bilden.
- **1:1 binärer Beziehungstyp:** Erweitern einer Relation mit Fremdschlüssel.
- **1:N binärer Beziehungstyp:** Erweitern einer Relation mit Fremdschlüssel.
- **M:N Beziehungstyp:** Erstellen einer neuen Relation.
- **N-ärer Beziehungstyp:** Erstellen einer neuen Relation.

## Generalisierung

### Arten um dieses ER-Diagramm auf Relationen abzubilden

#### Alternative 1: Hauptklassen

Ein bestimmtes Entity wird abgebildet als *ein Tupel* in einer einzigen Relation (zur zugehörigen Hauptklasse). (Das bedeutet, alle Attribute der Subklassen werden in die Relation der Superklasse integriert. Nicht zutreffende Attribute sind dann NULL.)

- `employee: {{ empID, name, rank, office, department }}` (Die `employee`-Tabelle enthält alle Attribute der Subklassen. Wenn ein Mitarbeiter z.B. kein Professor ist, wären `rank` und `office` NULL.)
- `professor: {{ empID, name, rank, office }}` (Alternativ könnte auch eine separate Tabelle für `professor` erstellt werden, die den Primärschlüssel der `employee`-Tabelle enthält und die spezifischen Attribute von `professor`.)
- `assistant: {{ empID, name, department }}` (Ähnlich wie bei `professor`, eine separate Tabelle für `assistant` mit dem Primärschlüssel der `employee`-Tabelle und den spezifischen Attributen von `assistant`.)

employee	
empID	name
2123	P. Müller
2124	A. Schmidt

employee:  
{{ empID, name }}

professor			
empID	name	rank	office
2125	Socrates	C4	226
2126	Russel	C3	232
2127	Kopernikus	C3	310
2128	Curie	C4	36

professor:  
{{ empID, name, rank, office }}

assistant		
empID	name	department
2150	C. Meyer	DBS
2151	B. Fischer	Physics

assistant:  
{{ empID, name, department }}

## Alternative 2: Partitionierung

Teile eines bestimmten Entitys werden in *mehreren Relationen* abgebildet, der Schlüssel wird dupliziert. (Das bedeutet, die Superklasse hat eine Relation, und jede Subklasse hat eine eigene Relation, die jeweils den Primärschlüssel der Superklasse als Fremdschlüssel enthält.)

- employee: {{ empID, name }} (Diese Tabelle enthält die gemeinsamen Attribute aller Mitarbeiter.)
- professor: {{ empID -> employee, rank, office }} (Die professor -Tabelle enthält den Fremdschlüssel empID (der auf employee verweist) und die spezifischen Attribute rank und office . empID ist hier auch der Primärschlüssel.)
- assistant: {{ empID -> employee, department }} (Die assistant -Tabelle enthält den Fremdschlüssel empID (der auf employee verweist) und das spezifische Attribut department . empID ist hier auch der Primärschlüssel.)

employee	
empID	name
2123	P. Müller
2124	A. Schmidt
2125	Socrates
...	...
2150	C. Meyer
2151	B. Fischer

employee:  
 $\{ [ \underline{\text{empID}}, \text{name} ] \}$

professor		
empID	rank	office
2125	C4	226
...	...	...

professor:  
 $\{ [ \underline{\text{empID}} \rightarrow \text{employee}, \text{rank}, \text{office} ] \}$

assistant	
empID	department
2150	DBS
2151	Physics

assistant:  
 $\{ [ \underline{\text{empID}} \rightarrow \text{employee}, \text{department} ] \}$

### Alternative 3: Vollständige Redundanz

Ein bestimmtes Entity wird *redundant* in mehreren Relationen gespeichert inklusive aller geerbten Attribute. (Jede Subklasse und die Superklasse erhalten eigene Relationen, die alle relevanten Attribute, einschließlich der geerbten, enthalten. Dies führt zu Redundanz, da dieselben Informationen mehrfach gespeichert werden.)

- employee:  $\{ [ \underline{\text{empID}}, \text{name} ] \}$
- professor:  $\{ [ \underline{\text{empID}}, \text{name}, \text{rank}, \text{office} ] \}$  (Die professor -Tabelle enthält alle Attribute, die für einen Professor relevant sind, inklusive der geerbten von employee .)
- assistant:  $\{ [ \underline{\text{empID}}, \text{name}, \text{department} ] \}$  (Die assistant -Tabelle enthält alle Attribute, die für einen Assistenten relevant sind, inklusive der geerbten von employee .)

employee	
empID	name
2123	P. Müller
2124	A. Schmidt
2125	Socrates
...	...
2150	C. Meyer
2151	B. Fischer

employee:  
{{ empID, name }}

professor			
empID	name	rank	office
2125	Socrates	C4	226
...	...	...	...

professor:  
{{ empID, name, rank, office }}

assistant		
empID	name	department
2150	C. Meyer	DBS
2151	B. Fischer	Physics

assistant:  
{{ empID, name, department }}

#### Alternative 4: Eine einzige Relation

Alle Entitys werden in *einer einzigen Relation* gespeichert und ein besonderes Attribut hinzugefügt, welches die Zugehörigkeit zu einem bestimmten Subtyp angibt. (Dies ist eine gängige Methode, bei der eine einzige breite Tabelle für die Superklasse erstellt wird, die alle Attribute aller Subklassen enthält. Ein zusätzliches "Typ"-Attribut gibt an, zu welcher spezifischen Subklasse das jeweilige Tupel gehört.)

- employee: {{ empID, name, type, rank, office, department }} (Das type-Attribut könnte Werte wie 'employee', 'professor' oder 'assistant' enthalten. Attribute, die für einen bestimmten Typ nicht zutreffen, sind NULL.)

employee: {{ empID, name, type, rank, office, department }}

employee					
empID	name	type	rank	office	department
2123	P. Müller	employee	⊥	⊥	⊥
2124	A. Schmidt	employee	⊥	⊥	⊥
2125	Socrates	professor	C4	226	⊥
2126	Russel	professor	C3	232	⊥
2127	Kopernikus	professor	C3	310	⊥
2128	Curie	professor	C4	36	⊥
2150	C. Meyer	assistant	⊥	⊥	DBS
2151	B. Fischer	assistant	⊥	⊥	Physics

#### Allgemeine Lösung

Im Allgemeinen ist **Alternative 2 (Partitionierung)** zu verwenden.

Teile eines bestimmten Entitys werden in mehreren Relationen abgebildet, der Schlüssel wird dupliziert. (Diese Alternative wird bevorzugt, da sie eine gute Balance zwischen Redundanz und Komplexität bietet und NULL-Werte minimiert.)

- `employee: {{ empID, name }}`
- `professor: {{ empID -> employee, rank, office }}` (Der Primärschlüssel `empID` der `professor`-Relation ist gleichzeitig ein Fremdschlüssel, der auf die `employee`-Relation verweist.)
- `assistant: {{ empID -> employee, department }}` (Ähnlich ist `empID` der `assistant`-Relation Primär- und Fremdschlüssel zu `employee`.)

## 3. Normalization

### Info

Anomalien = unerwünschte Nebeneffekte

### Hauptquellen für Anomalien

- **Redundanz:** Informationen sind mehrfach in der Datenbank gespeichert.
- **Speicherplatzverschwendungen:** Durch redundante Daten oder unnötige Null-Werte (`NULL`) wird Speicherplatz ineffizient genutzt.
- **Unvollständige Beziehungen:** Existenz von Entitäten ohne die dazugehörigen abhängigen Informationen, z.B. Professor:innen ohne Lehrveranstaltungen (LVAs) oder LVAs ohne zugeordnete Professor:innen.

### Arten von Anomalien

#### 1. Änderungsanomalien (*Update Anomaly*)

- **Problem:** Wenn eine Information, die mehrfach in der Datenbank vorkommt, an einer Stelle geändert wird, aber nicht an allen anderen Stellen, führt dies zu Inkonsistenzen.
- **Beispiel:** Sokrates zieht von Raum 226 in Raum 338. Wenn diese Änderung nur für einige seiner Lehrveranstaltungen (aber nicht für alle) aktualisiert wird, sind die Daten inkonsistent.

#### 2. Einfügeanomalien (*Insertion Anomaly*)

- **Problem:** Eine neue LVA (Lehrveranstaltung) kann nicht eingefügt werden, ohne gleichzeitig Null-Werte (*NULL values*) für Attribute zu setzen, die eigentlich nicht unbekannt sein sollten (z.B. ein zugehöriger Professor:in).
- **Beispiel:** Ein neuer Kurs kann nicht erfasst werden, wenn der zugehörige Professor noch nicht existiert oder zugewiesen ist, obwohl der Kurs selbst schon bekannt ist.

#### 3. Löschanomalien (*Deletion Anomaly*)

- **Problem:** Das Löschen einer bestimmten Information führt zum Verlust von weiteren, eigentlich unabhängigen Informationen.
- **Beispiel:** Wenn die letzte LVA von einem: einer Professor:in gelöscht wird, gehen alle Informationen über diese:n Professor:in (Name, ID, Rang, Büro) verloren.

## Ziel beim Datenbankentwurf

- **Vermeidung von Redundanz:** Informationen sollen nur einmal gespeichert werden.
- **Vermeidung von Null-Werten (*NULL values*):** Minimierung von Attributen, die bei der Erfassung leer bleiben müssen.
- **Vermeidung von Anomalien:** Sicherstellen der Konsistenz und Integrität der Daten bei allen Operationen.
- **Abbildung von Beziehungen:** Alle Beziehungen zwischen Attributen müssen korrekt und eindeutig abgebildet sein.

## Funktionale Abhängigkeiten

### Symbole

- Schema  $\mathcal{R} = \{A, B, C, D\}$  (Menge aller Attribute, die in einer Relation vorkommen können)
- Ausprägung/Instanz  $R$  (Eine konkrete Tabelle/Beziehung, die dem Schema  $\mathcal{R}$  entspricht)
- Attributmengen  $\alpha \subseteq \mathcal{R}$  und  $\beta \subseteq \mathcal{R}$  (Teilmengen der Attribute aus dem Schema)

#### ⌚ Definition einer funktionalen Abhängigkeit

Eine **funktionale Abhängigkeit**  $\alpha \rightarrow \beta$  in  $R$  liegt genau dann vor, wenn für alle legalen Ausprägungen  $R$  von  $\mathcal{R}$  gilt:

$$\forall r, s \in R : r. \alpha = s. \alpha \Rightarrow r. \beta = s. \beta$$

- Das bedeutet: Wenn zwei Tupel (Zeilen)  $r$  und  $s$  in einer Relation  $R$  in ihren Werten für die Attributmenge  $\alpha$  übereinstimmen ( $r. \alpha = s. \alpha$ ), dann müssen sie auch in ihren Werten für die Attributmenge  $\beta$  übereinstimmen ( $r. \beta = s. \beta$ ).
- Die  $\alpha$ -Werte identifizieren die  $\beta$ -Werte eindeutig.
- Bzw.: Die  $\alpha$ -Werte bestimmen die  $\beta$ -Werte funktional.

## Triviale funktionale Abhängigkeit

Eine funktionale Abhängigkeit  $\alpha \rightarrow \beta$  wird **trivial** genannt, wenn  $\beta \subseteq \alpha$ .

- **Beispiel:** Wenn wir die Abhängigkeit  $\{A, B\} \rightarrow A$  betrachten, ist dies trivial, da  $A$  bereits Teil von  $\{A, B\}$  ist. Dies ist immer wahr und liefert keine neue Information über die Datenintegrität.

## Semantische Konsistenzbedingungen

Funktionale Abhängigkeiten sind **semantische Konsistenzbedingungen**, die sich aus der **jeweiligen Anwendungssemantik** und **nicht aus der aktuellen Ausprägung einer Relation** ergeben. Sie müssen zu jedem (gültigen) Datenbankzustand eingehalten werden.

## Schlüssel

### ⌚ Superschlüssel

- $\alpha \subseteq \mathcal{R}$  ist ein **Superschlüssel** wenn  $\alpha \rightarrow \mathcal{R}$ .
  - D.h.,  $\alpha$  bestimmt alle anderen Attributwerte in der Relation.
  - Ein Superschlüssel ist also eine Attributmenge, die die gesamte Relation eindeutig identifizieren kann.
- Die Menge aller Attribute bildet einen Superschlüssel:  $\mathcal{R} \rightarrow \mathcal{R}$ .
  - Wenn man alle Attribute kennt, kann man natürlich auch die gesamte Relation eindeutig identifizieren.
- Superschlüssel sind **nicht notwendigerweise minimal!**
  - Ein Superschlüssel kann also überflüssige Attribute enthalten, die nicht zwingend zur Eindeutigkeit der Identifizierung notwendig wären.

### ⓘ Volle funktionale Abhängigkeit

$\beta$  ist **voll funktional abhängig** von  $\alpha$  wenn:

- $\alpha \rightarrow \beta$  (Es besteht eine funktionale Abhängigkeit von  $\alpha$  zu  $\beta$ .)
- und  $\alpha$  kann nicht mehr verkleinert (=linksreduziert) werden, d.h.

$$\forall A \in \alpha : (\alpha - \{A\}) \not\rightarrow \beta$$

- Das bedeutet, dass kein Attribut aus  $\alpha$  entfernt werden kann, ohne dass die funktionale Abhängigkeit zu  $\beta$  verloren geht.  $\alpha$  ist also die **minimale** Menge von Attributen, die  $\beta$  bestimmt.

### ⌚ Schlüsselkandidat

$\alpha \subseteq \mathcal{R}$  ist ein **Schlüsselkandidat**, wenn  $\mathcal{R}$  **voll funktional abhängig** von  $\alpha$  ist.

- Ein Schlüsselkandidat ist also ein **minimaler** Superschlüssel. Das bedeutet, er ist ein Superschlüssel, und wenn man irgendein Attribut aus ihm entfernt, ist er kein Superschlüssel mehr.
- Ein Schlüsselkandidat wird als **Primärschlüssel** ausgewählt!

- Aus der Menge aller Schlüsselkandidaten wird in der Regel einer als Primärschlüssel für eine Relation bestimmt. Dieser Primärschlüssel dient dann zur eindeutigen Identifikation der Tupel in der Relation.

## Ableitung funktionaler Abhängigkeiten

### Bestimmung funktionaler Abhängigkeiten

Hier sind einige Beispiele für funktionale Abhängigkeiten und wie daraus weitere abgeleitet werden können:

- $\{empID\}$   
 $\rightarrow \{empID, name, rank, office, city, street, zip, dialCode, region, inhabitants, government\}$
- $\{city, region\} \rightarrow \{inhabitants, dialCode\}$
- $\{zip\} \rightarrow \{region, city, inhabitants\}$
- $\{region, city, street\} \rightarrow \{zip\}$
- $\{region\} \rightarrow \{government\}$
- $\{office\} \rightarrow \{empID\}$

### Davon können weitere abgeleitet werden

Basierend auf den oben genannten funktionalen Abhängigkeiten können durch Inferenzregeln (z.B. Transitivität) weitere Abhängigkeiten abgeleitet werden:

- $\{office\} \rightarrow \{empID, name, rank, office, city, street, zip, dialCode, inhabitants, government\}$
- $\{zip\} \rightarrow \{government\}$

### Die Armstrong-Axiome

Seien  $\alpha, \beta, \gamma, \delta$  Teilmengen der Attribute aus  $R$ .

#### Armstrong Axiome

- **Reflexivität:**
  - Falls  $\beta \subseteq \alpha$ , dann  $\alpha \rightarrow \beta$ .
  - Insbesondere:  $\alpha \rightarrow \alpha$  (Eine Menge von Attributen determiniert sich selbst).
- **Erweiterung/Verstärkung:**
  - Falls  $\alpha \rightarrow \beta$ , dann  $\alpha\gamma \rightarrow \beta\gamma$ . (Wenn  $\alpha$   $\beta$  determiniert, dann determiniert  $\alpha$  zusammen mit weiteren Attributen  $\gamma$  auch  $\beta$  zusammen mit  $\gamma$ ).
- **Transitivität:**
  - Falls  $\alpha \rightarrow \beta$  und  $\beta \rightarrow \gamma$ , dann  $\alpha \rightarrow \gamma$ . (Wenn  $\alpha$   $\beta$  determiniert und  $\beta$   $\gamma$  determiniert, dann determiniert  $\alpha$  auch  $\gamma$ ).
- Die Armstrong-Axiome sind **korrekt und vollständig**:

- Sie sind **korrekt** in dem Sinne, dass sich mit ihnen nur korrekte funktionale Abhängigkeiten herleiten lassen.
- Sie sind **vollständig** in dem Sinne, dass sich mit ihnen **alle möglichen FDs ( $F^+$ ) von  $F$**  herleiten lassen.

### ① Weitere Ableitungsregeln

Nicht zwingend notwendig, aber oftmals komfortabel für Herleitungen.

- **Vereinigung:**
  - Wenn  $\alpha \rightarrow \beta$  und  $\alpha \rightarrow \gamma$ , dann auch  $\alpha \rightarrow \beta\gamma$ . (Wenn  $\alpha$  sowohl  $\beta$  als auch  $\gamma$  determiniert, dann determiniert  $\alpha$  auch die Vereinigung von  $\beta$  und  $\gamma$ ).
- **Dekomposition:**
  - Wenn  $\alpha \rightarrow \beta\gamma$ , dann  $\alpha \rightarrow \beta$  und  $\alpha \rightarrow \gamma$ . (Wenn  $\alpha$  die Vereinigung von  $\beta$  und  $\gamma$  determiniert, dann determiniert  $\alpha$  auch  $\beta$  und  $\gamma$  einzeln).
- **Pseudotransitivität:**
  - Wenn  $\alpha \rightarrow \beta$  und  $\gamma\beta \rightarrow \delta$ , dann auch  $\alpha\gamma \rightarrow \delta$ .

## Algorithmus zur Bestimmung der Attributhülle

**Input:**

- Eine Menge von FDs  $F$
- Eine Menge von Attributen  $\alpha \subseteq R$  (Teilmenge aller Attribute des Schemas  $R$ )

**Algorithmus-Schritte:**

Algorithmus attrClosure( $F, \alpha$ ):

```

result =  $\alpha$ 
repeat
  for each  $\beta \rightarrow \gamma$  in  $F$  do
    if  $\beta \subseteq \text{result}$  then
      result = result  $\cup$   $\gamma$ 
    end if
  end for
until result changes no more
  
```

## Kanonische Überdeckung

## Äquivalente FD-Mengen

- Zwei Mengen von FDs  $F$  und  $G$  werden als *äquivalent* angesehen ( $F \equiv G$ ), wenn deren Hüllen gleich sind, d.h.,  $F^+ = G^+$ .
- Beide Mengen erlauben die *gleiche Menge von FDs*.

### Beobachtung:

- $F^+$  kann **riesig** sein.
- Viele *redundante Abhängigkeiten* (d.h. Abhängigkeiten, die aus anderen in  $F$  abgeleitet werden können).
- In der Praxis unübersichtlich.

### Ziel:

- Finde die **kleinstmögliche Menge**  $F_c$  für  $F$ , so dass  $F_c^+ = F^+$ .
- Es gibt möglicherweise **mehrere minimale Mengen**!

## Kanonische Überdeckung $F_c$ (minimale Überdeckung)

Eine *minimale Überdeckung*  $F_c$  ist eine **kanonische Darstellung** einer Menge  $F$  von funktionalen Abhängigkeiten.

### Eigenschaften:

1.  $F_c \equiv F$ , also  $F_c^+ = F^+$  (Äquivalenz, falls die Hüllen gleich sind).
2. In  $F_c$  existieren **keine FDs  $\alpha \rightarrow \beta$ , bei denen  $\alpha$  oder  $\beta$  überflüssige Attribute enthalten**.  
D.h. es muss gelten:
  - (a)  $\forall A \in \alpha: (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\} \not\equiv F_c$  (Das Entfernen eines Attributs aus der linken Seite oder das Ersetzen von  $\alpha$  durch  $\alpha - A$  verändert die Hülle von  $F_c$ )
  - (b)  $\forall B \in \beta: (F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\} \not\equiv F_c$  (Das Entfernen eines Attributs aus der rechten Seite oder das Ersetzen von  $\beta$  durch  $\beta - B$  verändert die Hülle von  $F_c$ )
  - **Überprüfung mit Hilfe der Attributhülle, ob  $A \in \alpha$  in  $\alpha \rightarrow \beta$  überflüssig ist:**
    - $A \in \alpha$  ist überflüssig, wenn  $\beta \subseteq \text{attrClosure}((F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}, \alpha)$  (d.h., wenn  $\beta$  auch ohne  $A$  auf der linken Seite in Kombination mit den restlichen FDs abgeleitet werden kann).
  - **Überprüfung mithilfe der Attributhülle, ob  $B \in \beta$  in  $\alpha \rightarrow \beta$  überflüssig ist:**
    - $B \in \beta$  ist überflüssig, wenn  $B \in \text{attrClosure}((F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\}, \alpha)$  (d.h., wenn  $B$  auch dann abgeleitet werden kann, wenn es nicht auf der rechten Seite von  $\alpha \rightarrow \beta$  steht, sondern aus den verbleibenden FDs in  $F$  und der modifizierten FD  $\alpha \rightarrow (\beta - B)$ ).
3. Jede linke Seite einer funktionalen Abhängigkeit in  $F_c$  ist **einzigartig**.
  - Durch Anwendung der *Vereinigungsregel* wird  $(\alpha \rightarrow \beta)$  und  $(\alpha \rightarrow \gamma)$  durch  $\alpha \rightarrow \beta\gamma$  ersetzt. (Dies konsolidiert alle FDs mit gleicher linker Seite).

## Algorithmus minimale Überdeckung

### Hauptschritte:

#### 1. Führe FDs mit der gleichen linken Seite zusammen:

- $\alpha \rightarrow \beta_1, \dots, \alpha \rightarrow \beta_n$  wird zu  $\alpha \rightarrow (\beta_1 \cup \dots \cup \beta_n)$  (Anwendung der Vereinigungsregel, um die linke Seite einzigartig zu machen).

#### 2. Für jede FD $(\alpha \rightarrow \beta) \in F$ führe Linksreduktion durch:

- Überprüfe für jedes  $A \in \alpha$ , ob  $A$  überflüssig ist, d.h., ob  $\beta \subseteq \text{attrClosure}(F, \alpha - A)$ .
  - (*Erklärung:*  $A$  ist überflüssig, wenn die Attribute auf der rechten Seite  $\beta$  immer noch aus  $\alpha$  (ohne  $A$ ) abgeleitet werden können, unter Berücksichtigung aller anderen FDs in  $F$ ).
- Wenn ja, entferne  $A$ , indem  $\alpha \rightarrow \beta$  durch  $(\alpha - A) \rightarrow \beta$  ersetzt wird.

#### 3. Für jede FD $(\alpha \rightarrow \beta) \in F$ führe Rechtsreduktion durch:

- Überprüfe für jedes  $B \in \beta$ , ob  $B$  überflüssig ist, d.h., ob  $B \in \text{attrClosure}((F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\}, \alpha)$ .
  - (*Erklärung:*  $B$  ist überflüssig, wenn  $B$  auch dann in der Attributhülle von  $\alpha$  ist, wenn die FD  $\alpha \rightarrow \beta$  durch  $\alpha \rightarrow (\beta - B)$  (also ohne  $B$  auf der rechten Seite) ersetzt wird. Dies bedeutet, dass  $B$  durch andere FDs impliziert wird oder bereits in  $\alpha$  enthalten ist.)
- Wenn ja, entferne  $B$ , indem  $\alpha \rightarrow \beta$  durch  $\alpha \rightarrow (\beta - B)$  ersetzt wird.

#### 4. Entferne FDs der Form $\alpha \rightarrow \emptyset$ (sind möglicherweise in Schritt 3 entstanden, wenn alle Attribute auf der rechten Seite als überflüssig entfernt wurden).

#### 5. Gehe zu Schritt 1 (Wiederhole die Schritte, da eine Änderung in einem Schritt Auswirkungen auf die Überflüssigkeit in einem anderen haben kann).

- *Im Allgemeinen ist eine Runde (Schritte 1-4) plus ein weiteres Mal Schritt 1 genug.*

## Gültige und verlustlose Zerlegungen

---

- Zerlege ein Relationenschema  $R$  in mehrere Relationsschemata  $R_1, \dots, R_m$ , um Probleme des originalen Entwurfs zu beheben.

## Gültige und verlustlose Zerlegungen

Eine Zerlegung ist **gültig**, falls  $R = R_1 \cup R_2$ , d.h., wenn alle Attribute aus  $R$  in der Zerlegung enthalten bleiben.

- $R_1 := \pi_{R_1}(R)$
- $R_2 := \pi_{R_2}(R)$

Eine Zerlegung von  $R$  in  $R_1$  und  $R_2$  ist **verlustlos**, wenn Folgendes für alle möglichen Ausprägungen  $R$  von  $R$  gilt:

$$R = R_1 \bowtie R_2$$

Die in der ursprünglichen Ausprägung  $R$  des Schemas  $R$  enthaltenen Informationen müssen aus den Ausprägungen  $R_1, \dots, R_n$  der neuen Schemata  $R_1, \dots, R_n$  durch einen [natürlichen Join](#) rekonstruierbar sein.

## Verlustlose Zerlegung

### Formale Charakterisierung verlustloser Zerlegungen

Gegeben:

- Eine Zerlegung von  $R$  in  $R_1$  und  $R_2$
- $F_R$  ist die Menge der Funktionalen Abhängigkeiten (FDs) in  $R$

Eine Zerlegung ist **verlustlos**, wenn mindestens eine der folgenden FDs herleitbar ist:

- $(R_1 \cap R_2) \rightarrow R_1 \in F_R^+$  (die gemeinsamen Attribute bestimmen  $R_1$ )
- oder
- $(R_1 \cap R_2) \rightarrow R_2 \in F_R^+$  (die gemeinsamen Attribute bestimmen  $R_2$ )

#### :≡ Beispielzerlegung

beerDrinkers		
pub	guest	beer
Kowalski	Kemper	pils
Kowalski	Eickler	wheat beer
Innsteg	Kemper	wheat beer

zerlegt in

customer	
pub	guest
Kowalski	Kemper
Kowalski	Eickler
Innsteg	Kemper

drinks	
guest	beer
Kemper	pils
Eickler	wheat beer
Kemper	wheat beer

Rekonstruktion:  $R_1 \bowtie R_2$

beerDrinkers		
pub	guest	beer
Kowalski	Kemper	pils
<b>Kowalski</b>	<b>Kemper</b>	<b>wheat beer</b>
Kowalski	Eickler	wheat beer
<b>Innsteg</b>	<b>Kemper</b>	<b>pils</b>
Innsteg	Kemper	wheat beer

- Die Beziehung zwischen guest, pub und beer ging verloren.

- Die Verletzung der Verlustlosigkeit kann manchmal auch bedeuten, dass bei der Wiederherstellung zusätzliche Tupel entstehen.
- Also keine verlustlose Zerlegung.

## Abhängigkeitsbewahrung

### Zweites Kriterium für eine gute Zerlegung

Die für  $R$  geltenden Abhängigkeiten müssen in den neuen Schemata  $R_1, \dots, R_n$  verifizierbar sein.

- Wir können alle Abhängigkeiten lokal in  $R_1, \dots, R_n$  überprüfen.
- Wir vermeiden, dass wir den Join  $R_1 \bowtie \dots \bowtie R_n$  berechnen müssen, um überprüfen zu können, ob eine FD verletzt wird.

Eine Zerlegung bewahrt Abhängigkeiten, wenn

$F_R^+ = (F_{R_1} \cup \dots \cup F_{R_n})^+$  (Die Abschlussmenge der FDs im Originalschema ist gleich der Abschlussmenge der vereinigten FDs der Teilschemata).

$F_{R_i}$  sind die funktionalen Abhängigkeiten, die sich über  $R_i$  effizient überprüfen lassen.

Überprüfen, ob eine Zerlegung Abhängigkeiten bewahrt, ohne  $F^+$  berechnen zu müssen:

- ① Können FDs effizient über dem zerlegtem Schema getestet werden?
  - Weise FDs den Relationen zu, welche alle involvierten Attribute enthalten.  
Falls ja: die Zerlegung bewahrt Abhängigkeiten
- ② Falls nicht: verwende modifizierte Version von attrClosure um jede  $\alpha \rightarrow \beta$  in  $F$  zu testen  
Falls  $\beta \subseteq result$ , dann ist die FD bewahrt  
Wenn alle FDs bewahrt werden, dann bewahrt die Zerlegung Abhängigkeiten

$result = \alpha$

**repeat**

```

for each  $R_i$  in the decomposition do
   $t = (result \cap R_i)^+ \cap R_i$ 
   $result = result \cup t$ 
end for
until result changes no more
  
```

:≡ Beispiele

zipCatalog: {[ street, city, region, zip ]}

FDs

- $\{zip\} \rightarrow \{city, region\}$
- $\{street, city, region\} \rightarrow \{zip\}$

Zerlegung

- streets: {[ zip, street ]}
- cities: {[ zip, city, region ]}

Ist diese Zerlegung verlustlos?  
Bewahrt diese Zerlegung Abhängigkeiten?

zipCatalog			
city	region	street	zip
Frankfurt	Hesse	Goethestraße	60313
Frankfurt	Hesse	Galgenstraße	60437
Frankfurt	Brandenburg	Goethestraße	15234

$\pi_{zip, street}$

$\pi_{city, region, zip}$

street	
zip	street
15234	Goethestraße
60313	Goethestraße
60437	Galgenstraße

cities		
city	region	zip
Frankfurt	Hesse	60313
Frankfurt	Hesse	60437
Frankfurt	Brandenburg	15234

Die Zerlegung bewahrt die Abhängigkeiten **nicht**:

$\{street, city, region\} \rightarrow \{zip\}$  kann nicht über der Zerlegung überprüft werden und wird nicht durch andere FDs garantiert.

Gegeben

- $R = (A, B, C)$
- $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (B, C)$

Ist das eine verlustlose Zerlegung?

Hinweis (gemeinsame Attribute sind Superschlüssel in einer der Relationen):

- $(R_1 \cap R_2) \rightarrow R_1 \in F_R^+$
- $(R_1 \cap R_2) \rightarrow R_2 \in F_R^+$

**JA:**

- $R_1 \cap R_2 = \{B\}$  und  $B \rightarrow BC$

Bewahrt diese Zerlegung Abhängigkeiten?

Hinweis (wir können alle Abhängigkeiten lokal überprüfen):

$$(F_1 \cup F_2)^+ = F^+$$

**JA:**

- $F_1 = \{A \rightarrow B, A \rightarrow AB\} \cup \{\text{viele triviale Abhängigkeiten}\}$
- $F_2 = \{B \rightarrow C, B \rightarrow BC\} \cup \{\text{viele triviale Abhängigkeiten}\}$
- $(F_1 \cup F_2)^+ = F^+$

Gegeben

- $R = (A, B, C)$
- $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (\textcolor{red}{A}, C)$

Ist das eine verlustlose Zerlegung?

Hinweis (gemeinsame Attribute sind Superschlüssel in einer der Relationen):

- $(R_1 \cap R_2) \rightarrow R_1 \in F_R^+$
- $(R_1 \cap R_2) \rightarrow R_2 \in F_R^+$

**JA:**

- $R_1 \cap R_2 = \{A\}$  und  $A \rightarrow AB$

Gegeben

- $R = (A, B, C)$
- $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (\textcolor{red}{A}, C)$

Bewahrt diese Zerlegung Abhängigkeiten?

Hinweis (wir können alle Abhängigkeiten lokal überprüfen):

$$(F_1 \cup F_2)^+ = F^+$$

**NEIN:**

- $F_1 = \{A \rightarrow B, A \rightarrow AB\} \cup \{\text{viele triviale Abhängigkeiten}\}$
- $F_2 = \{\text{viele triviale Abhängigkeiten}\}$
- $(F_1 \cup F_2)^+ \neq F^+$
- $B \rightarrow C$  kann nicht überprüft werden ohne  $R_1 \bowtie R_2$  zu berechnen

## Normalformen

- Normalformen beschreiben die Güte des Entwurfs.
- 1NF, 2NF, 3NF, BCNF, 4NF, ...**
- Sie verbieten bestimmte funktionale Abhängigkeiten in einer Relation, um **Redundanz**, **Null-Werte** und **Anomalien** zu verhindern.
- Gute ER-Modelle führen typischerweise direkt zur **3NF** (oder höhere NF).
- Normalisierung verhindert Probleme, welche durch funktionale Abhängigkeiten zwischen den Attributen eines Entitytyps ausgelöst werden.

## Erste Normalform (1NF)

Eine Relation  $R$  ist in **1NF**, wenn alle Attribute atomar sind (keine zusammengesetzten, mengenwertigen oder relationenwertige Domänen).

1NF:

Nicht 1NF:

parents		
father	mother	child
Johann	Martha	{Else, Lucie}
Johann	Maria	{Theo, Josef}
Heinz	Martha	{Cleo}

parents		
father	mother	child
Johann	Martha	Else
Johann	Martha	Lucie
Johann	Maria	Theo
Johann	Maria	Josef
Heinz	Martha	Cleo

## Zweite Normalform (2NF)

Ein Relationsschema  $R$  mit FDs  $F$  ist in **2NF**, falls jedes Nicht-Primattribut  $A \in R$  voll funktional abhängig von jedem Kandidatenschlüssel der Relation ist.

- $(\kappa_j \rightarrow A) \in F^+$  und  $\kappa_j$  ist linksreduziert (**voll funktional abhängig**).

### ⌚ Prim / Nicht Prim

- Ein Attribut  $A$  ist **prim**, wenn  $A \in (\kappa_1 \cup \dots \cup \kappa_i)$  (Teil eines Kandidatenschlüssels).
- Ein Attribut  $A$  ist **nicht prim**, wenn  $A \in R - (\kappa_1 \cup \dots \cup \kappa_i)$  (nicht Teil eines Kandidatenschlüssels).

2NF verhindert partielle Abhängigkeiten:

- Es gibt keine funktionale Abhängigkeit eines Nicht-Prim-Attributs von einer Teilmenge eines Schlüssels.

2NF sorgt dafür, dass du keine Informationen in deiner Tabelle hast, die nur von einem *Teil* des Schlüssels abhängen. Jedes Attribut, das nicht Teil des Schlüssels ist, muss vom *ganzen* Schlüssel abhängen. Das macht deine Datenbank sauberer, effizienter und weniger anfällig für Fehler. Es ist der erste Schritt, um "komische" Abhängigkeiten in deiner Tabelle zu beseitigen.

---

## Dritte Normalform (3NF)

Ein Relationsschema  $R$  ist in **Dritter Normalform (3NF)**, wenn für jede für  $R$  geltende FD der Form  $\alpha \rightarrow B$  mit Attribut  $B \in R$  mindestens eine von drei Bedingungen gilt:

1.  $B \in \alpha$ , d.h. die FD ist **trivial** (Das Determinant enthält das Dependant).
2.  $\alpha$  ist ein **Superschlüssel** von  $R$ .
3.  $B$  ist **prim** (Schlüsselattribut, also Teil eines Kandidatenschlüssels).

## Eigenschaften der 3NF

- Nicht-Prim-Attribute dürfen keine anderen Attribute bestimmen.
- 3NF verhindert **partielle** und **transitive** Abhängigkeiten.
- **Ausnahme (Fall 3):** transitive Abhängigkeit mit Prim-Attributen als Endpunkte sind okay (wenn  $B$  prim ist, darf es auch transitiv von  $\alpha$  abhängen).
- 3NF "beinhaltet" 2NF (wenn eine Relation in 3NF ist, ist sie automatisch auch in 2NF).

Die 3NF verhindert **transitive Abhängigkeiten** und beinhaltet 2NF.

## Was die 3NF leistet:

- **Keine Redundanz von Informationen, die von anderen Nicht-Schlüssel-Attributen abhängen.** (Stell dir vor, du hättest eine Spalte "Postleitzahl" und daneben "Ort". Der Ort hängt von der Postleitzahl ab. Wenn die Postleitzahl selbst nicht der Schlüssel ist, dann ist das eine transitive Abhängigkeit und ein 3NF-Verstoß.)
  - **Verhindert partielle und transitive Abhängigkeiten.**
  - **Verbessert die Datenintegrität:** Weniger Anomalien beim Einfügen, Löschen und Ändern von Daten.
- 

## Boyce-Codd-Normalform (BCNF)

Ein Relationsschema  $R$  ist in **BCNF**, wenn für jede für  $R$  geltende FD der Form  $\alpha \rightarrow B$  mit Attribut  $B \in R$  mindestens eine von zwei Bedingungen gilt:

1.  $B \in \alpha$ , d.h. die FD ist **trivial**.

2.  $\alpha$  ist **Superschlüssel** von  $R$ .

Es ist das gleiche wie die **dritte Normalform** aber die letzte Bedingung fällt weg.

## Eigenschaften der BCNF

- **Unterschied zu 3NF:** Die dritte Bedingung der 3NF ( $B$  ist **prim**) fällt weg.
- Die Boyce-Codd-Normalform ist eine weitere Verschärfung der 3NF (d.h., BCNF "beinhaltet" 3NF).
- **Eliminierung transitiver Abhängigkeiten auch zu Prim-Attributen** (im Gegensatz zur 3NF, wo transitive Abhängigkeiten zu Prim-Attributen erlaubt sein können).

### ☰ Beispiel 3NF vs. BCNF

`cities: {[ city, state, govenor, inhabitants ]}`

FDs

- ✓  $\{city, state\} \rightarrow \{inhabitants\}$
- ✓  $\{state\} \rightarrow \{governor\}$
- ✓  $\{governor\} \rightarrow \{state\}$

Kandidatenschlüssel:

$\{city, state\}$  and  $\{city, governor\}$

Ist die Relation in 3NF? **JA**

- Trivialität? (Bedingung 1) – Keine der FDs ist trivial
- Ist die linke Seite ein Superschlüssel? (Bedingung 2) – FD1
- Ist die rechte Seite prim? (Bedingung 3) – FD2 und FD3

Ist die Relation in BCNF? **NEIN**

- Trivialität? (Bedingung 1) – Keine der FDs ist trivial
- Ist die linke Seite ein Superschlüssel? (Bedingung 2) – FD1
- ~~Ist die rechte Seite prim? (Bedingung 3) – FD2 und FD3~~

## BCNF Zerlegung

Es ist immer möglich, ein Relationsschema  $R$  mit FDs  $F$  in:

- **3NF Relationsschemata**  $R_1, \dots, R_n$  zu zerlegen, so dass die Zerlegung:
  - **verlustfrei** ist
  - **Abhängigkeiten bewahrt**
- **BCNF Relationsschemata**  $R_1, \dots, R_n$  zu zerlegen, so dass die Zerlegung:
  - **verlustfrei** ist

**Wichtig:** Es ist nicht immer möglich, eine BCNF-Zerlegung  $R_1, \dots, R_n$  von  $R$  zu erstellen, die alle Abhängigkeiten bewahrt.

---

## Zerlegsalgorithmus für BCNF

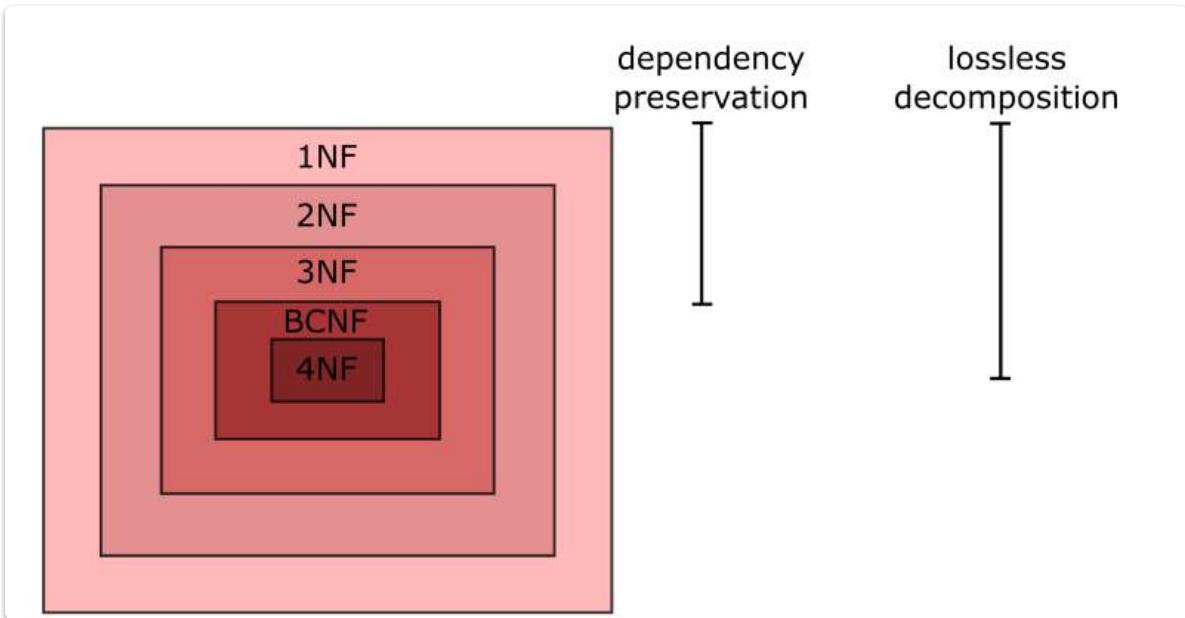
Der Algorithmus dient dazu, ein Relationenschema in BCNF zu überführen.

- Initialisierung: `result` ist eine Menge, die das ursprüngliche Relationenschema  $R$  enthält:  $\text{result} = \{R\}$ . Berechne die *Closure* (den Abschluss) aller funktionalen Abhängigkeiten  $F^+$  (aller FDs).
  - **Schleife:** Solange es ein Relationenschema  $R_i \in \text{result}$  gibt, das **nicht in BCNF** ist, führe folgende Schritte aus:
    - **Grund für Nicht-BCNF:** Es existiert eine für  $R_i$  geltende nicht-triviale funktionale Abhängigkeit (FD)  $\alpha \rightarrow \beta$  mit:
      - $\alpha \cap \beta = \emptyset$  (Dies bedeutet, die Attribute in  $\alpha$  und  $\beta$  sind disjunkt).
      - $\alpha \rightarrow R_i \notin F^+$  (Dies bedeutet,  $\alpha$  ist **kein Superschlüssel** von  $R_i$ ).
        - (*Erklärung: Ein Superschlüssel ist eine Menge von Attributen, die alle anderen Attribute in einem Relationenschema eindeutig identifiziert. Wenn  $\alpha$  ein Superschlüssel wäre, wäre die Relation in BCNF bzgl. dieser FD.*)
    - **Zerlegung:** Zerlege  $R_i$  in zwei neue Relationenschemata  $R_{i1}$  und  $R_{i2}$ :
      - $R_{i1} := \alpha \cup \beta$
      - $R_{i2} := R_i - \beta$  (Alle Attribute von  $R_i$ , außer denen in  $\beta$ )
    - **Aktualisierung des Ergebnisses:** Ersetze  $R_i$  in der `result`-Menge durch die beiden neuen Relationenschemata:
      - $\text{result} := (\text{result} - R_i) \cup R_{i1} \cup R_{i2}$
  - **Hinweis zur Superschlüsselprüfung:** Um zu überprüfen, ob  $\alpha$  ein Superschlüssel ist, kann auch  $\alpha^+$  (der Attributabschluss von  $\alpha$ ) berechnet werden, anstatt  $F^+$ . Wenn  $\alpha^+$  alle Attribute von  $R_i$  enthält, ist  $\alpha$  ein Superschlüssel von  $R_i$ .
- 

## Übersicht

Normalform	Verhindert (Hauptproblem)	Wie es (minimal) verhindert wird
1NF (Erste)	Atomare Werte / Wiederholende Gruppen	Jede Zelle enthält nur einen Wert; keine Listen oder Tabellen in Zellen. Eine Spalte für jeden Fakt.

Normalform	Verhindert (Hauptproblem)	Wie es (minimal) verhindert wird
2NF (Zweite)	Partielle Abhängigkeiten	Nicht-Primärattribute müssen vom <i>gesamten</i> Primärschlüssel abhängen, nicht nur von einem Teil davon. (Oft: Aufteilung in 2 Tabellen)
3NF (Dritte)	Transitive Abhängigkeiten (von Nicht-Primärattributen)	Nicht-Primärattribute dürfen nicht von <i>anderen Nicht-Primärattributen</i> abhängen. (Oft: Aufteilung in weitere Tabellen)
BCNF (Boyce-Codd)	Spezielle transitive Abhängigkeiten (auch mit Primärattributen)	Jede Spalte, die eine andere Spalte bestimmt, muss selbst ein <i>Superschlüssel</i> sein. Strenger als 3NF, beseitigt auch "seltene" Redundanzen.



## Denormalisierung

- Prozess, bei dem die Normalisierung „zurückgenommen“ wird, um Anfragen schneller bearbeiten zu können.
  - **Intuitivere Erklärung:** Manchmal wird bewusst eine Redundanz in der Datenbank zugelassen, um die Performance bei Lesezugriffen zu verbessern, auch wenn dies zu den Nachteilen der Normalisierung (Redundanz, Anomalien) führen kann. Dies ist ein Kompromiss zwischen Datenintegrität und Abfragegeschwindigkeit.

## 4. SQL

### Disclaimer

Ich fokussiere mich hier **nur** auf den Stoff der auch zur Prüfung angeblich kommt, also auf alles außer dem, was schon beim SQL Exam war.

## Einleitung

SQL ist eine **deklarative Anfragesprache** ("was" nicht "wie").

### Bestandteile von SQL

SQL setzt sich aus mehreren Teilen zusammen:

- **Datenbeschreibungssprache (Data Definition Language, DDL):**
  - Erstellt/ändert das Schema
  - `create, alter, drop`
- **Datenmanipulationssprache (Data Manipulation Language, DML):**
  - Ändert Datensammlungen
  - `insert, update, delete`
- **Anfragesprache (Data Query Language, DQL):**
  - Formuliert Anfragen auf den Ausprägungen
  - `select * from where ...`
- **Ablaufsteuerungssprache (Transaction Control Language, TCL):**
  - Steuert Transaktionen
  - `commit, rollback`
- **Datenaufsichtssprache (Data Control Language, DCL):**
  - Definiert/entzieht Zugriffsrechte
  - `grant, revoke`

## Relationen vs. Tabellen

### Mengen und Bags

- Bisher haben wir Relationen (Mengen) betrachtet.
- In SQL betrachten wir aber Tabellen (Bags, Multimengen).

### Was ist der Unterschied?

**Relationen (Mengen):**

- Tabellen können keine Duplikate enthalten.
- Tabellen haben keine definierte Ordnung.
- Tabellen haben nicht unbedingt einen Schlüssel.

## Tabellen (Bags, Multimengen) in SQL:

- Können Duplikate enthalten.
- Haben eine implizite Einfügeordnung (die aber bei Abfragen i.d.R. keine Rolle spielt, außer bei `ORDER BY`).
- Können Schlüssel definieren (Primärschlüssel, Fremdschlüssel), müssen es aber nicht.

# Grundlegende Datentypen in SQL

---

## Datentypen

- `character(n)`, `char(n)`
- `character varying(n)`, `varchar(n)`
- `integer`, `smallint`
- `numeric(p,s)`, `decimal(p,s)`
  - `p` = Präzision = max. Anzahl von Stellen (gesamt)
  - `s` = Scale = Anzahl von Stellen nach dem Komma
- `real`, `double`
- `blob` oder `raw` für sehr große binäre Daten
- `date` für Datumsangaben
- `xml` für XML-Dokumente
- ...

### `varchar(n)` vs. `char(n)`

- Beide sind auf Länge `n` beschränkt.
- `char(n)` belegt immer `n` Bytes.
- `varchar(n)` belegt nur den benötigten Platz, plus Längeninformation.

# Tabellen erstellen

---

## Tabelle erstellen

```
CREATE TABLE professor (
    empid    integer,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

## Tabelle erstellen (mit UNIQUE Constraint)

```
CREATE TABLE professor (
    empid    integer UNIQUE NOT NULL,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

Das `UNIQUE`-Constraint lässt in einer Spalte jeden Wert nur ein einziges Mal zu.

Tabellen kann man auch auf Basis von anderen Tabellen erstellen:

```
CREATE TABLE professor2 AS (SELECT * FROM professor);
```

## Primärschlüssel

`PRIMARY KEY` kennzeichnet ein Attribut als **Schlüsselattribut**.

### Primärschlüssel als Spalten-Constraint

```
CREATE TABLE professor (
    empid    integer PRIMARY KEY,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

### Primärschlüssel als Tabellen-Constraint

```
CREATE TABLE professor (
    empid    integer,
    name     varchar(10) NOT NULL,
    rank     char(2),
    PRIMARY KEY (empid)
);
```

`PRIMARY KEY` kennzeichnet ein Attribut als **Schlüsselattribut**.

Das `Primary-Key`-Constraint beinhaltet `not-null`- und `unique`-Constraints.

## Fremdschlüssel

Gültige Werte des Fremdschlüssels müssen Werten des referenzierten Attributs entsprechen.

### Fremdschlüssel als Tabellen-Constraint

```
CREATE TABLE course (
    courseId    integer,
    title       varchar(30) NOT NULL,
```

```

    ects      integer,
    taughtby   integer,
    PRIMARY KEY (courseId),
    FOREIGN KEY (taughtby) REFERENCES professor(empid)
);

```

Fremdschlüssel können auch **Schlüsselkandidaten**(!) referenzieren (garantiert durch das **UNIQUE -Constraint**).

### Fremdschlüssel, der auf Primärschlüssel und UNIQUE-Constraint verweist

```

CREATE TABLE employee (
    cpr      integer PRIMARY KEY,
    empid    integer UNIQUE NOT NULL,
    name     varchar(20)
);

CREATE TABLE salary (
    cpr      integer REFERENCES employee(cpr),
    empid    integer REFERENCES employee(empid),
    salary   integer
);

```

### Defaultwerte

- Wird für ein Attribut beim Einfügen kein Wert angegeben, so wird dieser auf **NULL** gesetzt (Standard- bzw. Defaultwert).
- Beim Erstellen einer Tabelle können wir einen anderen Defaultwert definieren.

### Defaultwert für ein Attribut definieren

```

CREATE TABLE wine (
    wineID    integer NOT NULL,
    name      varchar(20) NOT NULL,
    color     varchar(10) DEFAULT 'red',
    year      integer,
    vineyard  varchar(20)
);

```

Der Defaultwert für die Farbe eines Weins soll "red" sein.

### Zahlengeneratoren

**Zahlengeneratoren** erzeugen automatisch fortlaufende eindeutige IDs.

### Sequenz erstellen und als Defaultwert nutzen

```

CREATE SEQUENCE serial START 101;

CREATE TABLE wine (
    wineID      integer PRIMARY KEY DEFAULT nextval('serial'),
    name        varchar(20) NOT NULL,
    color       varchar(10),
    year        integer,
    vineyard    varchar(20)
);

```

Beim Hinzufügen von Weinen soll automatisch eine eindeutige `wineID` generiert werden.

## Tabellen verändern

### Attribute hinzufügen

```

ALTER TABLE professor
ADD COLUMN office integer;

```

### Attribut löschen

```

ALTER TABLE professor
DROP COLUMN name;

```

### Attributtyp ändern

```

ALTER TABLE professor
ALTER COLUMN name type varchar(30);

```

```

CREATE TABLE professor (
    empid    integer NOT NULL,
    name     varchar(10) NOT NULL,
    rank     char(2)
);

```

## Tabelle löschen

```
DROP TABLE professor;
```

## Tabelle leeren

```
TRUNCATE TABLE professor;
```

Kann nicht verwendet werden, wenn der Fremdschlüssel einer anderen Tabelle auf die zu löschen Tabelle zeigt.

## Daten einfügen

```
INSERT INTO professor VALUES
(2136, 'Curie', 'C4', 36),
(2137, 'Kant', 'C4', 7);

INSERT INTO student (studid, name) VALUES
(28121, 'Archimedes');

INSERT INTO student (name) VALUES
('Meier');

INSERT INTO takes
SELECT studid, courseid
FROM student, course
WHERE title = 'Logics';
```

## Daten löschen und ändern

### Daten löschen

```
DELETE FROM student
WHERE semester > 13;

DELETE FROM student;
```

### Daten ändern

```
UPDATE student
SET semester = semester + 1;

UPDATE student
SET semester = semester + 1 WHERE ...;
```

## Rechteverwaltung

### Rechte gewähren

```
GRANT select (empid), update (office)
ON professor
TO some_user, another_user;
```

## Rechte entziehen

```
REVOKE ALL
ON professor
FROM some_user, another_user;
```

Rechte auf Tabellen, Spalten, ...:

```
select, insert, update, delete, rule, references, trigger
```

## Transaktionssteuerung

---

BEGIN;

kennzeichnet den **Beginn** einer Transaktion.

COMMIT;

schließt eine Transaktion ab.

Alle Änderungen, die in der Transaktion gemacht wurden, werden für andere sichtbar und sind garantiert dauerhaft (auch wenn sich ein Absturz ereignen sollte).

ROLLBACK;

setzt eine Transaktion zurück.

Alle Änderungen, die in der Transaktion gemacht wurden, werden verworfen.

## Anfragesprache

---

Dieses Kapitel wird in der Zusammenfassung nicht behandelt, da es laut Dozentin für die Prüfung nicht relevant ist.

## Übersetzung von SQL-Anfragen in relationale Algebra

---

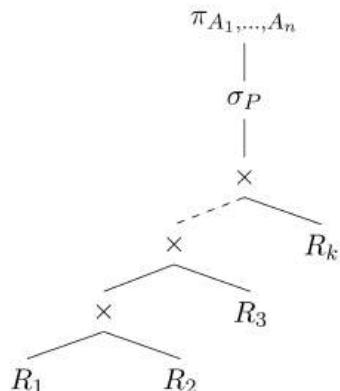
### Allgemeine Form einer (geschachtelten) SQL-Anfrage

Allgemeine Form einer  
(ungeschachtelten) SQL-Anfrage

```
SELECT A1, ..., An
FROM R1, ..., Rk
WHERE P;
```

Übersetzung in relationale Algebra

$$\pi_{A_1, \dots, A_n}(\sigma_P(R_1 \times \dots \times R_k))$$



## Mächtigkeit des SQL-Kerns

relationale Algebra	SQL
projection	SELECT DISTINCT
selection	WHERE ohne Schachtelung
join	FROM, WHERE mit Join oder NATURAL JOIN
renaming	FROM mit Tupelvariable; AS
difference	WHERE mit Schachtelung; EXCEPT
intersection	WHERE mit Schachtelung; INTERSECT
union	UNION

## Rekursion in SQL

### Beispiel 1: Summe der Zahlen von 1 bis 100

```

WITH RECURSIVE mytable(number) AS (
  VALUES(1)          -- nicht rekursiver Teil (Basisfall)
  UNION ALL
  SELECT number + 1  -- rekursiver Teil
  FROM mytable
  WHERE number < 100   -- Abbruchbedingung
)
SELECT sum(number)    -- eigentliche Anfrage
FROM mytable;

-- Result: 5050 (Summe der Zahlen von 1 bis 100)
  
```

- `WITH RECURSIVE mytable(number) AS (...)` : Definiert eine Common Table Expression (CTE) namens `mytable` mit einer Spalte `number`, die rekursiv aufgebaut wird.
- `VALUES(1)` : Der nicht-rekursive Teil initialisiert die CTE mit dem Startwert 1.
- `UNION ALL` : Kombiniert den nicht-rekursiven und den rekursiven Teil.
- `SELECT number + 1 FROM mytable WHERE number < 100` : Der rekursive Teil greift auf die vorherige Iteration von `mytable` zu und erzeugt den nächsten Wert (`number + 1`), solange die Bedingung `number < 100` erfüllt ist. Die CTE referenziert sich hier selbst.
- `SELECT sum(number) FROM mytable` : Die eigentliche Anfrage summiert alle Werte, die in der rekursiv erzeugten CTE `mytable` enthalten sind.

## Anfrage für Voraussetzungen von Kursen

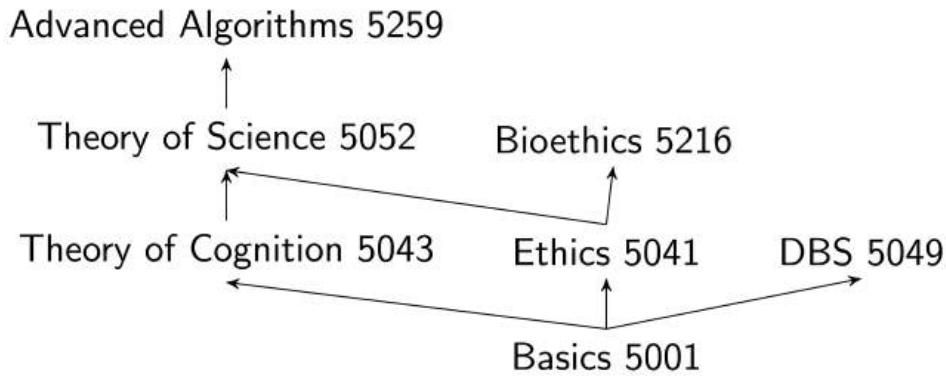
### Beispiel 2: Transitive Voraussetzungen von Kursen

```
WITH RECURSIVE transitiveCourse (pred, succ) AS (
    SELECT predecessor, successor
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
)
SELECT *
FROM transitiveCourse
ORDER BY (pred, succ) ASC;
```

- `WITH RECURSIVE transitiveCourse (pred, succ) AS (...)` : Definiert eine rekursive CTE namens `transitiveCourse` mit den Spalten `pred` (Voraussetzung) und `succ` (Nachfolger).
- `SELECT predecessor, successor FROM requires` : Der nicht-rekursive Teil wählt die direkten Voraussetzungen aus der Tabelle `requires`.
- `UNION ALL` : Kombiniert die direkten und die transitiven Voraussetzungen.
- `SELECT DISTINCT tc.pred, r.successor FROM transitiveCourse tc, requires r WHERE tc.succ = r.predecessor` : Der rekursive Teil verbindet die bereits gefundenen transitiven Voraussetzungen (`tc`) mit den direkten Voraussetzungen (`r`), um weitere transitive Beziehungen zu finden. Wenn der Nachfolger einer bereits bekannten Voraussetzung (`tc.succ`) der Vorgänger einer anderen Vorlesung (`r.predecessor`) ist, dann ist die ursprüngliche Voraussetzung (`tc.pred`) auch eine transitive Voraussetzung für den Nachfolger (`r.successor`). `DISTINCT` wird verwendet, um Duplikate zu vermeiden.
- `SELECT * FROM transitiveCourse ORDER BY (pred, succ) ASC` : Die eigentliche Anfrage wählt alle transitiven Voraussetzungen aus der CTE aus und sortiert das Ergebnis.

Dieses Beispiel zeigt, wie Rekursion verwendet werden kann, um transitive Beziehungen in hierarchischen Daten (wie Kursvoraussetzungen) zu ermitteln.

pred	succ
5001	5041
5001	5043
5001	5049
5001	5052
5001	5216
5001	5259
5041	5052
5041	5216
5041	5259
5043	5052
5043	5259
5052	5259



## Unendliche Rekursion verhindern

1. Die meisten DBMS beschränken die Rekursionstiefe mit einem Parameter.
2. Direkt in der Anfrage kodieren.

Beispiel zur Verhinderung unendlicher Rekursion durch Tiefenbegrenzung:

```

WITH RECURSIVE transitiveCourse (pred, succ, depth) AS (
    SELECT predecessor, successor, 0
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor, tc.depth + 1
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
        AND tc.depth < 1 -- Begrenzung der Rekursionstiefe
)
SELECT *
FROM transitiveCourse
ORDER BY (pred, succ) ASC;
  
```

- Im Vergleich zum vorherigen Beispiel wurde eine Spalte `depth` zur CTE `transitiveCourse` hinzugefügt, um die aktuelle Rekursionstiefe zu verfolgen.
- Im nicht-rekursiven Teil wird die Tiefe auf `0` initialisiert.
- Im rekursiven Teil wird die Tiefe bei jedem Schritt um `1` erhöht (`tc.depth + 1`).
- Die `WHERE`-Klausel im rekursiven Teil enthält nun die Bedingung `AND tc.depth < 1`. Dies begrenzt die Rekursionstiefe auf maximal 1. Sobald `tc.depth` den Wert 1 erreicht, wird der rekursive Schritt nicht mehr ausgeführt, wodurch eine unendliche Schleife verhindert wird.

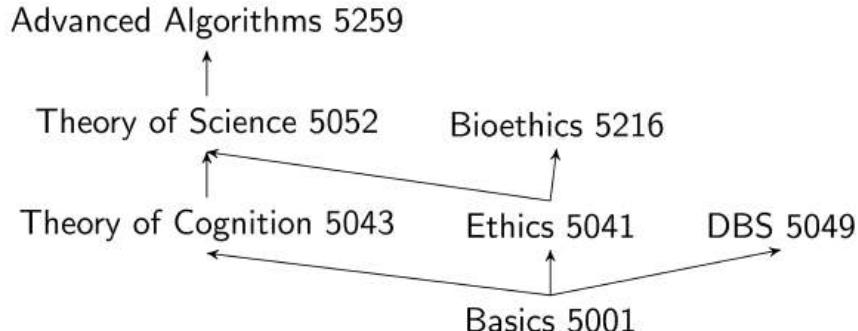
**Hinweis:** Der Wert `< 1` in diesem Beispiel ist eine sehr strenge Begrenzung und würde nur direkte Voraussetzungen und deren unmittelbare Voraussetzungen berücksichtigen. In realen

Szenarien wäre eine höhere Tiefenbegrenzung oder eine andere Abbruchlogik erforderlich, abhängig von der Struktur der Daten. Die meisten DBMS erlauben auch eine Konfiguration der maximalen Rekursionstiefe auf Systemebene.

### Ohne der Bedingung:

pred	succ	depth
5001	5041	0
5001	5043	0
5001	5049	0
5001	5052	1
5001	5216	1
5001	5259	2
5041	5052	0
5041	5216	0
5041	5259	1
5043	5052	0
5043	5259	1
5052	5259	0

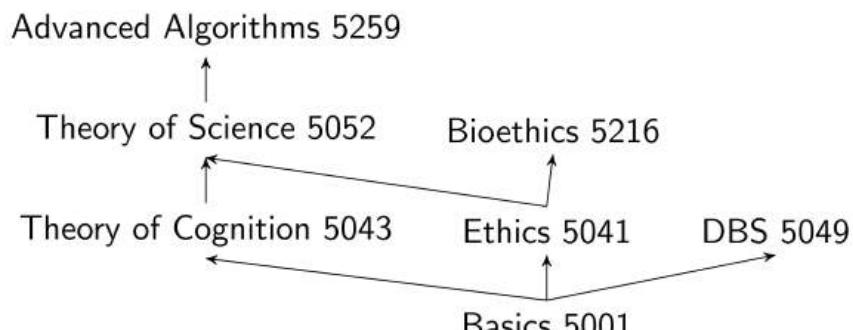
ohne Bedingung  $t.\text{depth} < 1$



### Mit der Bedingung:

pred	succ	depth
5001	5041	0
5001	5043	0
5001	5049	0
5001	5052	1
5001	5216	1
5001	5259	2
5041	5052	0
5041	5216	0
5041	5259	1
5043	5052	0
5043	5259	1
5052	5259	0

mit Bedingung  $t.\text{depth} < 1$



### Noch ein Beispiel von Rekursion

Alle Voraussetzungen einer bestimmten Vorlesung

```

WITH RECURSIVE transitiveCourse (pred, succ) AS (
  SELECT predecessor, successor
  FROM requires
  UNION ALL
  SELECT DISTINCT tc.pred, r.successor
  FROM transitiveCourse tc, requires r
)
  
```

```

        WHERE tc.succ = r.predecessor
    )
SELECT c2.title
FROM transitiveCourse tv, course c1, course c2
WHERE tv.succ = c1.courseid
    AND c1.title = 'Theory of Science'
    AND tv.pred = c2.courseid;

```

## Was genau ist das Resultat dieser Anfrage?

Das Resultat dieser Anfrage sind die Titel aller Vorlesungen, die (direkte oder indirekte) Voraussetzungen für die Vorlesung "Theory of Science" sind.

### Erläuterung der Anfrage:

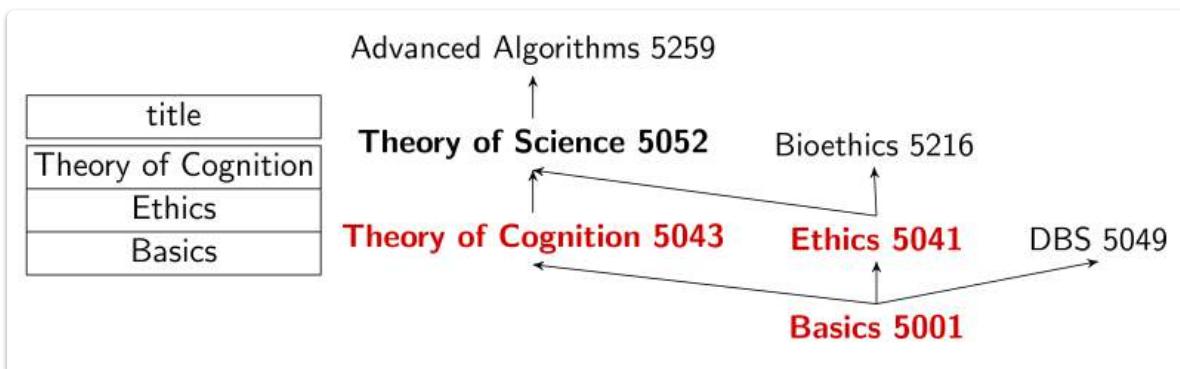
#### 1. Rekursive CTE `transitiveCourse`:

- Berechnet alle transitiven Beziehungen zwischen Voraussetzungen (`pred`) und Nachfolgern (`succ`).
- Der nicht-rekursive Teil wählt die direkten Voraussetzungen aus der Tabelle `requires`.
- Der rekursive Teil findet weitere Voraussetzungen, indem er bereits gefundene (`tc`) mit direkten Voraussetzungen (`r`) verknüpft.

#### 2. Hauptabfrage:

- Verbindet die rekursive CTE `transitiveCourse` (alias `tv`) mit der Tabelle `course` zweimal (alias `c1` und `c2`).
- `tv.succ = c1.courseid`: Verbindet den Nachfolger (`succ`) aus der transitiven Voraussetzungsliste mit der `courseid` der Vorlesung, für die wir die Voraussetzungen suchen (`c1`).
- `c1.title = 'Theory of Science'`: Filtert das Ergebnis, sodass wir nur die Voraussetzungen für die Vorlesung mit dem Titel 'Theory of Science' betrachten.
- `tv.pred = c2.courseid`: Verbindet die Voraussetzung (`pred`) aus der transitiven Voraussetzungsliste mit der `courseid` einer anderen Vorlesung (`c2`).
- `SELECT c2.title`: Gibt den `title` (`c2.title`) dieser vorausgesetzten Vorlesung aus.

Zusammenfassend liefert die Anfrage eine Liste der Titel aller Kurse, die in der Kette der Voraussetzungen irgendwann vor der Vorlesung "Theory of Science" stehen.





## Anzahl der Ergebnisse begrenzen

Um die Anzahl der Ausgaben zu begrenzen gibt es **verschiedene Möglichkeiten**

### SQL 2003: Window Function

```
SELECT *
FROM (SELECT
    ROW_NUMBER() OVER (ORDER BY semester DESC) AS number,
    studid, name, semester
    FROM student
) AS studTmp
WHERE number <= 5;
```

```
SELECT *
FROM (SELECT
    RANK() OVER (ORDER BY semester DESC) AS number,
    studid, name, semester
    FROM student
) AS studTmp
WHERE number <= 5;
```

### SQL 2008: Fetch-First-Klausel

```
SELECT *
FROM student
ORDER BY semester DESC
FETCH FIRST 5 ROWS ONLY;
```

Unterstützt von IBM DB2, Sybase SQL Anywhere, PostgreSQL,...

### Nicht-Standard-Syntax

```
SELECT * FROM student LIMIT 5;
```

Unterstützt von MySQL, Sybase SQL Anywhere, PostgreSQL,...

```
SELECT * FROM student WHERE ROWNUM <= 5;
```

Unterstützt von Oracle

```
SELECT TOP 5 FROM student;
```

Unterstützt von MS SQL server

**LIMIT** wird in der Prüfung akzeptiert.



# Abstraktionsebenen eines Datenbanksystems

Änderungen auf der logischen Ebene haben keine Auswirkungen auf externe Schemata und Anwendungsprogramme.



Datenunabhängigkeit:

- **Physische Unabhängigkeit**: Änderungen auf der physischen Ebene haben keinen Einfluss auf die logische Ebene.
- **Logische Datenunabhängigkeit**: Änderungen auf der logischen Ebene haben keinen Einfluss auf die Sichten (Views).

Mehr zu **Abstraktion** siehe: EP2

## Beispiele für Views:

### Beispiel 1: View `profsAndtheirCourses`

```

CREATE VIEW profsAndtheirCourses AS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid = c.taughtBy;

SELECT * FROM profsAndtheirCourses;
  
```

- `CREATE VIEW profsAndtheirCourses AS ...`: Erstellt eine virtuelle Tabelle (View) namens `profsAndtheirCourses`.
- Die Definition der View (`SELECT c.title, p.name ...`) wählt den Titel der Kurse (`c.title`) und die Namen der Professoren (`p.name`) aus, indem die Tabellen `professor` und `course` über die `empid` des Professors und `taughtBy` des Kurses verbunden werden.
- Die zweite `SELECT`-Anweisung greift auf die erstellte View zu, als wäre sie eine reguläre Tabelle, und gibt alle Spalten (Titel des Kurses und Name des Professors) aus.

### Beispiel 2: View `ectsPerStud`

```

CREATE VIEW ectsPerStud AS
SELECT s.name, s.studid, SUM(c.ects) AS sum
  
```

```

FROM student s, takes t, course c
WHERE t.courseid = c.courseid AND s.studid = t.studid
GROUP BY s.name, s.studid;

SELECT sum FROM ectsPerStud;

```

- `CREATE VIEW ectsPerStud AS ...`: Erstellt eine View namens `ectsPerStud`.
- Die Definition der View berechnet die Summe der ECTS-Punkte (`SUM(c.ects)`) für jeden Studenten. Dazu werden die Tabellen `student`, `takes` und `course` verbunden und nach dem Namen und der `studid` der Studenten gruppiert. Das Ergebnis der Summe wird als `sum` aliasiert.
- Die zweite `SELECT`-Anweisung greift auf die View zu und wählt die Spalte `sum` (die Summe der ECTS-Punkte pro Student) aus.

**Views/Sichten können verwendet werden, um abgeleitete Attribute darzustellen (ER Diagramm).**

- Views ermöglichen es, komplexe Abfragen zu kapseln und als einfache virtuelle Tabellen darzustellen.
- Dies kann die Datenmodellierung vereinfachen, indem abgeleitete Informationen (wie die Summe der ECTS pro Student oder die Liste der Kurse pro Professor) als scheinbar eigenständige Attribute oder Relationen dargestellt werden.
- Dies kann auch die Wiederverwendbarkeit von Abfragen und die Lesbarkeit von SQL-Code verbessern.

## Views ändern

`CREATE OR REPLACE VIEW :`

```

CREATE OR REPLACE VIEW profsAndtheirCourses AS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid = c.taughtBy;

```

- Ermöglicht das Ändern einer bestehenden View oder das Erstellen einer neuen View, falls noch keine mit dem angegebenen Namen existiert.
- **Wichtig:** `REPLACE VIEW` erwartet dieselben Spalten in derselben Reihenfolge mit denselben Datentypen wie die ursprüngliche View. Andernfalls kann es zu Fehlern oder unerwartetem Verhalten kommen.

`DROP VIEW :`

```

DROP VIEW ectsPerStud;
CREATE VIEW ectsPerStud AS
SELECT s.name, s.studid, SUM(c.ects) AS sum

```

```
FROM student s, takes t, course c
WHERE t.courseid = c.courseid
  AND s.studid = t.studid
GROUP BY s.name, s.studid;
```

- `DROP VIEW ectsPerStud;` : Entfernt die existierende View namens `ectsPerStud`.
- Anschließend wird die View mit einer neuen Definition neu erstellt. Dies ist eine Möglichkeit, eine View zu ändern, wenn die Struktur sich signifikant ändert.

### Alternative:

- Die Views erst löschen (`DROP VIEW`), dann neu erstellen (`CREATE VIEW`).
  - SQL-Erweiterungen, z.B. `ALTER VIEW` von PostgreSQL, bieten eine direktere Möglichkeit, die Definition einer View zu ändern, ohne sie vorher löschen zu müssen. Allerdings ist `ALTER VIEW` nicht Teil des Standard-SQL und wird nicht von allen Datenbankmanagementsystemen unterstützt.
-

# Sichten vs. materialisierte Sichten

---

## (Dynamische) Sicht (View)

- Entspricht einem Makro für eine Anfrage.
- Ergebnis der Anfrage wird nicht vorberechnet, sondern erst dann, wenn die Sicht benutzt wird.
- **Vorteil:** Die Daten in der Sicht sind immer aktuell, da die zugrundeliegende Abfrage bei jedem Zugriff neu ausgeführt wird.
- **Nachteil:** Die Performance kann leiden, insbesondere bei komplexen Sichten, da die Berechnung bei jeder Anfrage wiederholt wird.

## Materialisierte Sicht (Materialized View)

- Ergebnis der Sicht wird vorberechnet und persistent gespeichert (ähnlich einer regulären Tabelle).
- Rechenaufwand bevor irgendeine Anfrage ausgeführt wird (bei der Erstellung und Aktualisierung der materialisierten Sicht).
- **Vorteil:** Deutlich schnellere Abfragezeiten, da das Ergebnis bereits vorliegt.
- **Nachteil:** Die Daten in der materialisierten Sicht sind nicht immer sofort aktuell. Sie müssen explizit aktualisiert werden (manuell oder periodisch), was zusätzlichen Aufwand verursacht.

## Was ist die bessere Wahl? Laufzeit vs. Updates

Die Wahl zwischen einer dynamischen und einer materialisierten Sicht hängt von den spezifischen Anforderungen ab:

- **Dynamische Sicht ist besser, wenn:**
  - Daten häufig geändert werden und die Aktualität der Daten in der Sicht entscheidend ist.
  - Die zugrundeliegende Abfrage relativ einfach ist und die Performance keine kritische Rolle spielt.
  - Speicherplatz begrenzt ist, da keine zusätzlichen Daten gespeichert werden müssen.
- **Materialisierte Sicht ist besser, wenn:**
  - Die Performance von Abfragen auf die Sicht kritisch ist (z.B. bei häufig genutzten, komplexen Berichten).
  - Die Daten nicht sehr häufig geändert werden oder eine gewisse Latenz bei der Aktualisierung akzeptabel ist.
  - Ausreichend Speicherplatz für die Speicherung der vorberechneten Daten vorhanden ist.

Oft ist es ein Trade-off zwischen der Aktualität der Daten (dynamische Sicht) und der Performance der Abfragen (materialisierte Sicht).

## Updates und Views

```
CREATE VIEW howTough AS
SELECT empid, AVG(grade) AS avgGrade
FROM grades
GROUP BY empid;

UPDATE howTough
SET avgGrade = 1.0
WHERE empid = ( SELECT empid
                 FROM professor
                 WHERE name = 'Socrates');
```

Was ist hier das Problem?

- Problem: Wo speichere ich das?
- Wie sollen die Noten in der Ursprungstabelle verändert werden?

## Noch ein Beispiel:

```
CREATE VIEW courseView AS
SELECT title, ects, name
FROM course, professor
WHERE taughtBy = empid;

INSERT INTO courseView
VALUES ('Nihilism', '2', 'Nobody');
```

Tabellen:

- course: { courseid, title, ects, taughtBy }
- professor: { empid, name, rank, office }

Was ist das Problem? Welche Tupel sollen in die Ursprungstabellen eingefügt werden?

Das Problem bei diesem `INSERT`-Befehl auf die View `courseView` ist, dass die View auf einer Verknüpfung zweier Tabellen (`course` und `professor`) basiert und nicht alle Spalten der Basistabellen in der View enthalten sind. Insbesondere fehlt die Information, wie die eingefügten Werte den Primär- und Fremdschlüssen der Basistabellen zugeordnet werden sollen.

## Änderbarkeit von Sichten in SQL

Daten in einer View sind veränderbar (update/insert/delete), wenn ...

in SQL-92

- nur eine Tabelle
- nur Selektion und Projektion
- keine Aggregatfunktionen, Gruppierung und Duplikateliminierung

in SQL-99

- wie SQL-92
- mehrere Tabellen und Attribute, wenn diese mit Hilfe des Primärschlüssels eindeutig zugeordnet werden können
- D.h. auch Views mit Joins können manchmal geändert werden

## Änderbarkeit von Views in SQL

Views in Postgres sind nicht materialisiert!

```
CREATE VIEW stud AS
  SELECT *
  FROM student;
```

```
INSERT INTO stud VALUES (42, 'mueller', 11);
```

**ERROR: cannot insert into a view.**

**HINT: You need an unconditional ON INSERT DO INSTEAD rule.**

- Views in Postgres sind nicht änderbar!
- Aber: Jede View kann einzeln mit Hilfe von Regeln (rules) „freigeschaltet“ werden.

# Integritätsbedingungen

- Zusätzliches Instrument, um Inkonsistenzen zu verhindern.
- Ziel: Das Einfügen inkonsistenter Daten verhindern.

Welche Integritätsbedingungen haben wir in der Vorlesung schon kennengelernt?

- Keine zwei Tupel haben dieselben Werte in den Schlüsselattributen (Primärschlüsselbedingung).
- Kardinalitäten/Stelligkeit von Beziehungstypen (definiert die Anzahl der Beziehungen, die eine Entität eingehen kann).
- Generalisierung: jedes Entity vom Subtyp muss im Obertyp enthalten sein (Totalitätsbedingung bei Spezialisierung).
- Domänen (d.h. zulässige Werte) von Attributen (legt fest, welche Werte für ein Attribut gültig sind).
- Primärschlüssel und Fremdschlüssel (Primärschlüssel identifiziert Tupel eindeutig, Fremdschlüssel stellt Beziehungen zwischen Tabellen sicher und referenziert existierende Primärschlüsselwerte).
- NOT NULL, UNIQUE (Constraints, die sicherstellen, dass ein Attribut keine Nullwerte enthält bzw. dass alle Werte in einem Attribut eindeutig sind).

## Statische Integritätsbedingungen

Statische Integritätsbedingungen müssen von jedem Zustand der Datenbank erfüllt werden.

Beispiele:

- NOT NULL -Constraint:

```
CREATE TABLE professor (
    ...
    empid INTEGER NOT NULL,
    ...
);
```

- Stellt sicher, dass die Spalte `empid` in der Tabelle `professor` keine Nullwerte enthalten darf.

- CHECK -Constraint (Einschränkung des Wertebereichs):

```
CREATE TABLE student (
    ...
    semester INTEGER CHECK (semester BETWEEN 1 AND 20),
    ...
);
```

- Stellt sicher, dass der Wert der Spalte `semester` in der Tabelle `student` zwischen 1 und 20 (inklusive) liegen muss.
- **CHECK -Constraint (Aufzählungstypen):**

```
CREATE TABLE professor (
    ...
    rank VARCHAR(2) CHECK (rank IN ('C2', 'C3', 'C4')),
    ...
);
```

- Stellt sicher, dass der Wert der Spalte `rank` in der Tabelle `professor` einer der Werte 'C2', 'C3' oder 'C4' sein muss.
- **CREATE DOMAIN (Festlegung eines benutzerdefinierten Wertebereichs):**

```
CREATE DOMAIN wineColor varchar(5)
DEFAULT 'red'
CHECK (VALUE IN ('red', 'white', 'rose'));

CREATE TABLE wine (
    wineID INT PRIMARY KEY,
    name VARCHAR(20) NOT NULL,
    color wineColor,
    ...
);
```

- `CREATE DOMAIN wineColor ...` : Definiert einen benutzerdefinierten Datentyp namens `wineColor`, der ein `VARCHAR(5)` ist und standardmäßig den Wert 'red' hat. Ein `CHECK` -Constraint stellt sicher, dass nur die Werte 'red', 'white' oder 'rose' für diesen Datentyp zulässig sind.
- In der Tabelle `wine` wird die Spalte `color` mit dem benutzerdefinierten Datentyp `wineColor` deklariert, wodurch die für diesen Datentyp definierten Einschränkungen automatisch angewendet werden.

## Dynamische Integritätsbedingungen

### Referentielle Integrität

Referentielle Integrität bedeutet

Fremdschlüssel müssen auf existierende Tupel verweisen oder einen Nullwert enthalten.

Was passiert, wenn kein:e Professor:in mit empid 007 existiert?

**insert into course**

**values (5100, 'Spying for Dummies', 4, 007);**

Wie kann dieses „insert“ verhindert werden?

## Festlegung von Schlüsseln

Kandidatenschlüssel

- UNIQUE
- Mehrere Kandidatenschlüssel für eine Relation sind möglich
- Darf null sein!

Primärschlüssel

- PRIMARY KEY
- Nur ein Primärschlüssel pro Relation
- **Impliziert UNIQUE NOT NULL**

Fremdschlüssel

- FOREIGN KEY
- null ist möglich, kann aber mit NOT NULL verhindert werden

## Verhalten bei Änderungsoperationen

Dynamische Integritätsbedingungen müssen von Zustandsänderungen erfüllt werden.

Bei Änderung von referenzierten Daten haben wir mindestens 3 Optionen

- Zurückweisen der Änderungsoperation (Default-Verhalten)
- Propagieren der Änderungen (**CASCADE**)
- Verweise auf „unbekannt“ setzen: **set null**

In Postgres außerdem

- Auf Defaultwert setzen (SET DEFAULT)

## Beispiel

Update on table R

S		R	
	$\alpha$		$\underline{\kappa}$
	$\kappa_1$	→	$\kappa_1$
	$\kappa_2$	→	$\kappa_2$
...	...		...

UPDATE R

SET  $\kappa = \kappa'_1$   
WHERE  $\kappa = \kappa_1$

DELETE FROM R

WHERE  $\kappa = \kappa_1$

Wie soll damit umgegangen werden?

Da gibts mehrere Optionen:

Option 1

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ ));

S		R	
	$\alpha$		$\underline{\kappa}$
	$\kappa_1$	→	$\kappa_1$
	$\kappa_2$	→	$\kappa_2$
...	...		...

Ergebnis der Änderungsoperation: DB bleibt unverändert

Option 2

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON UPDATE CASCADE);

S	
	$\alpha$
	$\kappa'_1$
	$\kappa_2$
...	...

R	
	$\kappa$
	$\kappa'_1$
	$\kappa_2$
...	...

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON DELETE CASCADE);

S	
	$\alpha$
	$\kappa_2$
...	...

R	
	$\kappa$
	$\kappa_2$
...	...

Ergebnis der Änderungsoperation:  
Schlüssel in R und S geändert.

Ergebnis der Änderungsoperation:  
Schlüssel in R und S gelöscht.

### Option 3:

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON UPDATE SET NULL);

S	
	$\alpha$
	null
	$\kappa_2$
...	...

R	
	$\kappa$
	$\kappa'_1$
	$\kappa_2$
...	...

Ergebnis der Änderungsoperation:  
Tupel aus R auf  $\kappa'_1$ , in S auf null  
geändert.

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON DELETE SET NULL);

S	
	$\alpha$
	null
	$\kappa_2$
...	...

R	
	$\kappa$
...	...
	$\kappa_2$
...	...

Ergebnis der Änderungsoperation:  
Tupel aus R gelöscht, Schlüssel in S  
auf null geändert.

## Kaskadierendes Löschen

Referentielle Integrität: ON DELETE CASCADE

```
CREATE TABLE course (
  ...,
  taughtBy INTEGER REFERENCES professor(epid)
    ON DELETE CASCADE
);
```

```
CREATE TABLE takes (
  ...,
  courseid INTEGER REFERENCES course(courseid)
    ON DELETE CASCADE
);
```

```
DELETE FROM professor WHERE name = 'Socrates';
```

Beispiel siehe Slides: [hier](#)

## Komplexe Konsistenzbedingung

```
CREATE TABLE grades (
    studid      integer REFERENCES student ON DELETE CASCADE,
    courseid    integer REFERENCES course,
    grade       numeric(2,1) CHECK (grade BETWEEN 0.7 AND 5.0),
    PRIMARY KEY(studid, courseid)
    CONSTRAINT hasTaken
        CHECK (EXISTS (SELECT *
                      FROM takes h
                     WHERE h.courseid = grades.courseid AND
                           h.studid = grades.studid))
);
```

- Die CHECK-Klausel wird für jedes UPDATE und INSERT ausgeführt.
- Operation wird zurückgewiesen, wenn der Ausdruck zu false evaluiert!  
True und unknown widersprechen der Bedingung nicht!

- PostgreSQL unterstützt keine CHECK-Constraints, die andere Tabellen involvieren.
- Workaround durch die Verwendung von **Triggern**

## 5. Transaktionen

- Es könnte irgendwie zwischen Lesen und Schreiben was passieren.
- Alle Schritte müssen deshalb als Einheit betrachtet werden nach dem "Alles oder nichts"-Prinzip
- Sobald abgeschlossen, müssen die Änderungen *permanent gespeichert* werden.

### Transaktion

#### Was ist eine Transaktion?

- **Definition:** Eine Transaktion fasst mehrere Datenbankoperationen zu einer einzigen logischen Einheit zusammen.
- **Aktionen innerhalb einer Transaktion:**
  - Zugriff auf verschiedene Datenbankeinträge.
  - Möglicherweise Veränderung einiger Einträge.
- **Definition der Transaktionsgrenzen:** Erfolgt durch den Benutzer oder die Softwareanwendung.

#### Eigenschaften von Transaktionen: ACID

Die ACID-Eigenschaften sind grundlegend für zuverlässige Transaktionsverarbeitung in Datenbanken.

##### ⌚ Atomicity (Atomarität)

- **Prinzip:** Eine Transaktion wird als unteilbare Einheit behandelt.
- **Auswirkung:** Entweder werden *alle* Operationen innerhalb der Transaktion erfolgreich in der Datenbank durchgeführt (Commit), oder *keine* von ihnen wird übernommen (Rollback).
- **Implementierung:** Typischerweise durch die Verwendung von Log-Einträgen (Protokolldateien), die Änderungen vor der eigentlichen Durchführung festhalten und im Fehlerfall ein Zurücksetzen ermöglichen.

##### ⌚ Consistency (Konsistenz)

- **Prinzip:** Die Ausführung einer Transaktion in Isolation (ohne gleichzeitige andere Transaktionen) bewahrt den konsistenten Zustand der Datenbank.
- **Gewährleistung:**
  - Einhaltung definierter Constraints (z.B. Datentypen, Wertebereiche).
  - Durchführung von Checks (Überprüfungen von Bedingungen).
  - Einhaltung von Assertions (Zusicherungen über den Datenbankzustand).

- **Anwendungsdefinition:** Die Semantik der Konsistenz wird auch durch die Anwendung selbst definiert.
  - **Beispiel:** Bei einer Banküberweisung muss die Summe des abgebenden und empfangenden Kontos gleich bleiben – es darf kein Geld "entstehen" oder "verschwinden". Die Gesamtsumme aller beteiligten Konten muss vor und nach der Transaktion identisch sein.

### ⌚ Isolation

- **Prinzip:** Jede Transaktion operiert auf der Datenbank so, als ob sie die einzige aktive Transaktion wäre.
- **Auswirkung:** Zwischenergebnisse einer Transaktion dürfen für andere, gleichzeitig laufende Transaktionen nicht sichtbar sein. Dies verhindert unerwünschte Beeinflussungen und Inkonsistenzen.
- **Implementierung:** Typischerweise durch den Einsatz von Locks (Sperren), die den Zugriff auf betroffene Datenobjekte für andere Transaktionen einschränken, bis die aktuelle Transaktion abgeschlossen ist.

### ⌚ Durability (Dauerhaftigkeit)

- **Prinzip:** Änderungen, die von einer erfolgreich abgeschlossenen Transaktion (Commit) an der Datenbank vorgenommen wurden, müssen dauerhaft gespeichert bleiben und dürfen auch bei Systemfehlern (z.B. Stromausfall, Festplattenausfall) nicht verloren gehen.
- **Gewährleistung:**
  - Typischerweise durch die Verwendung von Log-Einträgen (Protokolldateien), in denen die durchgeführten Änderungen persistent protokolliert werden.
  - Im Fehlerfall können die Änderungen anhand des Logs wiederhergestellt werden (Recovery-Mechanismen).

## Operationen auf Transaktionsebene

**begin of transaction (BOT)**

Repräsentiert den Beginn einer Transaktion, d.h., alle Folgebefehle zusammen bilden eine Transaktion.

In SQL      BEGIN;

**commit**

Repräsentiert das Ende einer Transaktion, d.h., alle Änderungen werden festgeschrieben und sind auch für andere sichtbar.

In SQL      COMMIT;

**rollback oder abort**

Führt zu einem "roll back" der Transaktion, d.h., alle Änderungen werden zurückgesetzt/verworfen.

In SQL      ROLLBACK;

**"autocommit" Modus**

Jeder Befehl wird in einer eigenen Transaktion ausgeführt.

## Einfache Konsistenzprüfung (Checks)

```
CREATE TABLE emp(
    eid      INT          PRIMARY KEY,
    ename    VARCHAR(30)  NOT NULL,
    salary   INT          NOT NULL CHECK (salary > 0)
);
```

```
-- primary key violation
insert into emp values (11, 'Kim', 200);
-- Not null constraint violation
insert into emp values (44, NULL, 200);
-- Check statement violation
insert into emp values (44, 'Kim', -200);
```

Da sind einfache Konsistenzchecks die man bei Erstellung einer Tabelle definieren kann

- Viele der Fehler können **von DBMS erkannt** werden
- Verwende diese Checks!

## Savepoints

Transaktionen von langer Dauer können zusätzlich Savepoints definieren

`SAVEPOINT savepoint_name;`

Definiert einen Punkt/Zustand innerhalb einer Transaktion.

Eine Transaktion kann bis zum Savepoint **teilweise rückgängig gemacht werden.**

`ROLLBACK TO savepoint_name;`

Setzt die aktive Transaktion zurück bis zum Savepoints savepoint\_name

Man kann einfach Zwischenspeicherungen machen.

Ein Beispiel dazu wäre:

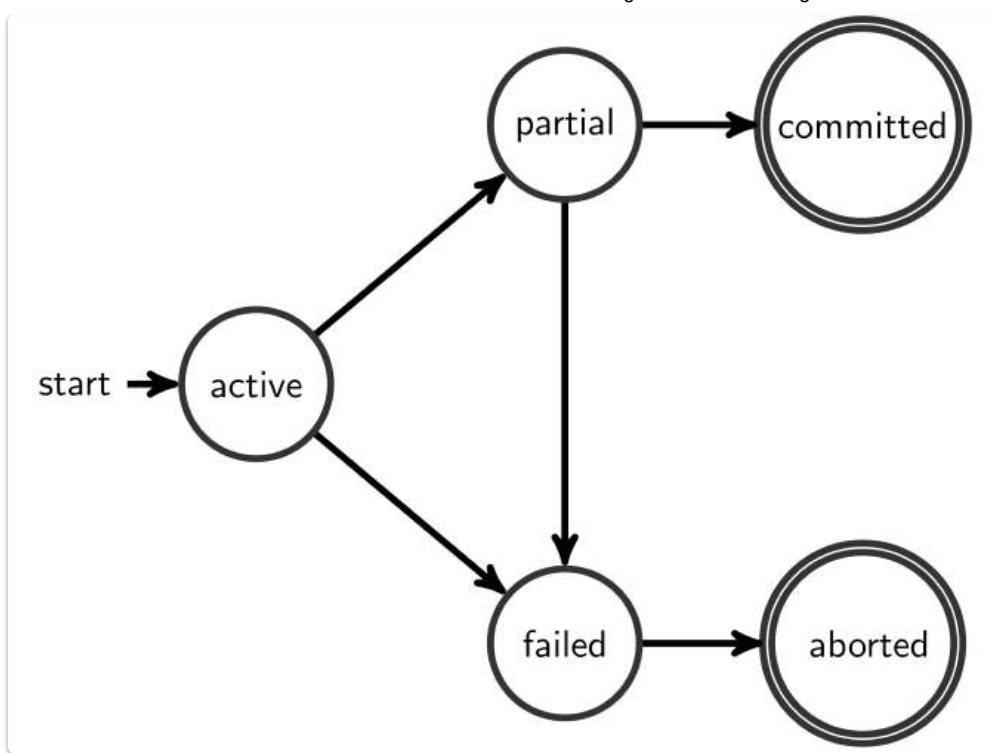
```
BEGIN;
INSERT INTO tab VALUES ...
SAVEPOINT A;
INSERT INTO tab VALUES...
SAVEPOINT B;
SELECT * FROM tab;
ROLLBACK TO A;
SELECT * FROM tab;
```

## Beenden einer Transaktion

Für den Abschluss einer Transaktion gibt es drei Möglichkeiten:

- ① Den **erfolgreichen** Abschluss durch **Commit**
- ② Den **erfolglosen** Abschluss durch ein **Abort**
- ③ Den **erfolglosen** Abschluss durch einen **Fehler**

## Zustandsdiagramm für Transaktionen



Es kann sein, dass wenn man fertig mit seiner Operation ist und committen will, dass man das dann nicht machen kann, **weil eine andere Person in der Zwischenzeit etwas verändert hat**. Dadurch kann allerdings nichts schiefgehen und es wird sicher aborted.

## Realisierung vom DBMS

Die beiden wichtigsten Komponenten der Transaktionsverwaltung sind:

### Mehrbenutzersynchronisation (Isolation)

- **Ziel:** Semantische Korrektheit bei Nebenläufigkeit.
- **Vorteil:** Hoher Durchsatz.
- **Serialisierbarkeit:** Ergebnis nebenläufiger Ausführung soll serieller Ausführung entsprechen.
- **Isolation Levels:** Schwächere Stufen für mehr Nebenläufigkeit möglich.

### Recovery (Atomicity und Durability)

- **Ziel:** Atomarität und Dauerhaftigkeit sicherstellen.
- **Rollback:** Zurücksetzen teilweise ausgeführter Transaktionen.
- **Wiederausführung:** Nach Ausfällen.
- **Persistenz:** Sicherstellen der Dauerhaftigkeit von Änderungen.

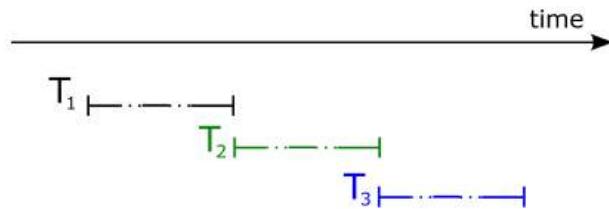
## Schedules und Serialisierbarkeit

### Nebenläufigkeit (Parallelität)

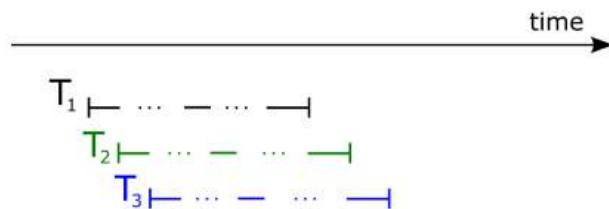
Das „I“ in ACID.

Ausführung der drei Transaktionen  $T_1$ ,  $T_2$ , und  $T_3$

(a) im Einzelbetrieb



(b) im (nebenläufigen) Mehrbenutzerbetrieb mit verzahnter Ausführung



## Probleme bei nebenläufiger Ausführung

1. Lost Updates (verlorengegangene Änderung durch Überschreiben)

Schritt	$T_1$	$T_2$
1.	read(A,a1)	
2.	$a1 := a1 - 300$	
3.		read(A,a2)
4.		$a2 := a2 * 1.03$
5.		<b>write(A,a2)</b>
6.	<b>write(A,a1)</b>	
7.	read(B,b1)	
8.	$b1 := b1 + 300$	
9.	write(B,b1)	

2. Dirty Read (Abhängigkeit von nicht freigegebenen Änderungen)

Steps	$T_1$	$T_2$
1.	read(A,a1)	
2.	$a1 := a1 - 300$	
3.	<b>write(A,a1)</b>	
4.		<b>read(A,a2)</b>
5.		$a2 := a2 * 1.03$
6.		<b>write(A,a2)</b>
7.	read(B, b1)	
8.	...	
9.	abort	

3. Non-Repeatable Read (Abhängigkeit von anderen Updates)

$T_1$	$T_2$
<pre>update account set balance=42000 where accountID=12345</pre>	<pre>select sum(balance) from account</pre> <pre>select sum(balance) from account</pre>

#### 4. Phantomproblem (Abhängigkeit von neuen/gelöschten Tupeln)

$T_1$	$T_2$
<pre>insert into account values (C,1000,...)</pre>	<pre>select sum(balance) from account</pre> <pre>select sum(balance) from account</pre>

## Formale Definition eines Schedules

- **Schedule (Historie):** Sequenz von Operationen mehrerer Transaktionen.
- **Nebenläufige Transaktionen:** Operationen können verzahnt sein.
- **Operationen:**
  - `read(Q, q)` : Liest Datenobjekt Q in lokale Variable q.
  - `write(Q, q)` : Schreibt lokale Variable q in Datenobjekt Q.
  - Arithmetische Operationen
  - `commit` : Beendet Transaktion erfolgreich.
  - `abort` : Bricht Transaktion ab.

## Arten von Schedules

- **Serieller Schedule:** Operationen von Transaktionen werden nacheinander ohne Überlappung ausgeführt.
- **Nebenläufiger Schedule:** Operationen von Transaktionen werden zeitlich überlappend ausgeführt.

## Gültiger Schedule

- Ein Schedule ist **gültig (valid)**, wenn das Ergebnis der Ausführung "korrekt" ist (Definition der Korrektheit hängt vom Kontext ab).

## Korrektheit

⌚ **Definition  $D_1$ :**

- Eine nebenläufige Ausführung von Transaktionen muss die Datenbank in einem **konsistenten Zustand** hinterlassen.
- Voraussetzung: Jede einzelne Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand, wenn sie isoliert ausgeführt wird.
- Also muss auch in einem konsistenten Zustand gestartet werden

### ⌚ Definition $D_2$ (Ergebnisäquivalenz):

- Eine nebenläufige Ausführung von Transaktionen muss **ergebnisäquivalent** zu einer seriellen Ausführung derselben Transaktionen sein.
- **Ergebnisäquivalent** bedeutet: Die finalen Zustände der Datenbank müssen identisch sein, egal in welcher seriellen Reihenfolge die Transaktionen ausgeführt worden wären.
- Deshalb ist auch  $D_2$  besser als  $D_1$

## Konflikt Serialisierbarkeit

### Mögliche Konflikte zwischen Transaktionen

Sobald ein write drinnen ist, haben wir das Potenzial zu einem Konflikt

„Konflikte“ zwischen Paaren von Transaktionen ( $T_1$  und  $T_2$ ) und deren Operationen.

schedule $S_A$	
$T_1$	$T_2$
write(X, x)	
	read(X, x)

**Konflikt**

schedule $S_C$	
$T_1$	$T_2$
	read(X, x)
write(X, x)	

**Konflikt**

schedule $S_B$	
$T_1$	$T_2$
write(X, x)	
	write(X, x)

**Konflikt**

schedule $S_D$	
$T_1$	$T_2$
read(X, x)	
	read(X, x)

**Kein Konflikt**

### Definition ( $D_4$ ) Conflict Serializability

- Ein Schedule ist **conflict serializable**, wenn er **konfliktäquivalent** zu einem seriellen Schedule ist.
- **Konfliktäquivalente Schedules:**
  - Zwei Schedules  $S$  und  $S'$  sind konfliktäquivalent, wenn sie die gleichen Operationen in der gleichen Reihenfolge für jedes Datenobjekt haben, mit Ausnahme von Paaren aufeinanderfolgender Operationen, die **keinen Konflikt** aufweisen und deren Reihenfolge vertauscht wurde.
- **Kein Konflikt zwischen Operationen I und J:**
  - Sie gehören nicht zur gleichen Transaktion.
  - Mindestens eine der Operationen ist ein Lesevorgang.
  - Sie greifen nicht auf dasselbe Datenobjekt zu.
- **Konstruktion konfliktäquivalenter Schedules:** Durch wiederholtes Vertauschen nicht-konfliktierender, aufeinanderfolgender Operationen kann ein nebenläufiger Schedule in einen seriellen Schedule überführt werden, wenn er conflict serializable ist.

Wir tauschen die Reihenfolge so lange bis wir eine serialisierbare Schedule haben.

## Conflict Graph (Precedence Graphs)

- **Ziel:** Analyse der Konfliktserialisierbarkeit eines Schedules.
- **Konstruktion:** Gerichteter Graph wird für einen gegebenen Schedule erstellt.
- **Annahme:** Innerhalb einer Transaktion erfolgt ein `read` auf ein Datenobjekt immer vor einem `write` auf dasselbe Objekt.

Gegeben ist ein Schedule für die Transaktionen  $T_1, T_2, \dots, T_n$

- Die Knoten des Conflict Graphs sind die Transaktions-Ids.
- Eine Kante von  $T_i$  nach  $T_j$  zeigt einen Konflikt zwischen  $T_i$  und  $T_j$  an, wobei  $T_i$  den relevanten Zugriff früher durchführt.
- Manchmal werden Kantenlabels mit dem Namen des involvierten Datenobjekts annotiert.

Beispiel für einen Conflict Graph des Schedules  $S_1$



## Serialisierbarkeit feststellen

**Gegeben:**

- Ein Schedule  $S$
- Ein Conflict Graph

## Wie feststellen, ob $S$ conflict serializable ist?

Ein Schedule  $S$  ist conflict serializable, wenn der Conflict Graph azyklisch ist.

### Intuition:

- Ein **Konflikt** zwischen zwei Transaktionen erzwingt eine bestimmte Ausführungsreihenfolge.
- Die Conflict Serializability entspricht der Existenz einer **topologischen Sortierung** des Conflict Graphen.

### Warum Conflict Serializability?

Wir verwenden Conflict Serializability (und keine andere Definition der Serialisierbarkeit), weil es eine praktikable Implementierung gibt.

#### ☰ Beispiel für Conflict Graph

schedule $S_6$				
$T_{10}$	$T_{11}$	$T_{12}$	$T_{13}$	$T_{14}$
read(Y, y) read(Z, z)	read(X, x)			read(V, v) read(W, w) write(W, w)
	read(Y, y) write(Y, y)	read(Z, z) write(Z, z)	read(Y, y) write(Y, y) read(Z, z) write(Z, z)	
read(T, t)				
read(U, u)				

```

graph TD
    T10((T10)) --> T11((T11))
    T10((T10)) --> T12((T12))
    T11((T11)) --> T13((T13))
    T12((T12)) --> T13((T13))
    T14((T14)) --- T13((T13))
  
```

Wir haben 5 Knoten weil wir 5 einzelne Transaktionen haben.

Kanten werden erstellt, je nach dem was von was Abhängig ist. Also dadurch dass Y in T10 gelesen und in T11 und T13 beschrieben wird, zeigt T10 auf T11 und T12. T10 zeigt außerdem noch auf T13 wegen dem Z.

So macht man das dann für alle anderen Kanten auch.

Einfach die Operationen der Transaktionen durchgehen, schauen wo die Konflikte sind und dann die Kanten einzeichnen.

Was sagt uns der Graph jetzt?

- er hat keinen Zyklus

- --> Er ist Konflikt serialisierbar

verschiedene Ergebnisse

es gibt jetzt verschiedene Reihenfolgen die funktionieren:

$T_{10}, T_{11}, T_{12}, T_{13}$ , and  $T_{14}$  Yes

$T_{14}, T_{10}, T_{12}, T_{11}$ , and  $T_{13}$  Yes

$T_{14}, T_{13}, T_{12}, T_{11}$ , and  $T_{10}$  No

## Recoverable Schedules und Cascadeless Schedules

Transaktionen können fehlschlagen

- Abort vom User aus
- Rollback
- Irgendwelche Fehler

### Recoverable Schedules

- Wenn  $T_i$  fehlschlägt, muss die Transaktion zurückgesetzt werden, um die **Atomicity (Atomarität)** Eigenschaft zu erhalten (siehe Recovery).
- Wenn eine andere Transaktion  $T_j$  Daten gelesen hat, die von  $T_i$  geschrieben wurden, dann muss auch  $T_j$  zurückgesetzt werden.  
⇒ DBS muss sicherstellen, dass Schedules recoverable sind.
- Dieser Schedule ist nicht recoverable.

schedule $S_A$	
$T_i$	$T_j$
read(X, x)	read(X, x)
write(X, x)	write(X, x)
	commit
rollback	

Ein Schedule ist **recoverable**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt:

Wenn  $T_j$  Daten liest, die von  $T_i$  geschrieben wurden, dann muss  $T_i$  vor  $T_j$  committet werden.

schedule $S_A$	
$T_i$	$T_j$
read(X, x)	
write(X, x)	
rollback	
	read(X, x)
	write(X, x)
	commit

schedule $S_B$	
$T_i$	$T_j$
read(Y, y)	
write(Y, y)	
rollback	
	read(X, x)
	write(X, x)
	commit

recoverable

recoverable

## Cascading Rollbacks (Kaskadierendes Rücksetzen)

schedule $S_{11}$		
$T_{22}$	$T_{23}$	$T_{24}$
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
	read(A, a)	
		read(A, a)
		read(B, b)
rollback		

- $T_{22}$  Rollback  $\Rightarrow$  wir müssen auch  $T_{23}$  und  $T_{24}$  zurücksetzen, weil diese "dirty data" lesen. (Cascading Rollback)
- Der Schedule ist nicht cascadeless (kommt nicht ohne kaskadierendes Rücksetzen aus).
- Aber der Schedule ist recoverable.

Ist recoverable weil nicht committed wurde

## Cascadeless Schedules

Ein Schedule ist **cascadeless**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt: Wenn  $T_j$  Daten liest, die von  $T_i$  geschrieben wurden, dann muss  $T_i$  vor dem Lesezugriff von  $T_j$  bereits committed sein.

schedule $S_{11'}$		
$T_{22}$	$T_{23}$	$T_{24}$
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
rollback		
	read(A, a) commit	read(A, a) read(B, b) commit

Dieser Schedule ist auch recoverable

Cascading Rollbacks können verhindert werden, indem nur von Transaktionen gelesen wird, die bereits committed sind.

Ein Schedule ist **cascadeless**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt:

Wenn  $T_j$  Daten liest, die von  $T_i$  geschrieben wurden, dann muss  $T_i$  vor dem Lesezugriff von  $T_j$  bereits committed sein.

Vorteil:

- Cascading Rollbacks können verhindert werden, indem nur von Transaktionen gelesen wird, die bereits committed sind.

Jeder cascadeless Schedule ist auch recoverable.

Cascading Rollbacks:

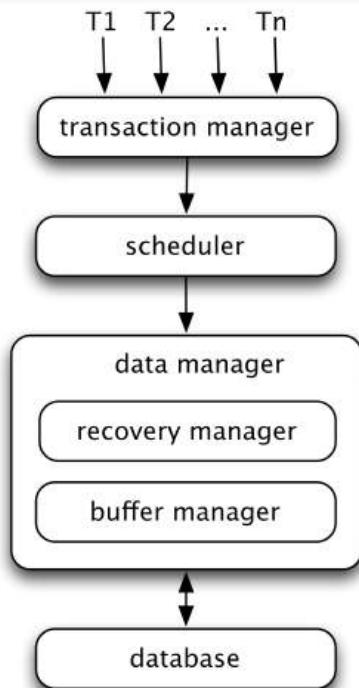
- Können schnell **zeitaufwändig** werden.

**Schlussfolgerung:**

- Es ist sinnvoll, sich auf Schedules zu beschränken, die **cascadeless** sind.

## Datenbank Scheduler

# Der Datenbank-Scheduler



Aufgabe des Schedulers:

Operationen der Transaktionen  $T_1, \dots, T_n$  in eine Reihenfolge bringen, die serialisierbar ist (und ohne kaskadierendes Rücksetzen auskommt).

## Synchronisationsverfahren

- Pessimistisch
  - Sperrbasierte Synchronisation
  - Zeitstempel-basierte Synchronisation
- Optimistisch

Basierend auf "Datenbanksysteme: Ein Einführung"  
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

## Sperrbasierte Synchronisation

- **Ziel:** Sicherstellen von (Konflikt-)serialisierbaren Schedules durch Verzögern von Transaktionen, die die Serialisierbarkeit verletzen würden.
- **Zwei Arten von Sperren auf ein Datenobjekt Q:**
  - S (shared lock, read lock, Lesesperrre)
  - X (exclusive lock, write lock, Schreibsperrre)
- **Sperroperationen:**
  - `lock_S(Q)` - Setzt ein Shared Lock auf Datenobjekt Q. Mehrere Transaktionen können gleichzeitig ein Shared Lock auf demselben Objekt halten.
  - `lock_X(Q)` - Setzt ein Exclusive Lock auf Datenobjekt Q. Nur eine Transaktion kann ein Exclusive Lock auf einem Objekt halten.
  - `unlock(Q)` - Freigabe des Locks auf Datenobjekt Q.

## Privilegien bei Locks

- Eine Transaktion, die
  - ein Exclusive Lock hält, darf einen **schreibenden oder lesenden Zugriff** durchführen.
  - ein Shared Lock hält, darf einen **lesenden Zugriff** durchführen.

## Verträglichkeitsmatrix (Kompatibilitätsmatrix)

	NL	S	X
S	OK	OK	-
X	OK	-	-

- **NL** - No Lock (keine Sperre)
- **OK** bedeutet, dass die Locks kompatibel sind und von verschiedenen Transaktionen gleichzeitig gehalten werden können.
- **-** bedeutet, dass die Locks inkompatibel sind und nicht gleichzeitig von verschiedenen Transaktionen gehalten werden können.
- **Wichtige Regeln:**
  - Nebenläufige Transaktionen dürfen nur kompatible Locks verwenden.
  - Eine Transaktion muss ggf. auf Freigabe eines Locks warten.

### ☰ Beispiel

#### Ein Beispiel

- $T_{15}$  überweist 50 EUR von Konto B auf Konto A.
- $T_{16}$  zeigt den gesamten Kontostand der Konten A und B.

$T_{15}$	$T_{16}$
<b>lock_X(B)</b>	
read(B, b)	
$b \leftarrow b - 50$	
write(B, b)	<b>lock_S(A)</b>
<b>unlock(B)</b>	read(A, a)
<b>lock_X(A)</b>	<b>unlock(A)</b>
read(A, a)	<b>lock_S(B)</b>
$a \leftarrow a + 50$	read(B, b)
write(A, a)	<b>unlock(B)</b>
<b>unlock(A)</b>	display(A+B)

Resultat bei serieller Ausführung:

$T_{16}$  zeigt 300

Bei diesem Schedule:

$T_{16}$  zeigt 250

**Ursache des Problems:**

Der Exclusive Lock auf B wurde zu früh freigegeben.

Initial: A = 100 und B = 200

#### Probleme bei zu früher Eingabe

schedule $S_7$		Probleme bei zu früher Freigabe
$T_{15}$	$T_{16}$	
<b>lock_X(B)</b> read(B, b) $b \leftarrow b - 50$ write(B, b) <b>unlock(B)</b>		<ul style="list-style-type: none"> <li>Initial: A = 100 und B = 200</li> <li>Serieller Schedule <math>T_{15};T_{16}</math> zeigt 300</li> <li>Serieller Schedule <math>T_{16};T_{15}</math> zeigt 300</li> <li><math>S_7</math> zeigt 250</li> </ul>
<b>lock_S(A)</b> read(A, a) <b>unlock(A)</b> <b>lock_S(B)</b> read(B, b) <b>unlock(B)</b> display(A+B)		<p><b>Frühe Sperrfreigaben</b> können zu <b>inkorrekt</b>en Resultaten führen (Non-Serializable Schedules), aber sie erlauben einen höheren Grad an Nebenläufigkeit.</p>
<b>lock_X(A)</b> read(A, a) $a \leftarrow a + 50$ write(A, a) <b>unlock(A)</b>		

### Probleme bei später Sperrfreigabe

Lösung: Verzögern wir einfach die Sperrfreigaben bis zum Ende der Transaktion

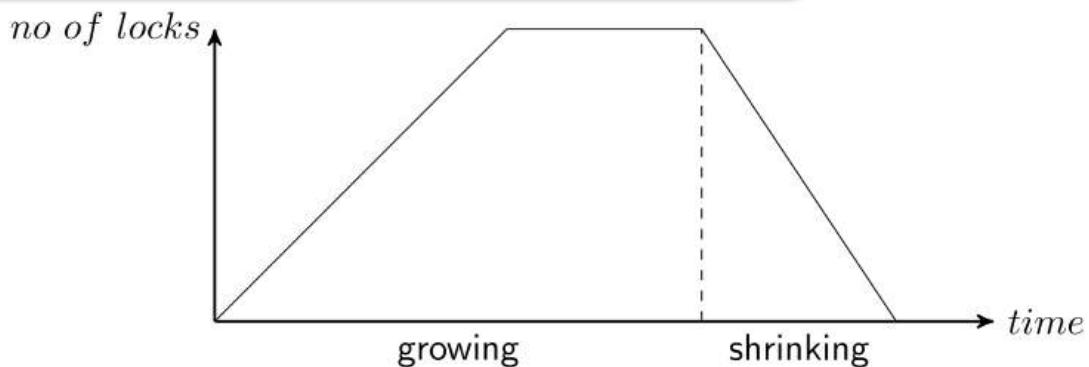
schedule $S_8$	
$T_{17}$	$T_{18}$
<b>lock_X(B)</b>	
read(B, b)	
$b \leftarrow b - 50$	
write(B, b)	
	<b>lock_S(A)</b>
	read(A, a)
...	...
<b>unlock(B)</b>	<b>unlock(A)</b>

- Späte Sperrfreigaben verhindern Non-Serializable Schedules, aber sie erhöhen die Chancen von **Deadlocks**.
- Lernen Sie damit zu leben!

## Das Zwei-Phasen-Sperrprotokoll (2PL)

- Ziel:** Sicherstellen der Konfliktserialisierbarkeit von Schedules.
- Besteht aus zwei Phasen:**
  - 1. Phase (Anforderungsphase, growing phase):**
    - Transaktionen dürfen Locks anfordern.
    - Transaktionen dürfen **keine** Locks freigeben.
  - 2. Phase (Freigabephase, shrinking phase):**
    - Transaktionen dürfen **keine** neuen Locks anfordern.

- Transaktionen dürfen bisher erworbene Locks freigeben.
- Übergang:** Sobald der erste Lock freigegeben wird, wechselt die Transaktion von der ersten in die zweite Phase.



Die Abbildung zeigt den typischen Verlauf der Anzahl der gehaltenen Locks über die Zeit während der Ausführung einer Transaktion unter dem 2PL-Protokoll. In der "growing" Phase steigt die Anzahl der Locks an oder bleibt konstant. Sobald die erste Freigabe erfolgt, beginnt die "shrinking" Phase, in der die Anzahl der gehaltenen Locks monoton fällt.

### ☰ Beispiele: 2PL: Ja oder nein?

schedule $S_A$		schedule $S_B$		schedule $S_C$		schedule $S_D$	
$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$		
lock_X(A)	lock_X(A)		lock_X(A)			lock_X(A)	
lock_X(B)	lock_X(B)			lock_X(B)		lock_X(B)	
lock_X(C)	lock_X(C)			lock_X(C)		unlock(B)	
unlock(A)	unlock(B)			unlock(C)		lock_X(C)	
unlock(C)		lock_X(B)		unlock(B)		unlock(A)	
unlock(B)	unlock(C)		unlock(A)			unlock(C)	
Ja		unlock(B)		Ja		Nein	
	Ja						

- Analyse der Schedules hinsichtlich des 2PL-Protokolls:**

- $S_A$ : Hält sich an 2PL. Die Transaktion  $T_1$  erwirbt alle Locks (A, B, C) und gibt sie dann frei. Es gibt keine Freigabe eines Locks vor der Anforderung eines neuen Locks.
- $S_B$ : Hält sich an 2PL. Die Transaktion  $T_2$  erwirbt alle Locks (A, B, C) und gibt sie dann frei.
- $S_C$ : Hält sich an 2PL. Die Transaktion  $T_3$  erwirbt Lock B und gibt es dann frei. Die Transaktion  $T_4$  erwirbt Lock A und gibt es dann frei.
- $S_D$ : Verstößt gegen 2PL. Die Transaktion  $T_6$  gibt Lock B frei, bevor sie Lock C anfordert. Dies verletzt die Regel, dass nach der Freigabe eines Locks keine neuen Locks mehr angefordert werden dürfen.

## Eigenschaften des Zwei-Phasen-Sperrprotokolls

- **2PL erzeugt nur serialisierbare Schedules:**
  - Garantiert Konfliktserialisierbarkeit.
  - 2PL erzeugt eine Untermenge aller möglichen serialisierbaren Schedules. Es gibt serialisierbare Schedules, die nicht durch 2PL erzeugt werden können.
- **2PL schützt nicht vor Deadlocks:** Auch wenn sich alle Transaktionen an das 2PL halten, können Deadlocks entstehen, wenn Transaktionen auf Locks warten, die von anderen Transaktionen gehalten werden, welche wiederum auf Locks warten, die die ersten Transaktionen halten.
- **2PL schützt nicht vor Cascading Rollbacks:** Wenn eine Transaktion Daten liest, die von einer anderen Transaktion geschrieben wurden, welche später abbricht (Rollback), muss die lesende Transaktion möglicherweise ebenfalls zurückgesetzt werden. 2PL alleine verhindert solche kaskadierenden Rollbacks nicht.
- **"Dirty" Reads sind möglich (Lesen von Transaktionen die noch nicht committed wurden):** Im Standard-2PL können Transaktionen Daten lesen, die von anderen Transaktionen geschrieben, aber noch nicht festgeschrieben (committed) wurden. Wenn die schreibende Transaktion später abbricht, hat die lesende Transaktion inkonsistente Daten gelesen.

## Cascading Rollbacks

- Ein Abort einer Transaktion kann zu einem Abort anderer Transaktionen führen.

schedule $S_{11}$			schedule $S_{11'}$		
$T_{22}$	$T_{23}$	$T_{24}$	$T_{22'}$	$T_{23'}$	$T_{24'}$
<b>lock_X(A)</b>			<b>lock_X(A)</b>		
<b>lock_X(B)</b>			<b>lock_X(B)</b>		
<b>unlock(A)</b>			<b>unlock(A)</b>		
	<b>lock_X(A)</b>		commit		
	<b>unlock(A)</b>			<b>lock_X(A)</b>	
abort		<b>lock_X(A)</b>		<b>unlock(A)</b>	
			commit		<b>lock_X(A)</b>
• Diese Schedules verwenden 2PI					

- **Beobachtung:** Diese Schedules verwenden 2PL.
- **Folge eines Aborts:** Abort in  $T_{22} \Rightarrow T_{23}$  und  $T_{24}$  müssen ebenfalls einen Abort auslösen (Cascading Rollback).
- **Wie können wir Cascading Rollbacks verhindern?**
  - Transaktionen dürfen keine **uncommitted Daten lesen** (siehe  $S_{11'}$ ). In  $S_{11'}$  liest  $T_{23'}$  keine uncommitteten Daten von  $T_{22'}$ , da  $T_{22'}$  die Daten (Lock auf A und B) erst freigibt, nachdem sie committed hat.

## Striktes und rigoroses Zwei-Phasen-Sperrprotokoll

### Striktes 2PL

- **Regel:** Exclusive Locks werden nicht vor dem Commit der Transaktion freigegeben.
- **Vorteil:** Verhindert "Dirty Reads", da keine uncommitted Daten gelesen werden können.

## Rigoroses 2PL

- **Regel:** Alle Locks (Shared und Exclusive) werden erst nach dem Commit der Transaktion freigegeben.
- **Eigenschaft:** Transaktionen können in der Commit-Reihenfolge serialisiert werden.

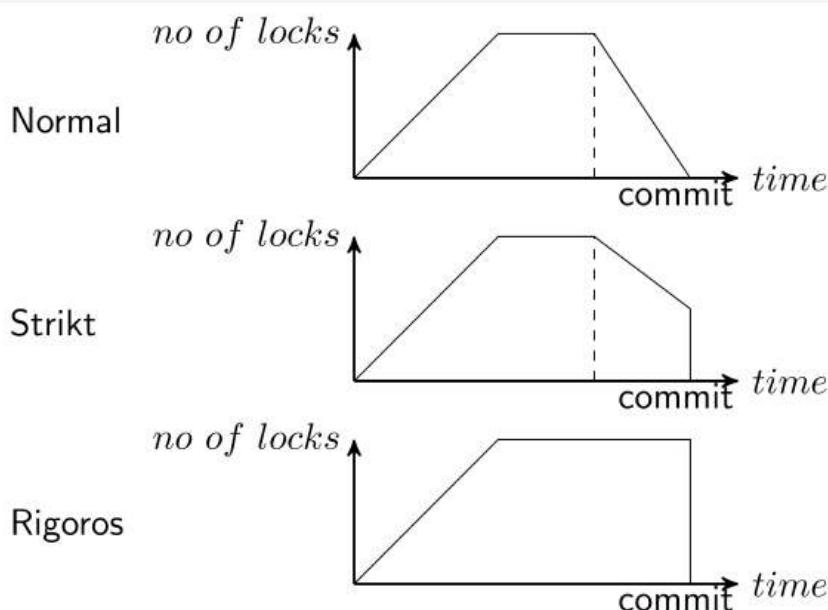
## Gemeinsamer Vorteil

- Keine Cascading Rollbacks, da keine uncommitteten Daten von anderen Transaktionen gelesen werden.

## Nachteil

- Weniger Nebenläufigkeit, da Locks länger gehalten werden und somit andere Transaktionen länger warten müssen.

## Übersicht 2PL Protokolle



## Vorteil von rigorosem 2PL gegenüber striktem 2PL

Rigoroses 2PL: Transaktionen können in Commit-Reihenfolge serialisiert werden.

## Konvertierung von Sperren

- **Ziel:** 2PL anwenden, aber einen höheren Grad an Nebenläufigkeit erlauben.
- **Erste Phase (Growing Phase):**
  - S-Lock erhalten (lock\_S)
  - X-Lock erhalten (lock\_X)

- Konvertieren (Upgrade) eines S-Locks zu einem X-Lock. Eine Transaktion, die bereits einen Shared Lock auf einem Datenobjekt hält, kann diesen in einen Exclusive Lock umwandeln, falls keine andere Transaktion ebenfalls einen Shared Lock auf demselben Objekt hält.
- **Zweite Phase (Shrinking Phase):**
  - Freigabe eines S-Locks ( unlock )
  - Freigabe eines X-Locks ( unlock )
  - Konvertieren (Downgrade) eines X-Locks zu einem S-Lock. Eine Transaktion, die einen Exclusive Lock hält, kann diesen in einen Shared Lock umwandeln.
- **Wichtiger Hinweis:**
  - Dieses Protokoll garantiert weiterhin Serialisierbarkeit,
  - ist aber abhängig vom Anwendungsprogrammierer (passende Lock-Operationen müssen explizit im Code eingefügt werden).

### ☰ Beispiele zu den Arten

schedule $S_1$	schedule $S_2$	schedule $S_3$
$T_1$	$T_2$	$T_3$
<b>lock_S(A)</b>	<b>lock_S(A)</b>	<b>lock_S(A)</b>
<b>lock_S(B)</b>	<b>lock_S(B)</b>	<b>lock_S(B)</b>
<b>lock_X(B)</b>	<b>lock_X(B)</b>	<b>lock_X(B)</b>
<b>lock_S(C)</b>	commit	<b>unlock(B)</b>
<b>unlock(A)</b>	Rigoros	<b>lock_S(C)</b>
<b>unlock(C)</b>		<b>unlock(A)</b>
commit		commit
Strikt		keine 2 Phasen

## Übersicht über 2PL-Schedules

## Alle Schedules

Konfliktserialisierbare Schedules

2PL-Schedules

Strikte 2PL-Schedules

Rigorose 2PL-Schedules

Serielle Schedules

## Erkennung von Deadlocks

### Deadlocks

- **Problem:** 2PL allein kann Deadlocks nicht verhindern.

$T_1$	$T_2$	
<b>lock_X(A)</b>		
read(A) write(A)	<b>lock_S(B)</b> read(B)	
<b>lock_X(B)</b>	<b>lock_S(A)</b>	$T_1$ muss warten auf $T_2$ $T_2$ muss warten auf $T_1$ ⇒ Deadlock
...	...	

- **Erläuterung des Deadlocks im Beispiel:**

- $T_1$  hält ein exklusives Lock auf A und versucht, ein exklusives Lock auf B zu erhalten.
- $T_2$  hält ein shared Lock auf B und versucht, ein shared Lock auf A zu erhalten.
- Da  $T_1$  ein exklusives Lock auf A hält, kann  $T_2$  kein shared Lock auf A erhalten.
- Da  $T_2$  ein shared Lock auf B hält, kann  $T_1$  kein exklusives Lock auf B erhalten (ein exklusives Lock ist inkompatibel mit einem shared Lock).
- Beide Transaktionen warten aufeinander und können nicht fortfahren, was zu einem Deadlock führt.

- **Lösungen für Deadlocks:**

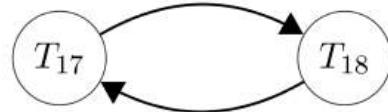
- **Erkennung von Deadlocks und anschließend Recovery:** Das System erkennt, wenn ein Deadlock aufgetreten ist, und ergreift Maßnahmen, um ihn aufzulösen (z.B. Abbruch einer der beteiligten Transaktionen).

- **Prävention:** Strategien, die verhindern, dass Deadlocks überhaupt entstehen können (z.B. durch die Art und Weise, wie Locks angefordert werden).
- **Timeout:** Eine Transaktion wartet nur eine bestimmte Zeit auf einen Lock. Wenn die Zeit abläuft, wird angenommen, dass ein Deadlock vorliegt, und die Transaktion wird abgebrochen.

## Erkennung von Deadlocks

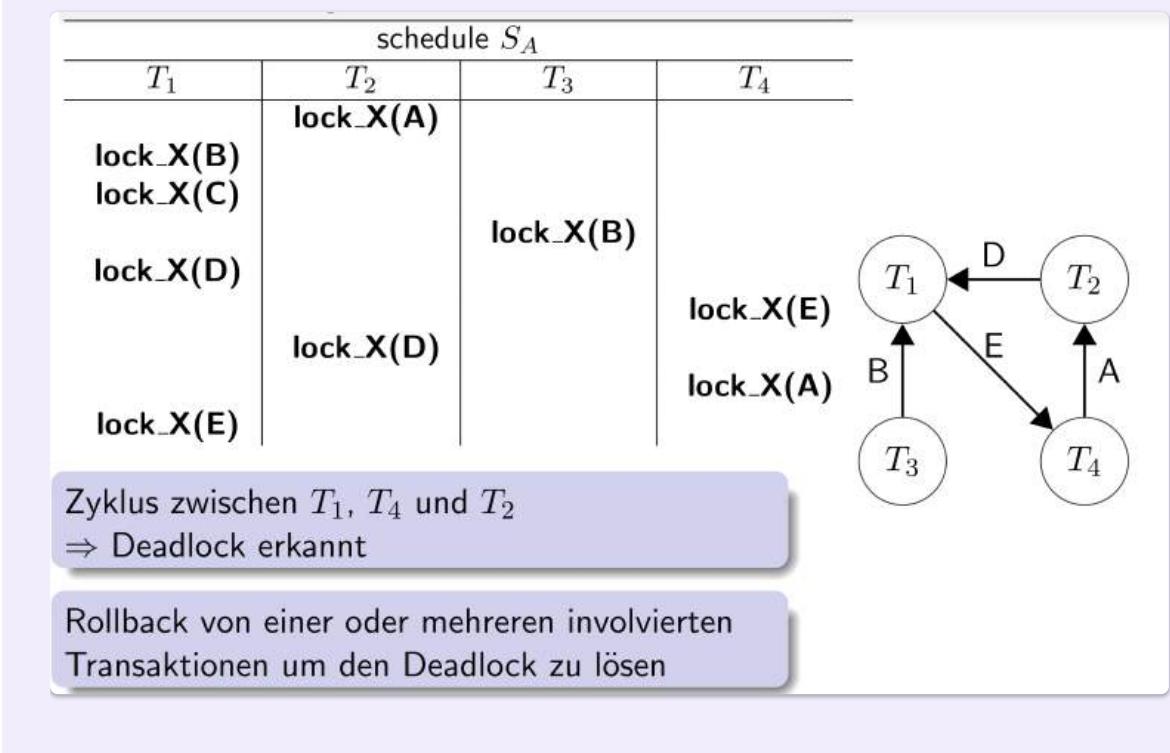
- **Methode:** Erstellen eines Wartegraphen ("Wait-for graph") und Prüfung auf Zyklen.
  - Ein Knoten für jede aktive Transaktion  $T_i$ .
  - Eine gerichtete Kante  $T_i \rightarrow T_j$  existiert, wenn Transaktion  $T_i$  auf die Lock-Freigabe von Transaktion  $T_j$  wartet.
  - **Ein Deadlock existiert, wenn der Wartegraph einen Zyklus enthält.**
- **Vorgehen bei Erkennung eines Deadlocks:**
  - Ein passendes Opfer auswählen (eine der Transaktionen im Zyklus).
  - Abbruch des Opfers und Freigabe aller zugehörigen Sperren, um den Zyklus im Wartegraph zu unterbrechen. Die abgebrochene Transaktion muss später neu gestartet werden.

schedule $S_8$	
$T_{17}$	$T_{18}$
<b>lock_X(B)</b>	
read(B, b)	
$b \leftarrow b - 50$	
write(B, b)	
	<b>lock_S(A)</b>
	read(A, a)
	<b>lock_S(B)</b>
<b>lock_X(A)</b>	



- **Wartegraph für den Schedule  $S_8$  im Deadlock-Zustand:**
  - $T_{17} \rightarrow T_{18}$  (da  $T_{17}$  auf ein Lock von  $T_{18}$  wartet - genauer gesagt, auf die Freigabe des Shared Locks auf B, um ein Exclusive Lock zu erhalten).
  - $T_{18} \rightarrow T_{17}$  (da  $T_{18}$  auf ein Lock von  $T_{17}$  wartet - genauer gesagt, auf die Freigabe des Exclusive Locks auf A, um ein Shared Lock zu erhalten).
  - Der Zyklus  $T_{17} \rightarrow T_{18} \rightarrow T_{17}$  zeigt einen Deadlock an.

### ☰ Beispiel



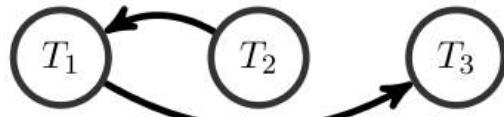
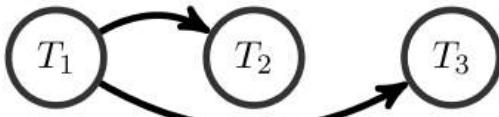
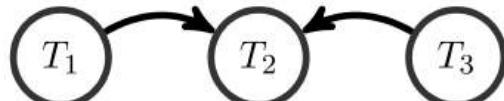
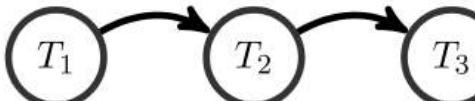
## Rollback Kandidaten

- **Wahl einer guten Opfer-Transaktion (bei Deadlock-Erkennung):**
  - Rollback von einer oder mehreren Transaktionen, die am Zyklus beteiligt sind, um den Deadlock aufzulösen.
- **Kriterien für die Auswahl des Opfers:**
  - **Die letzte Transaktion im Zyklus (Minimierung des Rollback-Aufwands):** Oft wird die Transaktion gewählt, die zuletzt in den Wartezyklus eingetreten ist, da sie möglicherweise weniger Arbeit verrichtet hat.
  - **Diejenige, welche die meisten Locks hält (Maximierung der freigegebenen Ressourcen):** Durch den Abbruch einer Transaktion mit vielen Locks werden mehr Ressourcen für andere Transaktionen freigegeben, was potenziell die Wahrscheinlichkeit weiterer Deadlocks verringert.
- **Vermeidung von Starvation:**
  - Aufpassen, dass nicht immer dasselbe Opfer gewählt wird (Starvation). Eine Transaktion könnte wiederholt als Opfer ausgewählt und immer wieder zurückgesetzt werden, ohne jemals zum Abschluss zu kommen.
  - **"Rollback Counter":** Eine mögliche Lösung ist ein Zähler für jede Transaktion, der festhält, wie oft sie bereits zurückgesetzt wurde. Ab einem gewissen Grenzwert wird diese Transaktion nicht mehr als Opfer für einen Rollback ausgewählt, um Starvation zu verhindern.

## Wertegraph

Welcher Wartegraph passt zu diesem Schedule?

schedule $S_A$		
$T_1$	$T_2$	$T_3$
	<b>lock_X(A)</b>	
<b>lock_X(A)</b> <b>lock_X(B)</b>		<b>lock_X(B)</b> <b>lock_X(C)</b>



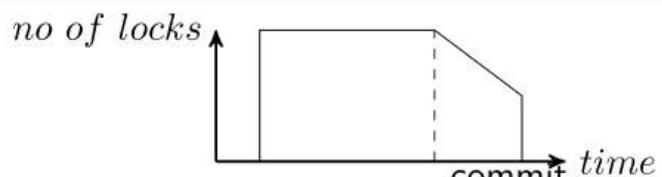
Hier passt nur Option 3 (links unten), weil  $T_1$  wartet auf  $T_2$  und muss auch auf  $T_3$  warten. Da es keinen Zyklus gibt, sieht das gut aus und wir haben keine Probleme / kein Deadlock

## Vermeidung von Deadlocks

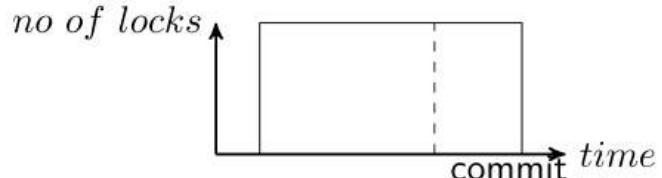
### Konservatives 2PL-Protokoll

- **Problem:** Normales, striktes und rigoroses 2PL können Deadlocks nicht verhindern.
- **Zusätzliche Anforderung im konservativen 2PL:** Alle benötigten Locks (Shared und Exclusive) werden **gleich zu Beginn** einer Transaktion gesetzt. Wenn nicht alle Locks sofort verfügbar sind, wartet die Transaktion, bis alle angefordert werden können.

konservatives striktes 2PL



konservatives rigoros 2PL



Die Abbildungen zeigen, dass im konservativen 2PL die Anzahl der gehaltenen Locks während der gesamten Lebensdauer der Transaktion bis zum Commit konstant bleibt. Es gibt keine "growing" oder "shrinking" Phase im herkömmlichen Sinne.

- **Verwendung:** Nur in wenigen Applikationen verwendbar, da es die Nebenläufigkeit stark einschränkt.

## Zusammenfassung Mehrbenutzersynchronisation

- Mehrere Protokolle zur Mehrbenutzersynchronisation wurden entwickelt.
  - **Hauptziel:** Nur serialisierbare, recoverable und cascadeless Schedules zuzulassen.
  - **Zwei-Phasen-Sperrprotokoll (2PL):** Ein weit verbreitetes Protokoll zur Sicherstellung der Konfliktserialisierbarkeit.
  - Die meisten relationalen DBMS benutzen **rigoroses 2PL**, um Serialisierbarkeit und die Vermeidung von Cascading Rollbacks zu gewährleisten.
- **Deadlock-Behandlung:**
  - **Erkennung von Deadlocks (Wartegraph)** und anschließendes Recovery (Rollback einer Transaktion).
  - **Verhindern von Deadlocks (konservatives 2PL):** Durch das sofortige Anfordern aller benötigten Locks wird die Entstehung von Zyklen in Wartegraphen vermieden.
- **Trade-off:** Serialisierbarkeit vs. Nebenläufigkeit. Strengere Serialisierungsprotokolle (wie rigoroses oder konservatives 2PL) reduzieren oft die mögliche Nebenläufigkeit.

## Recovery (Wiederherstellung)

---

Recovery ist ein essentieller Bestandteil von Datenbanksystemen, der die **Atomarität** und **Dauerhaftigkeit** von Transaktionen gewährleistet, selbst im Falle von Systemfehlern.

## Fehlerklassifikation und Grundprinzipien

- **Atomarität (Atomicity):**
    - Eine Transaktion ist eine unteilbare Einheit von Operationen. Entweder werden **alle** Operationen einer Transaktion erfolgreich abgeschlossen (Commit) und ihre Änderungen dauerhaft gemacht, oder **keine** von ihnen hat einen permanenten Effekt auf die Datenbank (Rollback/Abort).
    - Wenn eine Transaktion abbricht, müssen alle ihre Änderungen rückgängig gemacht werden.
  - **Dauerhaftigkeit (Durability):**
    - Sobald eine Transaktion committed wurde, müssen ihre Änderungen **dauerhaft** in der Datenbank gespeichert sein und auch Systemabstürze oder andere Fehler überdauern.
    - Die Recovery-Komponente des DBMS stellt sicher, dass diese Eigenschaften (Atomarität und Dauerhaftigkeit) trotz Fehlern aufrechterhalten werden.
  - **Garantie des DBMS:** Ein Datenbankmanagementsystem garantiert, dass eine Transaktion:
    - entweder **vollständig ausgeführt** wird und ein permanentes Ergebnis liefert (*committed*).
    - oder **keinen Effekt** auf die Datenbank hat (*aborted*).
-

## Wie kann Dauerhaftigkeit garantiert werden?

Dauerhaftigkeit ist ein komplexes Thema, da Daten auf verschiedenen Speicherebenen existieren und unterschiedliche Fehlerquellen berücksichtigen müssen.

- **Szenario 1: Daten im Arbeitsspeicher, Commit, aber noch nicht auf Festplatte geschrieben**
  - **Szenario:** Eine Transaktion ändert Daten im Arbeitsspeicher. Ein Commit wird durchgeführt, aber die Änderungen sind noch nicht physisch auf der Festplatte.
  - **Problem:** Bei einem **Blackout (Systemausfall)** gehen diese Änderungen unwiederbringlich verloren. Der Benutzer nimmt fälschlicherweise an, die Transaktion sei erfolgreich abgeschlossen.
- **Szenario 2: Daten teilweise auf Festplatte geschrieben, Commit**
  - **Szenario:** Eine Transaktion ändert Daten im Arbeitsspeicher. Einige Änderungen wurden auf die Festplatte geschrieben, andere noch nicht. Ein Commit erfolgt.
  - **Problem:** Bei einem **Blackout** ist die Datenbank **inkonsistent**, da nur ein Teil der Transaktionsänderungen persistent ist.
- **Szenario 3: Daten vollständig auf einer Festplatte geschrieben, Commit**
  - **Szenario:** Änderungen wurden vollständig auf die Festplatte geschrieben.
  - **Problem:** Ein **Hardware-Fehler** (z.B. Verlust der Festplatte) führt zum Datenverlust der Transaktion, obwohl sie committed war.
  - **Schlussfolgerung:** Das Schreiben auf eine einzelne Festplatte reicht nicht aus, um vollständige Dauerhaftigkeit gegen Hardware-Fehler zu garantieren.
- **Szenario 4: Daten vollständig auf mehreren lokalen Festplatten geschrieben (Redundanz)**
  - **Szenario:** Daten sind redundant auf mehreren lokalen Festplatten gespeichert.
  - **Problem:** Bei **katastrophalen Ereignissen** wie Feuer, Überschwemmung oder Erdbeben am physischen Standort können alle lokalen Kopien gleichzeitig zerstört werden.
  - **Schlussfolgerung:** Zusätzliche **externe Backups** und **Disaster-Recovery-Pläne** mit geografisch verteilten Daten sind notwendig.
- **Szenario 5: Daten vollständig auf mehreren Festplatten in geografisch verschiedenen Computerzentren**
  - **Szenario:** Daten sind redundant und geografisch verteilt gespeichert.
  - **Problem:** Bei einem **gleichzeitigen katastrophalen Ereignis** in **allen** Computerzentren (extrem unwahrscheinlich) wären dennoch alle Daten verloren.
  - **Schlussfolgerung:** Obwohl dies in der Praxis einen extrem hohen Grad an Ausfallsicherheit darstellt, zeigt es die fundamentale Abhängigkeit von der physischen Existenz der Datenträger.

## Dauerhaftigkeit (Durability)

- **Relativität:** Dauerhaftigkeit ist **relativ**. Je mehr Kopien an geografisch verteilten Orten existieren, desto höher ist die Ausfallsicherheit.
  - **Garantiebedingungen:** Dauerhaftigkeit kann nur garantiert werden, wenn:
    1. Die Kopien der Daten **zuerst persistent** aktualisiert werden (z.B. auf Festplatten geschrieben).
    2. **Erst dann** der Benutzer über den erfolgreichen Commit informiert wird.
  - **Write-Ahead Logging (WAL):** Dieses Prinzip stellt sicher, dass Änderungen an der Datenbank zuerst in einem **LogFile** protokolliert werden, bevor sie in die eigentliche Datenbank geschrieben werden. Das Log ermöglicht die Wiederherstellung nach einem Absturz.
    - Wir gehen davon aus, dass die WAL-Regel erfüllt ist.
  - **Strategien zur Erhöhung der Dauerhaftigkeit:**
    - **Log-Based Recovery:** Wiederherstellung des Datenbankzustands nach einem Fehler mithilfe des Transaktionslogs.
    - **Volle Redundanz:** Spiegelung aller Daten auf mehreren Computern/Festplatten/Rechenzentren, die identische Operationen ausführen.
- 

## Fehlerklassifikation im Detail

Datenbanksysteme müssen mit verschiedenen Arten von Fehlern umgehen können:

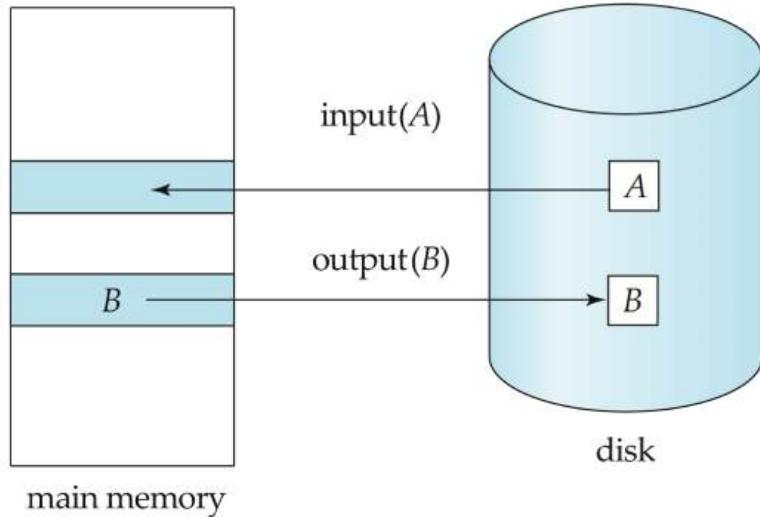
1. **Transaktionsfehler:**
  - Fehler, die innerhalb einer Transaktion auftreten, bevor sie committed wurde (z.B. logischer Fehler in der Anwendung, Constraint-Verletzung, Deadlock-Abbruch).
  - **Maßnahme:** Die Änderungen der betroffenen Transaktion müssen **rückgängig gemacht (Rollback)** werden, um die Atomarität zu gewährleisten.
2. **Systemabsturz (SystemCrash):**
  - Fehler, die zu einem Verlust des Hauptspeicherinhalts führen (z.B. Stromausfall, Betriebssystemfehler, DBMS-Softwarefehler).
  - **Anforderungen an die Recovery:**
    - Änderungen von **committeden** Transaktionen müssen **erhalten** bleiben (Dauerhaftigkeit).
    - Änderungen von **nicht committeden** Transaktionen müssen **rückgängig** gemacht werden (Atomarität).
3. **Festplattenfehler (DiskFailure):**
  - Fehler, die zu einem Verlust von Daten auf der Festplatte führen (z.B. Head-Crash, Bad Sectors).
  - **Maßnahme:** Die Wiederherstellung basiert in der Regel auf:

- **Archiven oder Datenbank-Dumps:** Regelmäßige Sicherungskopien der gesamten Datenbank.
- **Transaktionslogs:** Werden verwendet, um Änderungen seit dem letzten Backup wiederherzustellen (Log-Based Recovery), um den aktuellen Zustand der Datenbank zu erreichen.

## Datenspeicher

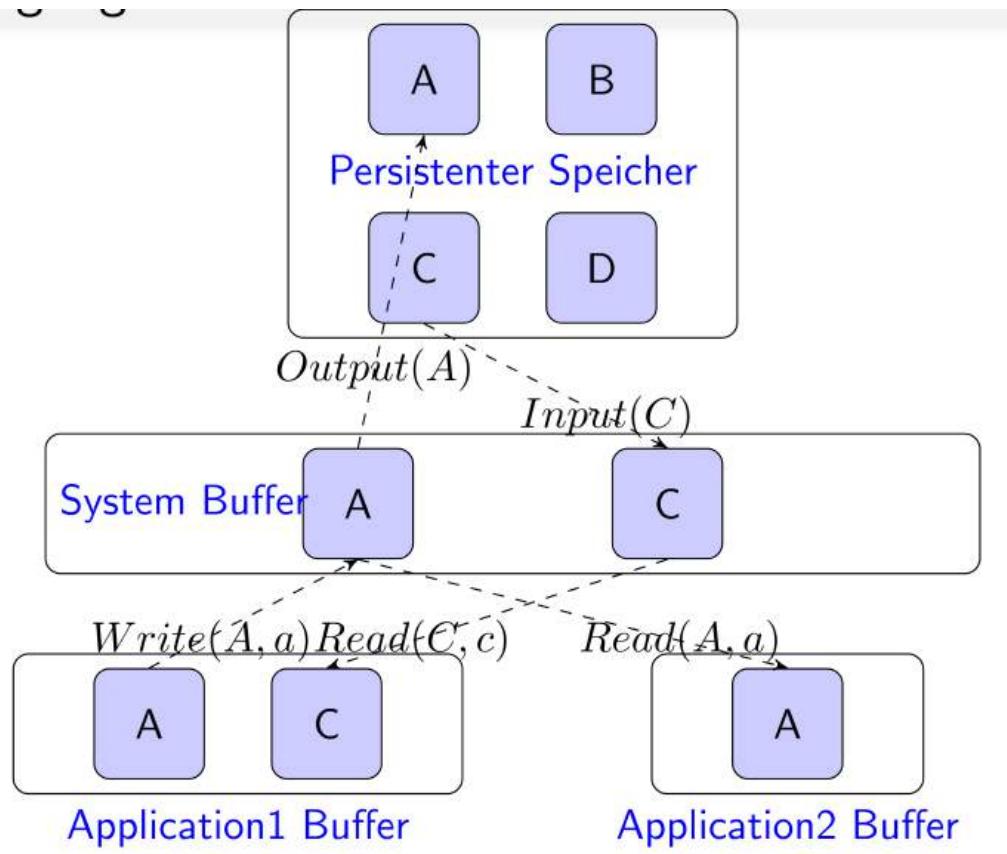
### Zweistufige Speicherhierarchie

Daten werden in Seiten (Pages) und Blöcken (Blocks) organisiert.



- **Flüchtiger Speicher (Volatile Storage)** (Arbeitsspeicher)
- **Nichtflüchtiger Speicher (Non-Volatile Storage)** (Festplatte)
- Stabiler Speicher (Stable storage) (RAIDS, Remote Backups, ...)

### Bewegung der Daten



## Speicheroperationen

- Transaktionen greifen auf die Datenbank zu und verändern Werte.
- Operationen, um Blöcke mit Datenobjekten zwischen der Festplatte und dem Arbeitsspeicher (System-Buffer) zu bewegen:**
  - Input(Q):** Transferiert den Block, der das Datenobjekt Q beinhaltet, in den Arbeitsspeicher. Dieser Vorgang ist notwendig, um auf die Daten zugreifen zu können.
  - Output(Q):** Transferiert den Block, der das Datenobjekt Q beinhaltet, auf die Festplatte. Dieser Vorgang dient dazu, Änderungen persistent zu speichern.
- Operationen, um Werte zwischen Datenobjekten und lokalen Variablen zu verschieben:**
  - read(Q, q):** Weist den Wert des Datenobjekts Q der lokalen Variable q der Transaktion zu.
  - write(Q, q):** Weist den Wert der lokalen Variable q dem Datenobjekt Q in der Datenbank (im Buffer) zu. Diese Änderung wird möglicherweise erst später mit Output(Q) auf die Festplatte geschrieben.

## Log-Einträge

### Die WAL-Regel für Log-Based Recovery

#### WAL (Write Ahead Logging)

- Regel 1 (Commit-Regel):** Bevor eine Transaktion in den Commit-Zustand wechselt, müssen **alle zugehörigen Log-Einträge** auf einen stabilen Speicher (z.B. Festplatte)

geschrieben worden sein, **inklusive dem Commit-Log-Eintrag** selbst. Dies stellt sicher, dass ein Commit auch nach einem Systemabsturz nachvollzogen werden kann.

- **Regel 2 (Write-Regel):** Bevor eine modifizierte Seite (oder ein Block) im Arbeitsspeicher in die Datenbank (nichtflüchtiger Speicher) geschrieben werden kann, müssen **alle zugehörigen Log-Einträge** für diese Änderung bereits auf einem stabilen Speicher geschrieben worden sein. Dies gewährleistet, dass die Log-Informationen zur Wiederherstellung verfügbar sind, falls ein Absturz während des Schreibens der Daten in die Datenbank erfolgt.

**Zusammenfassend:** Die WAL-Regel besagt, dass das Transaktionslog, das alle Änderungen und den Commit-Status festhält, immer persistent gespeichert sein muss, bevor die eigentlichen Datenbankänderungen auf die Festplatte geschrieben werden oder bevor eine Transaktion als committed gilt. Dies ist die Grundlage für eine zuverlässige Wiederherstellung nach Systemausfällen.

## Logging

Im normalen Betrieb

- **Transaktionsstart:** Am Beginn registriert sich eine Transaktion  $T$  selbst im Log: `[T start]`
- **Datenobjektänderung (write(X, x)):** Wenn ein Datenobjekt  $X$  durch die Transaktion  $T$  auf den neuen Wert  $x$  gesetzt wird:
  1. **Log-Eintrag:** Folgender Log-Eintrag wird dem Log hinzugefügt:
    - `[T, X, V-alt, V-neu]`
      - $T$ : Transaktions-ID
      - $X$ : Name des Datenobjekts
      - $V - alt$ : Alter Wert des Objekts vor der Änderung
      - $V - neu$ : Neuer Wert des Objekts nach der Änderung
  2. **Schreiben des neuen Werts:** Der neue Wert von  $X$  wird im Arbeitsspeicher (Buffer) aktualisiert. Der Buffer Manager schreibt diese geänderten Seiten später asynchron auf die Festplatte.
- **Transaktionsende (Commit):** Am Ende der Transaktion fügt Transaktion  $T$  den Eintrag `[T commit]` ins Log ein.
- **Commit-Definition:** Eine Transaktion gilt als **committed** genau dann, wenn der Commit-Eintrag (nach allen vorherigen Log-Einträgen der Transaktion) **im Log steht**. Dies ist ein zentraler Punkt für die WAL-Regel.
- **Zusätzliche Steuerungseinträge:**
  - **Start-Eintrag:** `[TID start]` - Markiert den Beginn der Transaktion mit der ID TID.
  - **Commit-Eintrag:** `[TID commit]` - Markiert das erfolgreiche Ende (Commit) der Transaktion mit der ID TID.
  - **Abort-Eintrag:** `[TID abort]` - Markiert das abgebrochene Ende (Abort) der Transaktion mit der ID TID.

## Beispiele

### ☰ Generelles Log-Einträge-Beispiel

schedule $S_1$			Log-Einträge Beispiel
$T_1$	$T_2$	$T_3$	
begin read(B, b) $b \leftarrow b+100$ write(B, b) commit	begin read(D, d) $d \leftarrow d+470$ write(D, d) commit	begin read(D, d) read(E, e) $d \leftarrow d-10$ write(D, d) $e \leftarrow e-20$ write(E, e) commit	[TID, DID, old, new] [T1 start] [T1, B, 300, 400] [T1 commit] [T2 start] [T2, D, 60, 530] [T2 commit] [T3 start] [T3, D, 530, 520] [T3, E, 70, 50] [T3 commit]

### ☰ Finde die Fehler

[T1 commit]

[T1, B, 300, 400]

[T1 start]

Commit vor Start

[T1 start]

[T1, B, 300, 400]

[T1 commit]

[T1, C, 40, 540]

Es darf keine Log-Einträge für T1 nach dem Commit geben.

[T1 start]

[T1, C, 40, 10]

[T1 start]

[T1, B, 300, 400]

[T1 commit]

[T1 start]

[T1, C, 40, 10]

[T1, B, 300, 400]

[T1 commit]

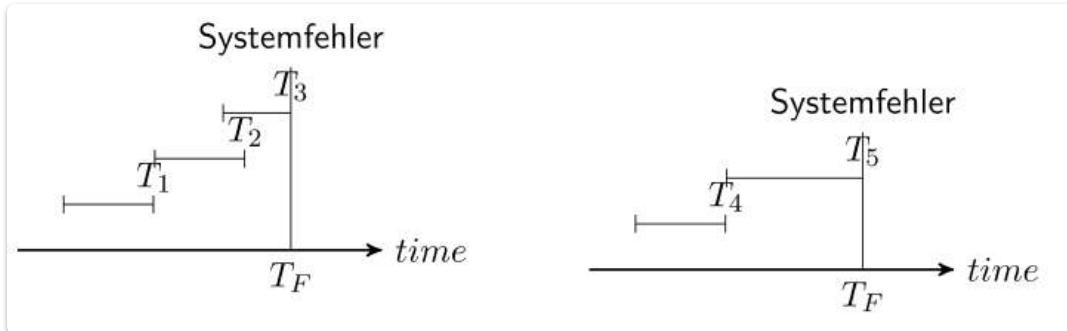
Korrekt!

Mehrere Start-Einträge für T1

## Log-Based Recovery

- **Operationen zur Wiederherstellung nach Fehlern:**

- **Redo:** Die Änderungen an der Datenbank wiederholen, die von zum Zeitpunkt des Systemfehlers committed waren.
- **Undo:** Den Zustand der Datenobjekte auf den Wert vor der Ausführung der Operationen von Transaktionen zurücksetzen, die zum Zeitpunkt des Systemfehlers nicht committed waren (aborted oder noch aktiv).



Die Abbildungen zeigen den Zeitpunkt des Systemfehlers ( $T_F$ ) im Verhältnis zur Lebensdauer verschiedener Transaktionen ( $T_1, T_2, T_3, T_4, T_5$ ).

- **Analyse der Transaktionen im Hinblick auf Redo/Undo:**

- **Linke Abbildung (Systemfehler nach  $T_3$ ):**
  - **Redo  $T_1$  und  $T_2$ :** Da  $T_1$  und  $T_2$  vor dem Systemfehler committed wurden, müssen ihre Änderungen in der Datenbank wiederhergestellt werden (Redo).
  - **Undo  $T_3$ :** Da  $T_3$  zum Zeitpunkt des Systemfehlers noch nicht committed war, müssen alle ihre bisherigen Änderungen rückgängig gemacht werden (Undo).
- **Rechte Abbildung (Systemfehler nach  $T_5$ ):**
  - **Redo  $T_4$ :** Da  $T_4$  vor dem Systemfehler committed wurde, müssen ihre Änderungen wiederholt werden (Redo).
  - **Undo  $T_5$ :** Da  $T_5$  zum Zeitpunkt des Systemfehlers nicht committed war, müssen ihre Änderungen rückgängig gemacht werden (Undo).

## Drei Recovery-Phasen

### Phase 1:

- **Redo (Vollständige Wiederholung der Historie)**
  - Alle Log-Einträge werden Schritt für Schritt in chronologischer Reihenfolge durchgegangen (vorwärts).
  - Alle im Log verzeichneten Änderungen werden in derselben Reihenfolge auf die Datenbank angewendet (Redo).
- **Bestimmen der "Undo"-Transaktionen:**
  - Für jeden `[T_i start]` Eintrag im Log wird die Transaktion  $T_i$  zur "Undo List" hinzugefügt. Dies sind potenziell unvollständige Transaktionen.
  - Wenn ein `[T_i commit]` oder `[T_i abort]` Eintrag für eine Transaktion  $T_i$  gefunden wird, wird  $T_i$  von der "Undo List" entfernt, da diese Transaktion entweder erfolgreich abgeschlossen oder explizit abgebrochen wurde. Die

Transaktionen, die am Ende dieser Phase noch in der "Undo List" sind, waren zum Zeitpunkt des Systemfehlers aktiv und nicht committed.

## Phase 2:

- **Undo (Rollback aller Transaktionen in der "Undo List")**
  - Alle Log-Einträge werden **rückwärts** durchgegangen (vom Ende zum Anfang).
  - Für jede Transaktion  $T_i$  in der "Undo List" werden alle ihre im Log verzeichneten Änderungen rückgängig gemacht (Undo).
  - Für jede rückgängig gemachte Operation wird ein **Compensation-Log-Eintrag** erstellt, der die inverse Operation protokolliert. Dies ist wichtig für den Fall eines erneuten Fehlers während der Undo-Phase.
  - Für jeden `[T_i start]` Eintrag einer Transaktion  $T_i$  in der "Undo List" wird ein `[T_i abort]` Eintrag ins Log geschrieben und  $T_i$  von der "Undo List" entfernt, da alle ihre Änderungen nun rückgängig gemacht wurden.
  - Die Undo-Phase stoppt, sobald die "Undo List" leer ist, d.h., alle unvollständigen Transaktionen zurückgesetzt wurden.

## Compensation-Log-Einträge

- **Format:** `[TID, DID, value]`
  - **TID:** ID der Transaktion, die den Ausgleich vornimmt (in der Recovery-Phase ist dies oft eine spezielle Systemtransaktion).
  - **DID:** ID des betroffenen Datenobjekts.
  - **value:** Der Wert, auf den das Datenobjekt zurückgesetzt wird (der ursprüngliche Wert vor der Änderung der unvollständigen Transaktion).
- **Zweck:** Erstellt zum Rückgängigmachen (Ausgleichen/Kompensieren) der Änderungen, die durch einen vorherigen Log-Eintrag der Form `[TID, DID, value, newValue]` einer unvollständigen Transaktion verursacht wurden. Der Compensation-Log-Eintrag stellt somit den ursprünglichen Zustand wieder her.
- **Redo-Only-Log-Eintrag:** Compensation-Log-Einträge sind in der Regel "Redo-Only". Das bedeutet, dass während der Redo-Phase eines späteren Recovery-Prozesses diese Einträge angewendet werden, um sicherzustellen, dass die Rückgängigmachung der unvollständigen Transaktion bestehen bleibt. Es ist nicht notwendig, Compensation-Log-Einträge selbst wieder rückgängig zu machen.
- **Verwendung im normalen Betrieb:** Compensation-Log-Einträge können auch für einen expliziten Rollback einer Transaktion während der normalen Ausführung verwendet werden. In diesem Fall werden sie erzeugt, um die bereits erfolgten Änderungen der Transaktion zu neutralisieren.

## ARIES

- **State-Of-The-Art Methode** für Log-Based Recovery.

- **Erweitert den vorgestellten Algorithmus** um einige Optimierungen und einer Vorab-Phase: Durchgehen des Logs, um Dirty-Pages zu identifizieren und um den „Startpunkt“ des Logs sowie die „Undo“-Transaktionen zu bestimmen.
- **Behandlung von Dirty Pages:** Verwendet eine Dirty Page Table (pageID, recLSN), eine Erweiterung der Pages (pageLSN, letzter Log-Eintrag der Änderungen vorgenommen hat).
- **Log-Einträge in ARIES haben eine Log Sequence Number (LSN):**  
[LSN, TransactionID, PageID, redoValue, undoValue, prevLSN]
- **Compensation Log-Eintrag in ARIES (CLR):**  
[LSN, TransactionID, PageID, redoValue, prevLSN, undoNxtLSN]

# 6. Physischer Datenbankentwurf

## Einleitung

---

### Specherebenen

- **Primärspeicher (Volatile Storage)**
    - z.B. Cache, Hauptspeicher, Register
    - Verliert den Inhalt, wenn der Strom abgeschaltet wird
    - Zugriffszeiten:
      - Register: 1-10 ns
      - Cache: 10-100 ns
      - Hauptspeicher: 100-1000 ns
  - **Sekundärspeicher (Non-Volatile Storage)**
    - z.B. Festplatte
    - Behält den Inhalt, wenn der Strom abgeschaltet wird
    - Zugriffszeit: ca. 10 ms (deutlich langsamer als Primärspeicher)
  - **Tertiärspeicher (Non-Volatile Storage)**
    - z.B. Magnetbänder (Archivspeicher)
    - Behält den Inhalt, wenn der Strom abgeschaltet wird
    - Zugriffszeit: sec (noch langsamer als Sekundärspeicher)
  - Je höher das Level (im Mehrebenenmodell), desto schneller der Zugriff.
  - **Zugriffslücke:** Die Diskrepanz in der Zugriffszeit zwischen Hauptspeicher und Plattenspeicher ist sehr groß ( $10^5$ ).
- 

### HDD (Magnetische Festplatte)

- Besteht aus:
    - Platten
    - Spuren (virtuell, konzentrisch auf den Platten)
    - Sektoren (segmentieren die Spuren)
    - Achse (um die sich die Platten drehen)
    - Arm mit Festplattenkopf (zum Lesen und Schreiben)
  - Meist sind Datenbanken auf magnetischen Festplatten gespeichert.
  - Datenbank ist oft zu groß für den Hauptspeicher.
  - Plattenspeicher ist persistent (nicht-flüchtig).
  - Plattenspeicher ist billiger als Hauptspeicher.
-

## Zugriffsoptimierung für Festplatten

Für jeden Speicherzugriff auf die Festplatte fallen folgende Zeiten an:

- **Seek Time:** Zeit, um den Lese-/Schreibkopf zur richtigen Spur zu bewegen.
- **Rotational Delay (Latenzzeit):** Zeit, bis der gewünschte Sektor unter dem Lese-/Schreibkopf rotiert ist.
- **Transfer Time:** Zeit, um die Daten tatsächlich zu übertragen.
- Jede **Spur (Track)** ist unterteilt in **Sektoren**.
- Eine **Seite (Block/Page)** ist eine kontinuierliche Sequenz von **Sektoren** eines einzigen Tracks.
- Die kleinste Einheit, die zwischen Festplatte und Hauptspeicher transferiert wird, ist eine Seite (Block/Page).

## Optimierung

- **Blöcke in der Reihenfolge anordnen**, in der sie auch benötigt werden (sequentielle Anordnung minimiert Seek Time und Rotational Delay).
- **Verwandte Informationen nahe beieinander ablegen** (reduziert die Notwendigkeit großer Kopfbewegungen).

Grundsätzlich werden relationale Daten als Sequenzen von Bits auf Festplatten gespeichert.

---

## Anforderungen an DBS

### Funktionale Anforderungen an Datenbanksysteme

- **Tupel (Records) sequenziell abarbeiten** können.
- **Effiziente Key-Value-Suche** ermöglichen.
- **Einfügen/Löschen von Tupeln (Records)** unterstützen.

### Performanzanforderungen an Datenbanksysteme

- **Wenig Speicherplatz verschwenden**.
- **Schnelle Antwortzeiten** gewährleisten.
- **Hoher Durchsatz von Transaktionen** ermöglichen.

## Dateiorganisation - Dateien und Tupel (Records)

### Dateiorganisation

#### Speichern von Datenbanken auf Festplatten

- Eine Datenbank wird als Menge von **Dateien (Files)** gespeichert.

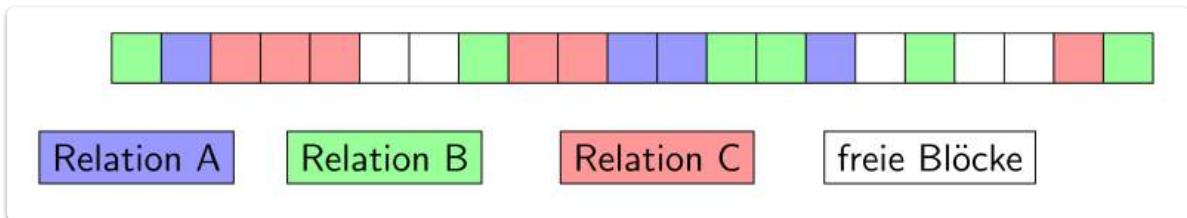
- Jede Datei enthält eine Menge von **Tupeln (Records)**.
- Ein Tupel/Record enthält eine bestimmte Anzahl von **Feldern (Fields)**.

*Mehrere Records werden in **Seiten/Blöcken (Pages/Blocks)** zusammengefasst (die Einheit für den Datentransfer zwischen Festplatte und Hauptspeicher).*

## Dateien auf Massenspeichern (normalerweise Festplatten)

- Ermöglichen schnellen Zugriff auf beliebige Tupel (im Vergleich zu sequentiellen Speichermedien wie Bändern).
- Sind Teil des Sekundärspeichers.

## Beispiel wie Relationen auf verschiedenen Blöcken gespeichert sein können



## Größen

### ⌚ Größe eines Records

- **Feste Größe (Fixed Size):** Alle Records in einer Datei haben die gleiche Länge. Dies erleichtert die Berechnung der Speicheradresse eines bestimmten Records.
- **Variable Größe (Variable Size):** Records in einer Datei können unterschiedliche Längen haben. Dies erfordert zusätzliche Informationen, um die Grenzen zwischen Records zu identifizieren (z.B. Längenangaben).

## Feste Größe (Fixed Size Records)

### ⓘ Löschen eines Tupels i

- Es gibt verschiedene Strategien, um die durch das Löschen entstandene Lücke zu schließen

### Verschieben von Tupeln:

- Verschiebe die nachfolgenden Tupel ( $i+1, \dots, n$ ) an die Positionen  $i, \dots, n-1$ , um die Lücke zu schließen. Dies ist aufwendig, da viele Datensätze bewegt werden müssen.
- Oder verschiebe das letzte Tupel ( $n$ ) an die Position  $i$ . Dies ist effizienter, hinterlässt aber möglicherweise eine Lücke am Ende.

## Markieren der Lücken:

- Markiere die Lücke als gelöscht und fülle sie später mit neuen Tupeln auf. \* **Free-List**: Eine Möglichkeit, die freien Lücken zu verwalten:
  - Markiere die erste Lücke im File-Header, um den Beginn der Liste freier Blöcke zu kennzeichnen.
  - Benutze die Lücken selbst, um auf weitere Lücken zu verweisen (verkettete Liste freier Speicherbereiche).

## Variable Größe (Variable Size Records)

- Tupel haben **unterschiedliche Größen** und beanspruchen unterschiedlich viel Speicherplatz.
- **Beispiel**: Attribute mit variabler Länge wie `varchar`.

Alternativen für variable Größe

- **Wenn maximale Größe bekannt**: Abbildung auf Tupel fester Größe (führt möglicherweise zu Speicherplatzverschwendungen, wenn die tatsächliche Größe oft kleiner ist).
- **Slotted-Page-Structure**: Eine effizientere Methode zur Speicherung von Tupeln variabler Größe innerhalb eines Blocks (Seite):
  - Tupel werden fortlaufend im Datenbereich des Blocks gespeichert.
  - Ein **Block Header** enthält **Pointer (Zeiger)** zu allen Tupeln innerhalb des Blocks sowie Informationen über den freien Speicherplatz.
  - Beim Zugriff auf ein Tupel wird der Pointer im Header verwendet, um die tatsächliche Position und Größe des Tupels im Datenbereich zu finden.
  - Dies ermöglicht effiziente Updates und Löschungen, da nur die Pointer im Header angepasst werden müssen, ohne die Tupel selbst verschieben zu müssen (innerhalb des Blocks).

## Organisation von Tupeln in Dateien

Bestimmen der Tupelreihenfolge innerhalb der Datei: Wie werden die Datensätze physisch in der Datei angeordnet?

## Heap Files

- Tupel können an **beliebiger Position** in der Datei abgelegt werden.
- Es gibt keine spezielle Ordnung. Neue Tupel werden einfach am Ende der Datei angehängt oder in freie Lücken eingefügt.
- Suche nach einem bestimmten Tupel erfordert möglicherweise das Durchsuchen der gesamten Datei.

## Sequenzielle Dateiorganisation

- Tupel werden **sequenziell in Reihenfolge des Search Keys** (z.B. ein bestimmtes Attribut) abgelegt.
- Die physische Reihenfolge der Datensätze entspricht der logischen Ordnung basierend auf dem Suchschlüssel.
- Dies ermöglicht effizientere Bereichsabfragen basierend auf dem Suchschlüssel.
- Einfügungen und Löschungen können aufwendig sein, da möglicherweise viele Datensätze verschoben werden müssen, um die Reihenfolge beizubehalten.
- **Suche: Binäre Suche** bei Suche anhand des Search-Keys möglich, was deutlich effizienter ist als der Linear Scan bei Heap Files (logarithmische Komplexität).
- **Einfügen:** Erfordert möglicherweise eine **Reorganisation der Datei**, um die Sortierreihenfolge beizubehalten. Neue Tupel müssen an der richtigen Position eingefügt werden, was das Verschieben anderer Tupel erfordern kann.
- Search Key kann aber muss nicht der Primary key sein.

## Hash Files

- Benutze eine **Hashfunktion**, um die **Position eines Tupels innerhalb der Datei zu bestimmen**.
- Der Wert des Suchschlüssels wird durch die Hashfunktion abgebildet, um die Speicheradresse des Tupels zu ermitteln (ungefähre Position).
- Ermöglicht sehr schnelle Zugriffe auf einzelne Tupel basierend auf dem Suchschlüssel (im Idealfall konstanter Zeitaufwand).
- Bereichsabfragen sind in der Regel ineffizient, da die gehaschten Adressen keine direkte Ordnung widerspiegeln.

## Indexstrukturen

---

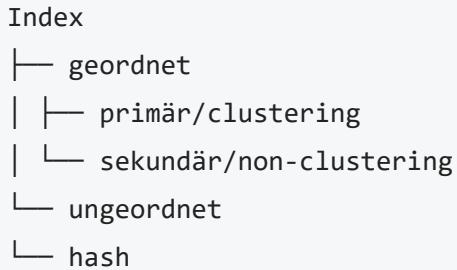
### Annahmen (im Kontext von Indexstrukturen)

- In einer Datenbank werden **viele Tupel abgelegt**.
- Viele Anfragen greifen je auf eine **kleine Menge von Tupeln** zu.
- Tupel müssen **verändert werden können** (Insert, Update und Delete).
- Die Datenbank kann **nicht im Offline-Zustand versetzt werden**, um eine Reorganisation durchzuführen (Indexe müssen dynamisch verwaltet werden).

**Ziel:** so wenig Daten wie möglich lesen, um die Performance von Datenbankabfragen zu optimieren.

### Übersicht: Indexstrukturen

## Klassifikation von Indexen:



- **Geordnet:** Der Index basiert auf einer sortierten Struktur, die effiziente Bereichsabfragen ermöglicht.
  - **Primär/Clustering Index:** Die physische Speicherung der Datensätze auf der Festplatte ist an die Sortierreihenfolge des Indexschlüssels angepasst. Es kann nur einen Clustering Index pro Tabelle geben.
  - **Sekundär/Non-Clustering Index:** Der Index ist eine separate Struktur, die Pointer zu den tatsächlichen Datensätzen enthält. Die physische Speicherung der Daten ist unabhängig von der Indexreihenfolge. Es können mehrere Non-Clustering Indexe pro Tabelle existieren.
- **Ungeordnet:** Der Index basiert auf einer Hash-Struktur, die schnelle Punktabfragen ermöglicht, aber ineffizient für Bereichsabfragen ist.
  - **Hash Index:** Verwendet eine Hashfunktion, um die Speicheradresse der Datensätze zu finden (ähnlich wie bei Hash Files).

## Variationen:

- **Single-Level Index:** Der Index besteht aus einer einzigen Ebene.
- **Multi-Level Index:** Der Index ist hierarchisch aufgebaut (z.B. B-Baum), um die Suche in sehr großen Datenbeständen zu beschleunigen.

Es ist möglich, **mehrere Indexe auf der gleichen Tabelle zu definieren**, um unterschiedliche Arten von Abfragen effizient zu unterstützen. Die Wahl der Indexe hängt von den häufigsten Abfragemustern ab.

## Übersicht: Primär vs. Sekundär & Dense vs. Sparse Index

### ⌚ Primär (Primary) vs. Sekundär (Secondary)

- = **Clustering vs. Non-Clustering** (synonyme Begriffe)
- **Entscheidende Frage:** Ist die **Datei nach dem Search-Key sortiert?**
  - **Ja:**  $\Rightarrow$  **Primär (Clustering) Index.** Die physische Anordnung der Daten auf der Festplatte entspricht der Sortierreihenfolge des Index-Schlüssels.
  - **Nein:**  $\Rightarrow$  **Sekundär (Non-Clustering) Index.** Der Index ist eine separate Struktur, die Pointer zu den Datensätzen enthält, welche in einer anderen (oder

keiner bestimmten Reihenfolge auf der Festplatte gespeichert sind.

### ⌚ Dense vs. Sparse

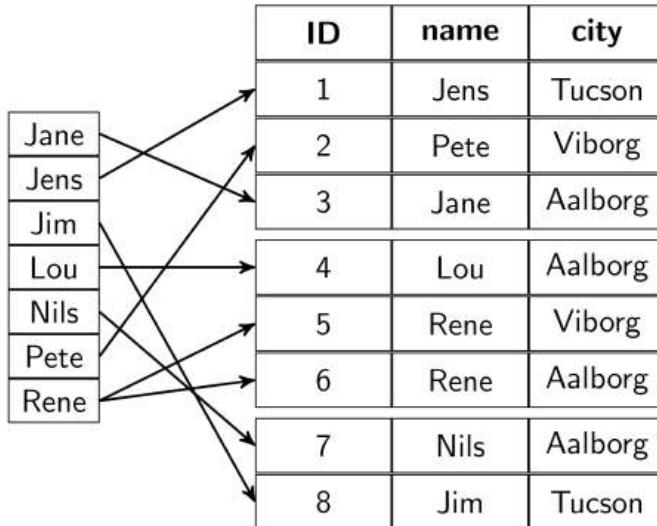
- **Entscheidende Frage:** Gibt es einen **separaten Eintrag im Index für jedes Tupel bzw. jeden vorkommenden Wert des Search-Keys?**
  - **Ja:**  $\Rightarrow$  **Dense Index** (dichter Index). Jeder Datensatz (oder jeder eindeutige Wert des Suchschlüssels) hat einen Eintrag im Index, der auf den tatsächlichen Datensatz zeigt.
  - **Nein:**  $\Rightarrow$  **Sparse Index** (dünner Index). Nur für einige Werte des Suchschlüssels existiert ein Eintrag im Index. Diese Einträge zeigen auf den Block, der den Datensatz mit diesem Suchschlüssel enthält. Um einen bestimmten Datensatz zu finden, muss möglicherweise der gesamte Block durchsucht werden. Sparse Indexe sind in der Regel kleiner als Dense Indexe.

### ☰ Beispiel - Primary Sparse Index

ID	name	city
1	Jens	Tucson
2	Pete	Viborg
3	Jane	Aalborg
4	Lou	Aalborg
5	Rene	Viborg
6	Rene	Aalborg
7	Nils	Aalborg
8	Jim	Tucson

- Definiert auf einer nach dem Search-Key sortierten Datei
- Ein Eintrag pro Seite/Block in der Datei

### ☰ Beispiel - Secondary Dense Index



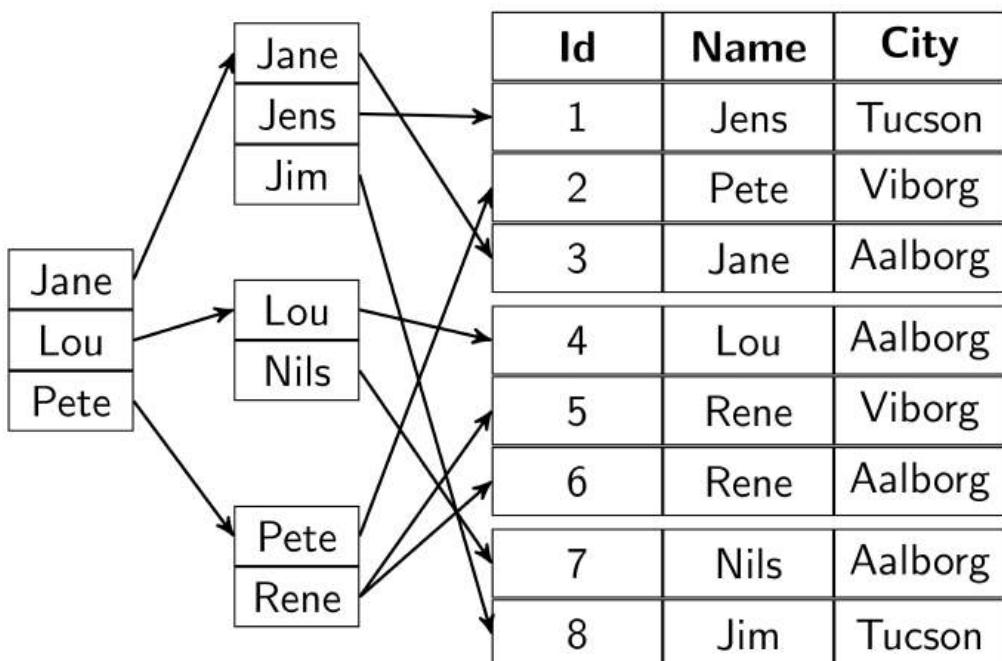
- Definiert auf einer nicht nach dem Search-Key sortierten Datei
- Ein Eintrag pro Tupel

## Tradeoff

- **Dense Index:** Ermöglicht **schnelleres Finden von Tupeln** in einigen Fällen, da der Indexeintrag direkt auf den Datensatz verweist.
- **Sparse Index:** Benötigt **weniger Speicherplatz**, da nicht für jeden Datensatz ein Eintrag existiert. Dies kann die Performance beim Auffinden eines Datensatzes erhöhen, da weniger Indexseiten durchsucht werden müssen, um den relevanten Block zu finden.

## Multi-Level Index

Bei Multi-Level Indexen hat man Indexe die auf andere Indexe zeigen.



- Ziel: der äußere Index (sparse) passt in den Hauptspeicher
- Der Index kann mehr als 2 Ebenen haben
- Einfügen und Löschen:
  - Kann zu einer **teuren Reorganisation von mehreren Ebenen von sortierten Dateien** führen, um die Sortierreihenfolge beizubehalten. Dies ist besonders aufwendig bei sequenzieller Dateiorganisation.

## Kombinationen von Konzepten (Indexarten)

	dense	sparse
primary (clustering)	✓	✓
secondary (non-clustering)	✓	%

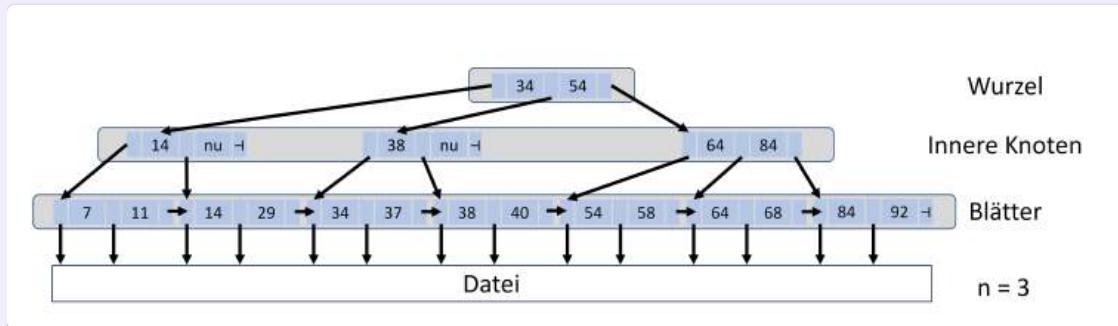
Ein Secondary Sparse Index ist nicht sinnvoll!

- Da die Tupel **nicht anhand des Search-Keys sortiert** sind (bei einem sekundären Index ist die physische Reihenfolge der Daten unabhängig vom Indexschlüssel), können wir anhand des ersten Tupels einer Seite **keine Rückschlüsse auf die restlichen Tupel der Seite** schließen. Ein Eintrag im Sparse Index würde nur auf den Anfang einer Seite zeigen, aber wir wüssten nicht, wo sich die anderen Tupel mit dem gesuchten Schlüssel (oder einem ähnlichen Schlüssel) auf dieser Seite befinden.
- **Daher sind Secondary Indexes immer dense.** Für jeden eindeutigen Wert des Suchschlüssels (oder für jede Seite, je nach Implementierung) muss ein Eintrag im Index vorhanden sein, um direkt zum entsprechenden Datensatz oder zur entsprechenden Seite navigieren zu können.

- **Clustering dense:** Indexeintrag für **jeden vorkommenden Wert des Search-Keys**  $\Rightarrow$  der Indexeintrag zeigt auf das **erste Tupel** mit diesem Wert (da die Daten physisch sortiert sind).
- **Non-clustering dense:** Indexeinträge für **alle Tupel** (oder für jede Seite, die Tupel mit dem entsprechenden Schlüssel enthält). Da die Daten nicht sortiert sind, muss der Index jeden einzelnen Datensatz (oder jede relevante Seite) adressieren.

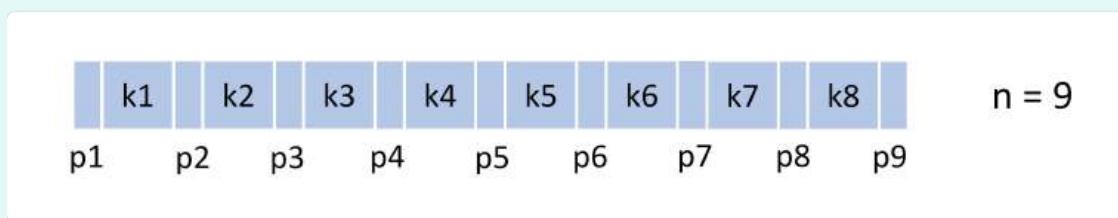
## Indexstrukturen - $B^+$ Bäume

### $\equiv B^+$ Baum Beispiel



- **Verzweigungsgrad/Ordnung (Branching Factor, Fanout)  $n$ :** Die maximale Anzahl von Kindknoten, die ein interner Knoten in einem B+-Baum haben kann. Dieser Parameter beeinflusst die Höhe des Baumes und somit die Anzahl der Zugriffe, die für eine Suche erforderlich sind. Ein höherer Verzweigungsgrad führt zu einem flacheren Baum.
- $\dashv$  = **unbenutzter Pointer:** Markiert eine Stelle in einem Knoten, an der kein Kindknoten oder kein Datensatz-Pointer gespeichert ist.
- **nu (not used):** Bezeichnet einen nicht benutzten Eintrag innerhalb eines Knotens. Dies kann auftreten, wenn Knoten nicht vollständig gefüllt sind, um Einfügungen zu erleichtern.
- **Pointer auf Blattebene zeigen auf Positionen in der Datei:** In B+-Bäumen enthalten die Blattknoten die eigentlichen Daten-Einträge (oder Pointer zu den Daten, im Fall von sekundären Indexen). Diese Pointer verweisen auf die physische Speicheradresse der Datensätze auf der Festplatte.

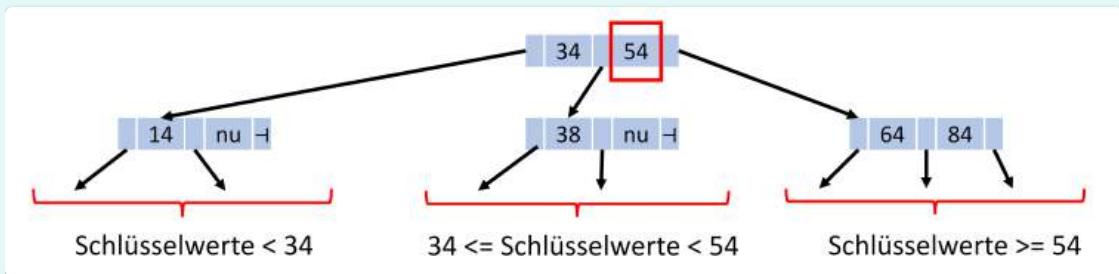
### B+-Baum Knotenstruktur



- **Wurzel, innere Knoten und Blätter haben die gleiche Struktur.**

- **Jeder Knoten hat  $n$  Pointer  $p_i$ .** Hier ist  $n = 9$ . Die Pointer in internen Knoten zeigen auf Kindknoten. In Blattknoten zeigen die Pointer auf Datensätze (oder Speicheradressen der Datensätze) oder, im Fall des letzten Pointers, auf den nächsten Blattknoten.
- **Jeder Knoten hat at most  $(n - 1)$  Search-Key-Werte  $k_j$ .** Hier also maximal  $9 - 1 = 8$  Schlüsselwerte. Die Schlüsselwerte in internen Knoten dienen als Wegweiser, um den richtigen Unterbaum für die Suche zu finden. In Blattknoten sind sie die tatsächlichen Suchschlüssel der gespeicherten Datensätze (oder eine Kopie davon).
- **Der letzte Pointer auf Blattebene zeigt auf den nächsten Blattknoten in Search-Key-Reihenfolge.** Dies ist ein wichtiges Merkmal von B+-Bäumen, das effiziente sequentielle Durchläufe (Bereichsabfragen) der Daten ermöglicht, ohne zum Wurzelknoten zurückkehren zu müssen. Die Blattknoten sind also einfach verkettet.

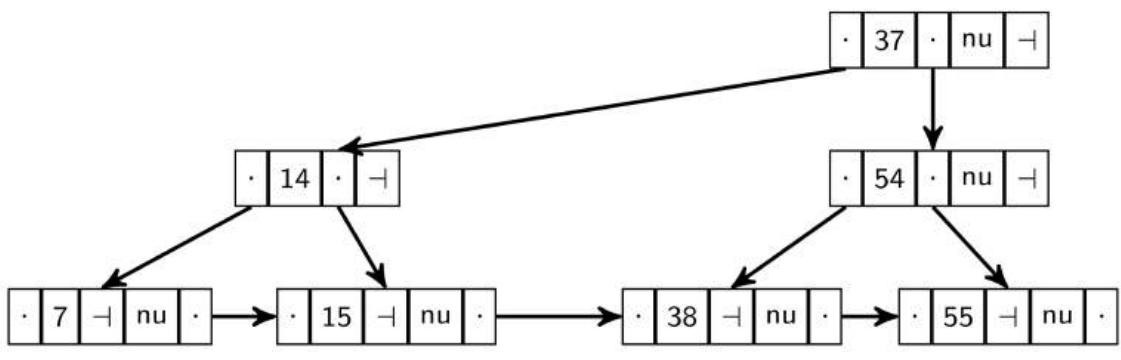
### B+-Baum Eigenschaften



- **Ordnung:**
  - Die Werte in jedem Knoten sind geordnet:  $k_i < k_j$ , wenn  $i < j$ . (Innerhalb eines Knotens sind die Schlüsselwerte sortiert.)
  - Teilbäume sind geordnet. (Alle Schlüsselwerte im linken Unterbaum eines Knotens sind kleiner als der kleinste Schlüsselwert im Knoten, und alle Schlüsselwerte im rechten Unterbaum sind größer oder gleich.)
- **Balanciert:** Alle Pfade von der Wurzel zu den Blattknoten haben die gleiche Länge. (Dies garantiert eine logarithmische Suchzeit, da alle Blätter die gleiche Tiefe haben.)
- **Verzweigung:** Jeder innere Knoten (nicht die Wurzel) hat zwischen  $\lceil \frac{n}{2} \rceil$  und  $n$  Kinder. ( $n$  ist der Verzweigungsgrad/Ordnung.)
  - Ausnahme: Der Wurzelknoten hat zwischen 2 und  $n$  Kinder (wenn er nicht selbst ein Blatt ist).
- **Blattknoten:** Haben zwischen  $\lceil \frac{n-1}{2} \rceil$  und  $n - 1$  Pointer auf Tupel (oder deren Speicheradressen) in der Datei und 1 Pointer auf den nächsten Blattknoten (für effiziente Bereichsabfragen).

### Ein minimaler B+-Baum

Ein minimal gefüllter B+-Baum für  $n = 3$ :



- Für  $n = 3$  bedeutet das:
  - Innere Knoten (nicht Wurzel) müssen mindestens  $\lceil \frac{3}{2} \rceil = 2$  Kinder haben.
  - Blattknoten müssen mindestens  $\lceil \frac{3-1}{2} \rceil = 1$  Schlüsselwert haben.
  - Die Wurzel kann weniger als 2 Kinder haben, wenn sie ein Blatt ist oder die einzige Wurzel.

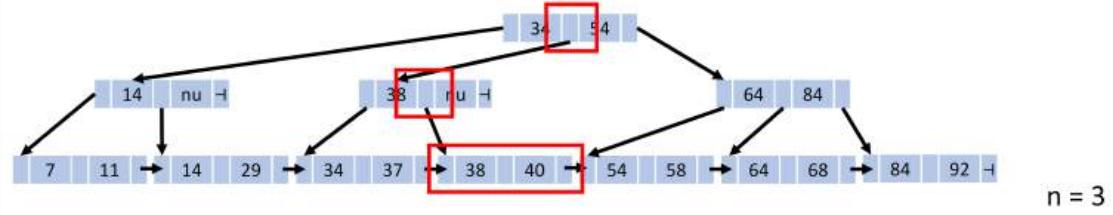
Der gezeigte Baum illustriert diese Minimalbedingungen. Beachte die Verkettung der Blattknoten für sequenzielle Zugriffe.

## Verwendung von B<sup>+</sup>-Bäumen in DBMS

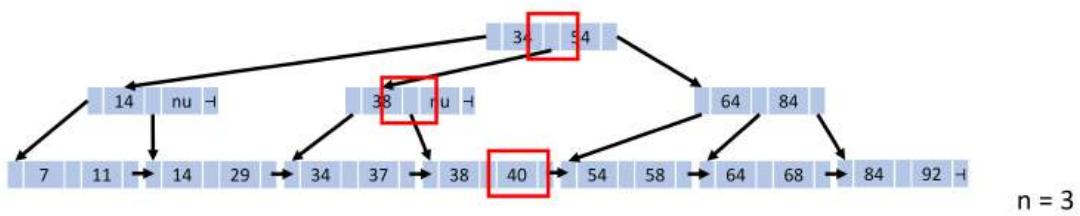
- **Knotengröße:** Jeder Knoten hat die Größe eines I/O-Blocks (optimiert für den Datentransfer zwischen Speicher und Festplatte).
- **Füllgrad:** Ein Knoten ist zu mindestens 50% gefüllt (gewährleistet eine gewisse Speichereffizienz).
- **Baumstruktur:** Ein B<sup>+</sup>-Baum ist in der Regel sehr flach, d.h. die Suche erfordert nur wenige wahlfreie Zugriffe (schnellere Suchzeiten).
- **Caching:** Die ersten 1-2 Ebenen des Baums sind in der Regel im Hauptspeicher "gecached" (ermöglicht sehr schnelle Zugriffe auf häufig benötigte Indexinformationen).
- **Logische vs. physikalische Nähe:** "Logisch" nahe bedeutet nicht unbedingt "physisch" nahe (die physische Anordnung auf der Festplatte kann fragmentiert sein). Das Lesen eines Knotens erfordert typischerweise einen I/O-Zugriff.
- **Innere Knoten:** Innere Knoten entsprechen einer Hierarchie von *sparse indexes* (enthalten nur wenige, aber repräsentative Schlüssel, um den Suchraum einzuschränken).

**Uniqueness-Constraints:** Uniqueness-Constraints auf Attributen in einer Datenbank werden durch B<sup>+</sup>-Bäume realisiert → **Primärschlüssel** (B<sup>+</sup>-Bäume eignen sich gut zur Durchsetzung von Eindeutigkeitsbedingungen und für effiziente Bereichsabfragen).

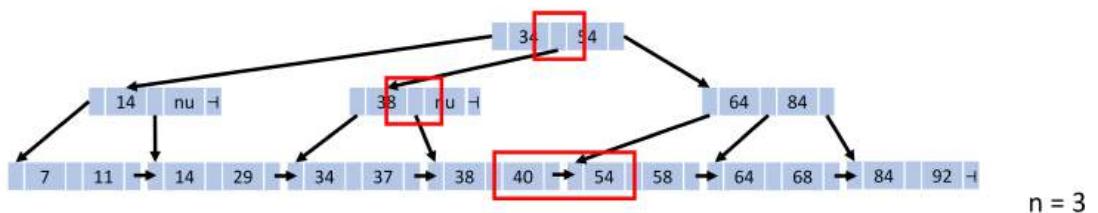
### ☰ Beispiel zum Lookup



- Suche nach Wert: 43 → nicht vorhanden

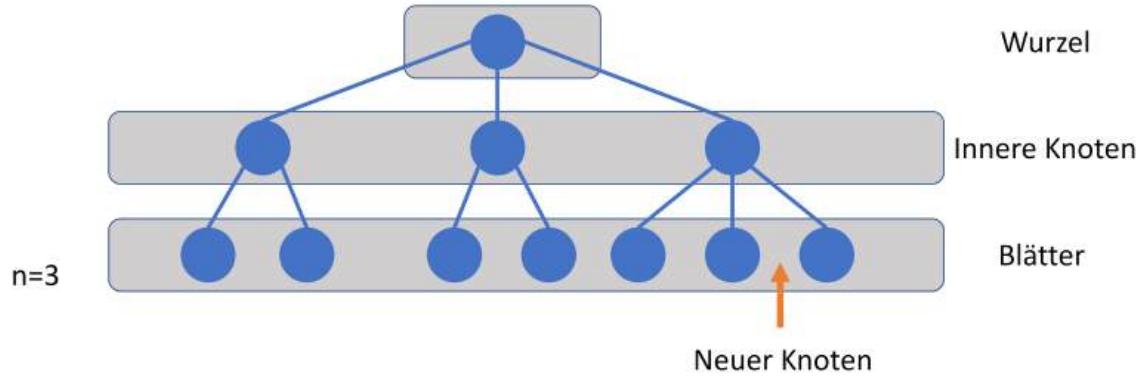


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden

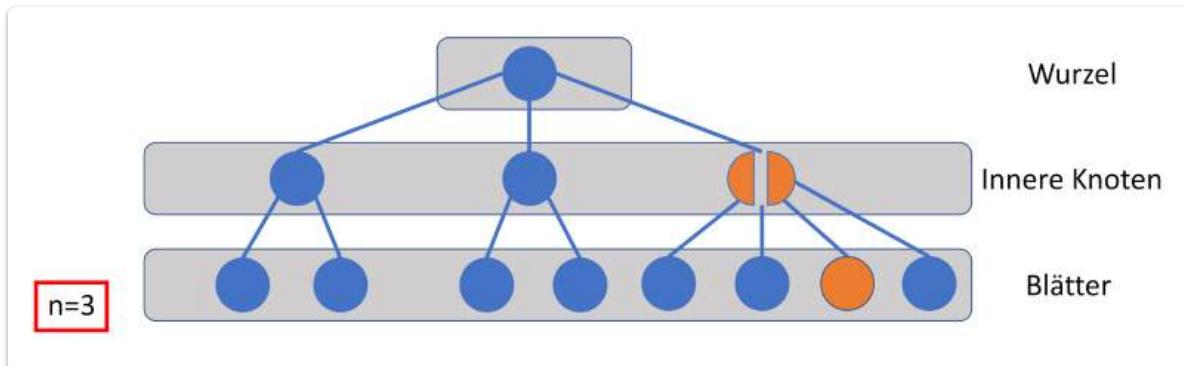


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden
- Suche nach Bereich: 40-55 → gefunden

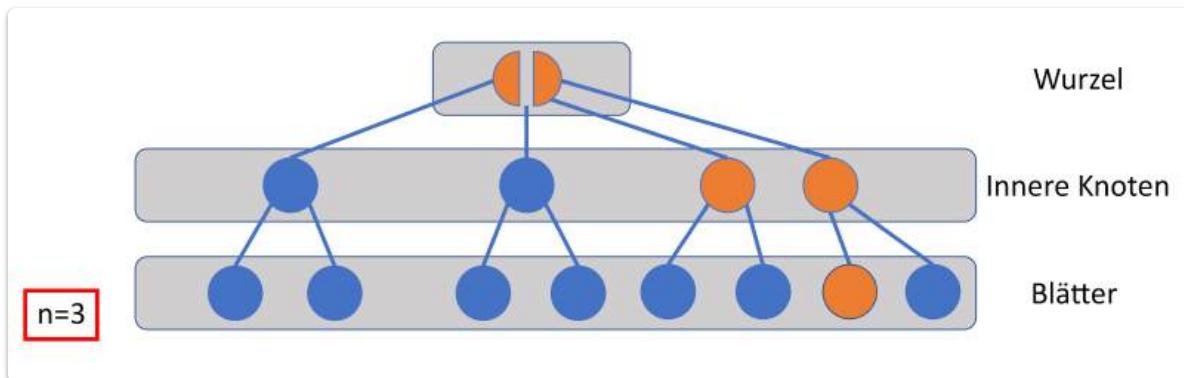
## Einfügen beim $B^+$ Baum



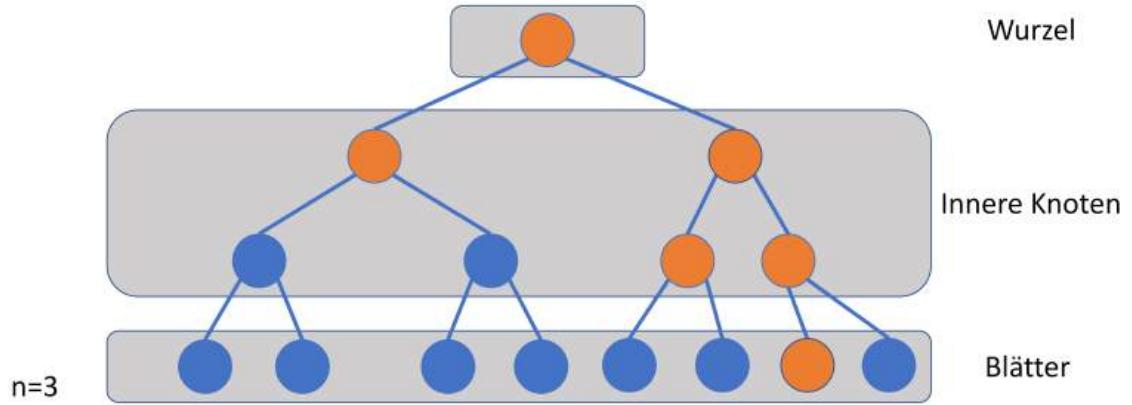
Wir wollen hier den neuen Knoten einfügen



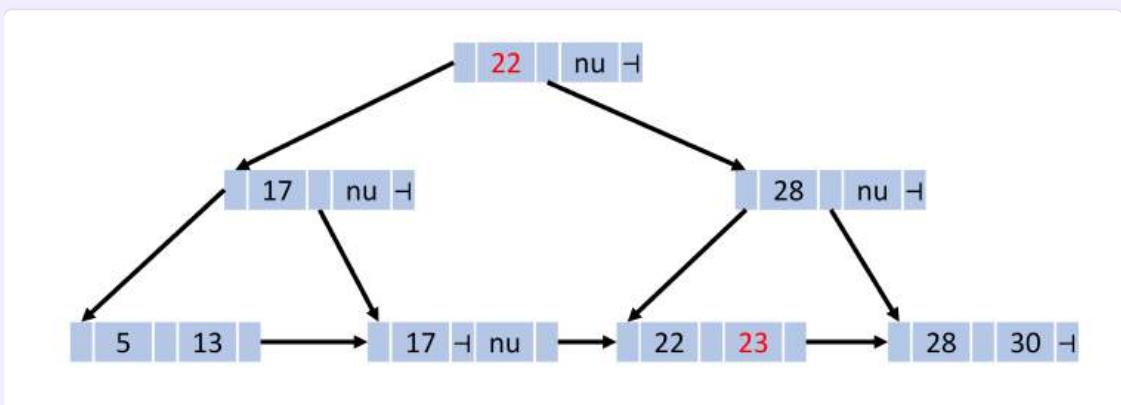
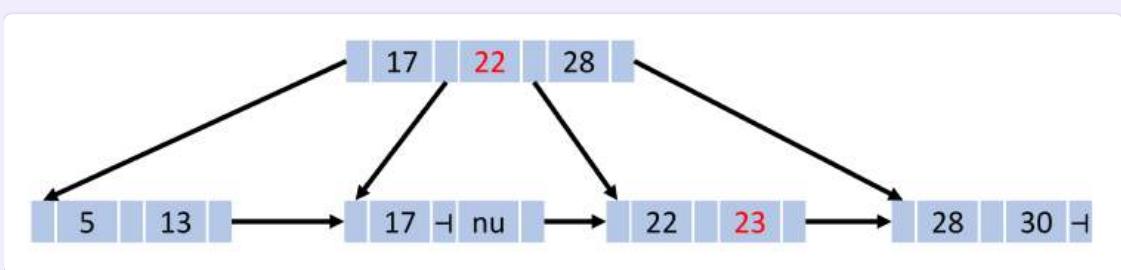
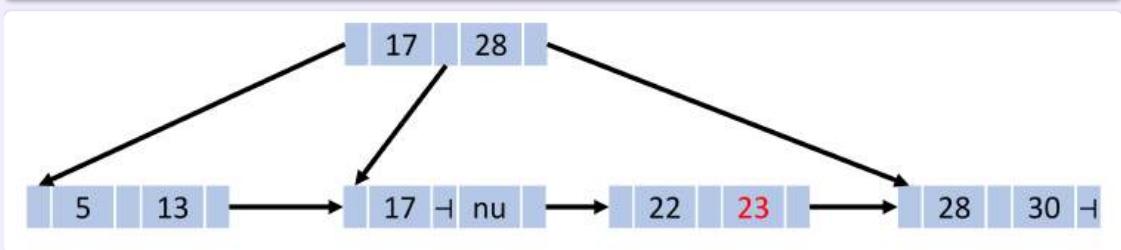
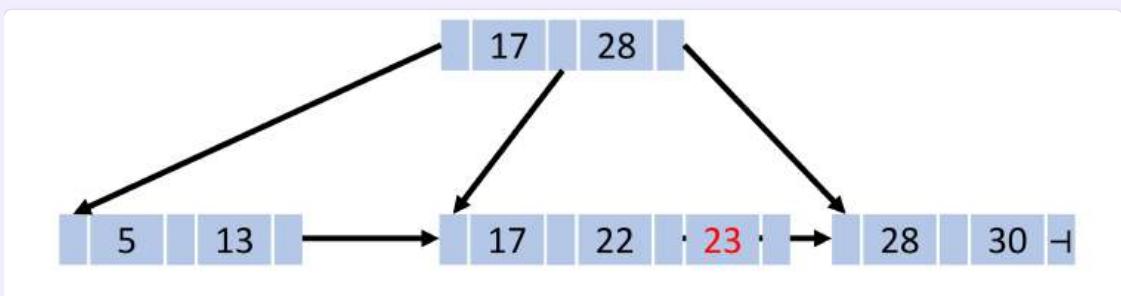
Da unser  $n = 3$  müssen wir eine Ebene darüber in 2 teilen



Dann haben wir das selbe Problem in einer Ebene darüber. Da müssen wir auch ein mal spalten und haben am Ende eine Ebene mehr



### ☰ Beispiel: Einfügen von 23



Mehr Beispiele: DBS-8\_Physischer Datenbankentwurf, p.32

## B<sup>+</sup>-Baum Löschen

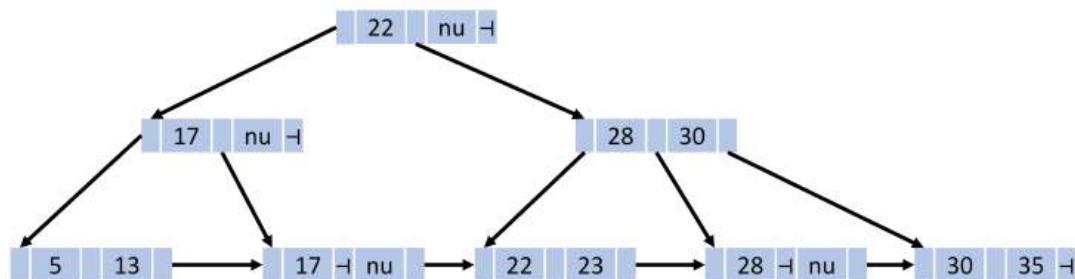
### Löschen von Schlüsselwert k:

- **Suche:** Finde das Blatt  $b$ , das den Schlüsselwert  $k$  enthält.
- **Löschen (bei ausreichendem Füllgrad):**
  - Falls  $b$  genügend gefüllt bleibt (mindestens  $\lceil \frac{n}{2} \rceil$  Einträge, wobei  $n$  die maximale Anzahl an Einträgen pro Knoten ist), lösche  $k$ .
  - **Wichtig:** Innere Knoten können dann Schlüssel enthalten, die nicht mehr in den Blättern existieren (diese dienen weiterhin als Wegweiser).
- **Verschmelzen oder Umverteilen (bei Unterschreitung des minimalen Füllgrads):**
  - **Verschmelzen:**
    - Falls Verschmelzen mit einem Nachbarknoten möglich ist (der Nachbarknoten hat nicht den minimalen Füllgrad überschritten, sodass nach dem verschmelzen beider Knoten der minimale Füllgrad nicht unterschritten wird), verschmelze die Knoten und passe den Pointer im Elternknoten an.
  - **Umverteilung:**
    - Falls Verschmelzen nicht möglich ist (kein geeigneter Nachbarknoten vorhanden), versuche eine Neuverteilung der Schlüssel und Pointer über den Elternknoten (ein Schlüssel wird vom Nachbarn "ausgeliehen").
- **Kaskadierendes Verschmelzen:** Verschmelzen muss unter Umständen auch in höheren Ebenen des Baumes erfolgen, wenn durch das Verschmelzen in einer unteren Ebene der minimale Füllgrad im Elternknoten unterschritten wird.
- **Baumtiefe:** Die Tiefe des Baumes kann sich um eine Ebene verringern, wenn die Wurzel nach einer Verschmelzungsoperation nur noch einen oder keinen Kindknoten hat.

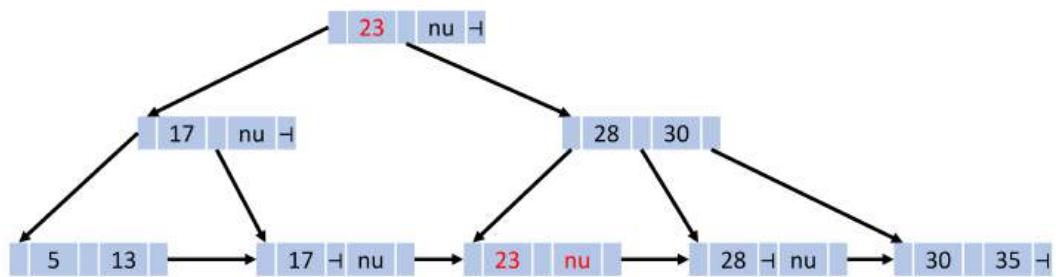
### :≡ Beispiele

#### Löschen von 22

n=3, Löschen von 22

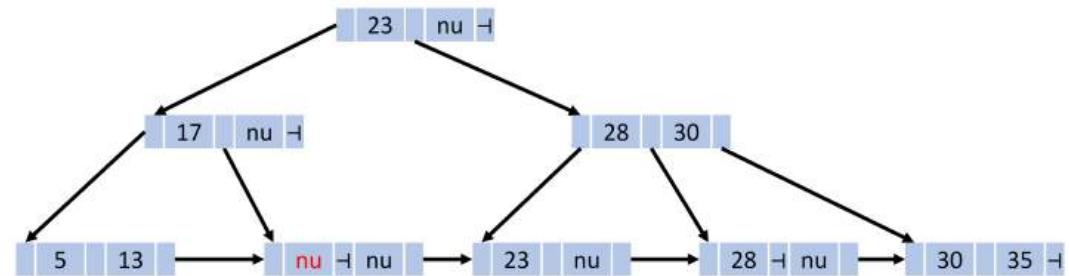


n=3, Löschen von 22

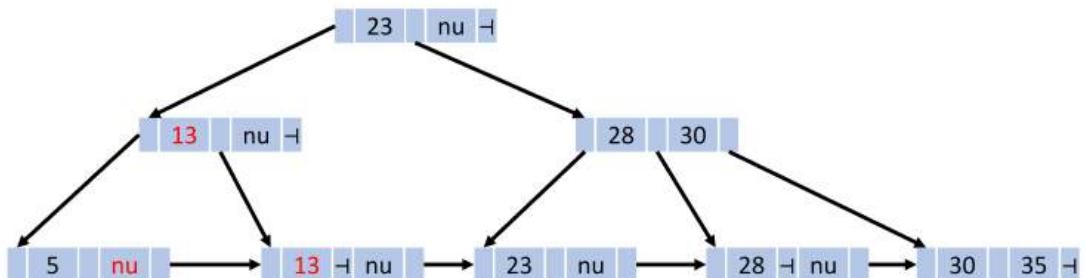


Löschen von 17

n=3, Löschen von 17

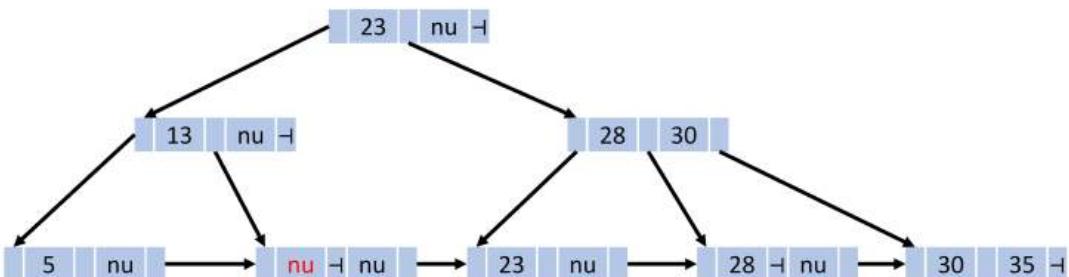


n=3, Löschen von 17

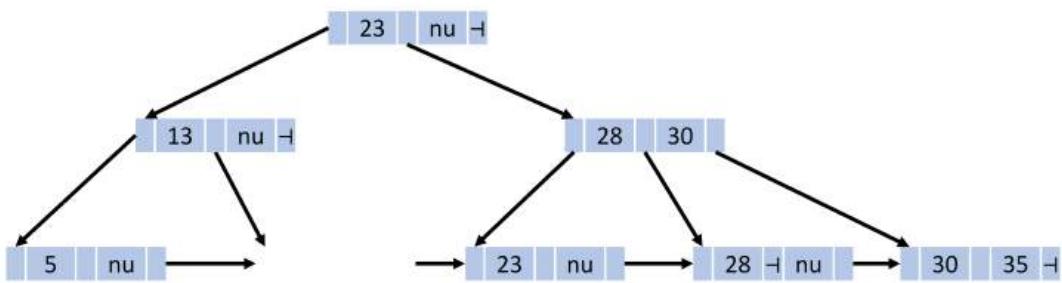


Löschen von 13

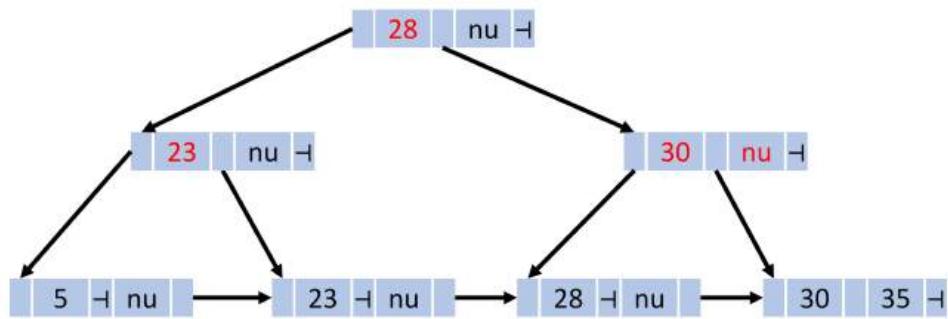
n=3, Löschen von 13



n=3, Löschen von 13

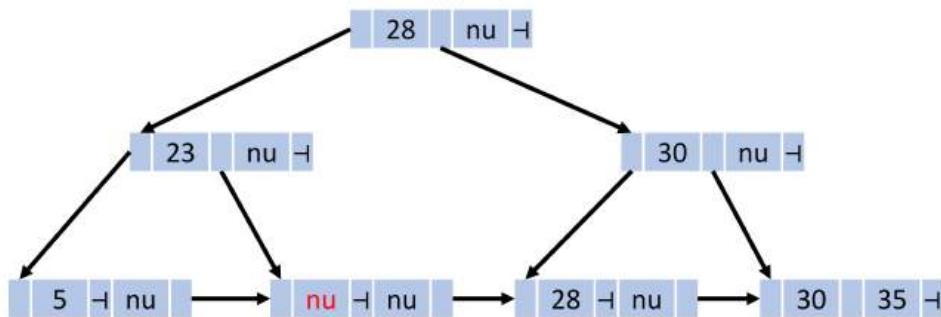


n=3, Löschen von 13

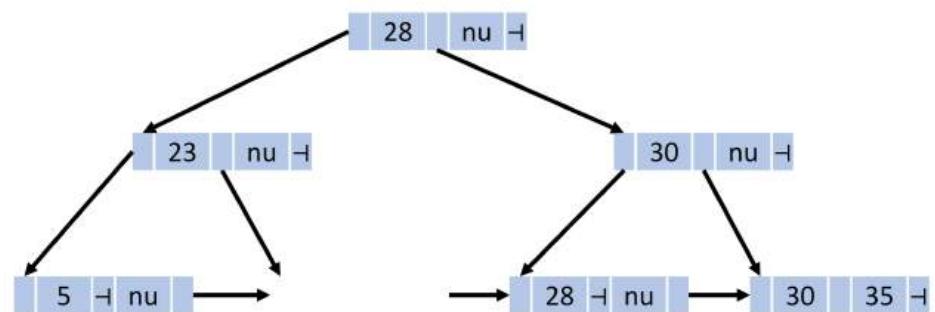


Löschen von 23

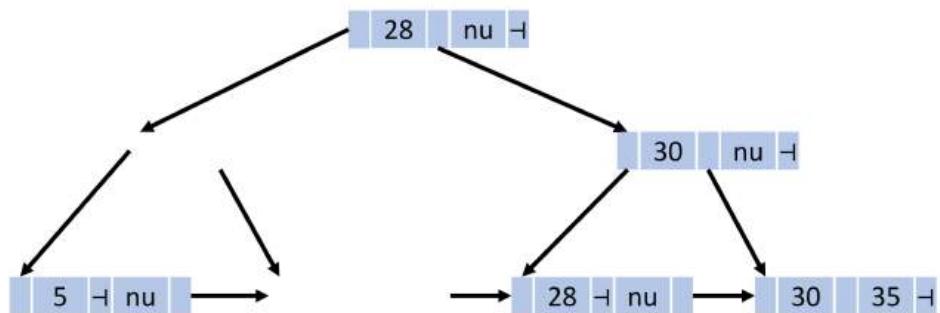
n=3, Löschen von 23



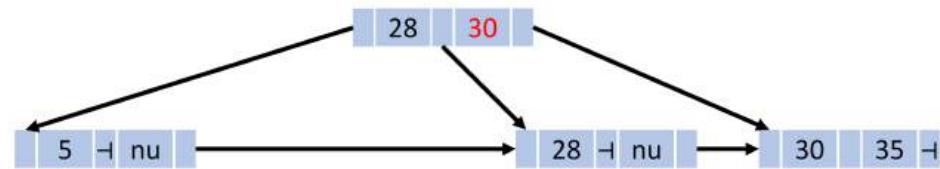
n=3, Löschen von 23



n=3, Löschen von 23



n=3, Löschen von 23

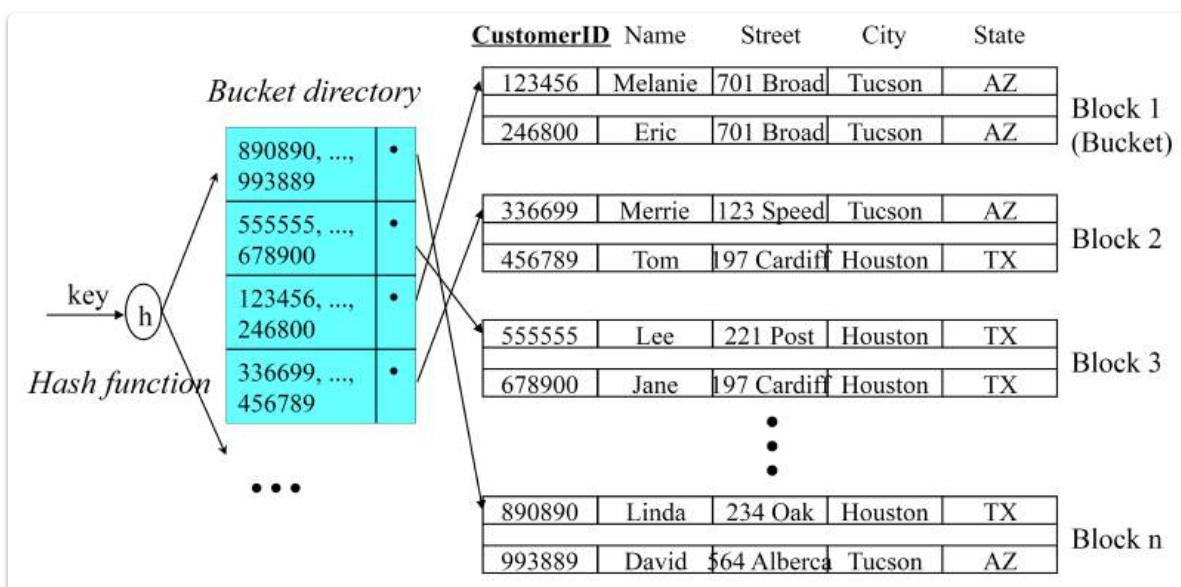


## Indexstrukturen - Hashing

### Statischer Hash-Index

Erstellen eines Indexes auf Basis einer Hashfunktion anstatt auf Basis eines Search-Keys.

- **Hashfunktion:**
  - Wahl einer geeigneten Hashfunktion  $h$ .
  - Hashfunktion  $h$  auf Search-Key-Wert  $k$  anwenden:  $h(k)$ .
  - Reserviere ein **Bucket** (Block/Seite) für jeden Wert von  $h(k)$ .



(Die Abbildung zeigt eine Tabelle mit Kundendaten, eine Hashfunktion  $h$ , ein Bucket

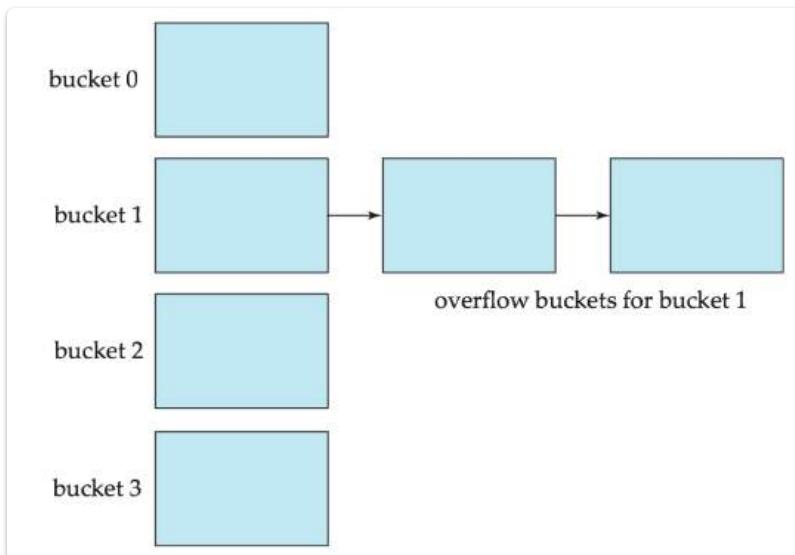
Directory, das Hashes auf Buckets abbildet, und die eigentlichen Datenblöcke (Buckets) mit den Datensätzen.)

### Suche:

- Ein Zugriff auf das **Bucket Directory** (um das zugehörige Bucket zu finden).
- Ein Zugriff auf die **Datei** (genauer: auf den Datenblock/das Bucket), um die gesuchten Datensätze zu finden.

### Gute Performance hängt von einer guten Hashfunktion ab:

- Bucket kann überfüllt sein.
- Zu viele Tupel werden auf das gleiche Bucket abgebildet  $\implies$  **Kollision**.
- **Lösung:**
  - **Overflow-Buckets:** Zusätzliche Buckets, die verkettet werden, um Überläufe aufzufangen.
  - **Overflow-Chains:** Verkettung von Blöcken innerhalb eines Buckets oder von Overflow-Buckets.



## Statischer Hash-Index

- **Open vs. Closed Hashing:**
  - **Open Hashing:**
    - Overflow Chains mit weiteren Overflow Buckets (jeder Bucket kann eine verkettete Liste von Überlauf-Buckets haben).
  - **Closed Hashing:**
    - Feste Anzahl von Buckets.
    - Beim Überlaufen muss eines der existierenden Buckets verwendet werden (z.B. durch Linear Probing, weitere Hashfunktionen).

## Static Hashing

**Problem bei Static Hashing:** Hashfunktion und Anzahl der Buckets muss bereits beim Erstellen bestimmt werden.

**Datenbanken wachsen und schrumpfen mit der Zeit!**

- **Initiale Bucketanzahl zu gering:**  
⇒ Viele Overflows, Performance leidet.
- **Initiale Bucketanzahl zu groß:**  
⇒ Underflow, Speicherplatz wird verschwendet.

**Lösungsansätze:**

- **Periodische Reorganisation:** Anpassen der Hashfunktion und der Bucketanzahl in regelmäßigen Abständen (kann ressourcenintensiv sein und zu Ausfallzeiten führen).
- **Dynamic Hashing:** Ermöglicht Modifikationen zu einem späteren Zeitpunkt (z.B. durch Erweitern oder Verkleinern der Bucketanzahl on-the-fly).

## Design Tuning

---

**Optimierungsgegenstände:**

- Clustering Index vs. Hashing
- Sparse vs. dense Index
- Clustering vs. non-clustering Index
- Beschleunigung von Joins durch Indexe

**Grundsätzliche Fragestellungen:**

- Sind die Kosten für eine periodische Reorganisation akzeptabel?
- Wie viele Updates gibt es wirklich?
- Optimierungsziel: durchschnittl. Laufzeit oder Worst-Case-Optimierung?
- Welche Arten von Anfragen werden erwartet?

## Nutzung von Indexen

**Manchmal werden existierende Indexe nicht verwendet:**

- **System-Katalog hat veraltete Informationen** (der Query-Optimierer trifft Entscheidungen basierend auf möglicherweise falschen Statistiken).
- **Optimierer könnte annehmen, dass die Tabelle klein ist** (bei kleinen Tabellen kann ein Full Table Scan effizienter sein als die Indexnutzung).
- **"Gute" und "schlechte" Typen von Anfragen (bezüglich Indexnutzung):**
  - `SELECT * FROM EMP WHERE salary/12 > 4000` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).

- `SELECT * FROM EMP WHERE salary > 48000` (Bereichsabfragen können Indexe gut nutzen).
- `SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G'` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).
- `SELECT * FROM EMP WHERE name LIKE 'G%'` (führende Wildcards können Indexnutzung einschränken, nachfolgende Wildcards oft nicht).
- `SELECT * FROM EMP WHERE name = 'Smith'` (Punktuelle Suchen auf indizierten Spalten sind ideal für Indexe).
- `SELECT * FROM EMP WHERE salary IS NULL` (Indexe können oft auch für IS NULL-Bedingungen genutzt werden).
- **Geschachtelte SQL-Anfragen** (die Effizienz der Indexnutzung kann von der Struktur der geschachtelten Anfrage abhängen).
- **Negationen** (`NOT`, `!=`) (können die Indexnutzung erschweren).
- **Anfragen mit OR** (die Indexnutzung bei OR-Bedingungen hängt von der Verfügbarkeit von Indexen auf den beteiligten Spalten ab und wie der Optimierer die Anfrage umformuliert).

# 7. Anfrageoptimierung

## Ausführen einer SQL-Anfrage

### Reihenfolge der Klauseln

Die Klauseln einer SQL-Anfrage werden in folgender Reihenfolge *angegeben*:

- `SELECT column(s)`
- `FROM table list`
- `WHERE condition`
- `GROUP BY grouping column(s)`
- `HAVING group condition`
- `ORDER BY sort list`

### Ausführungsreihenfolge einer SQL-Anfrage

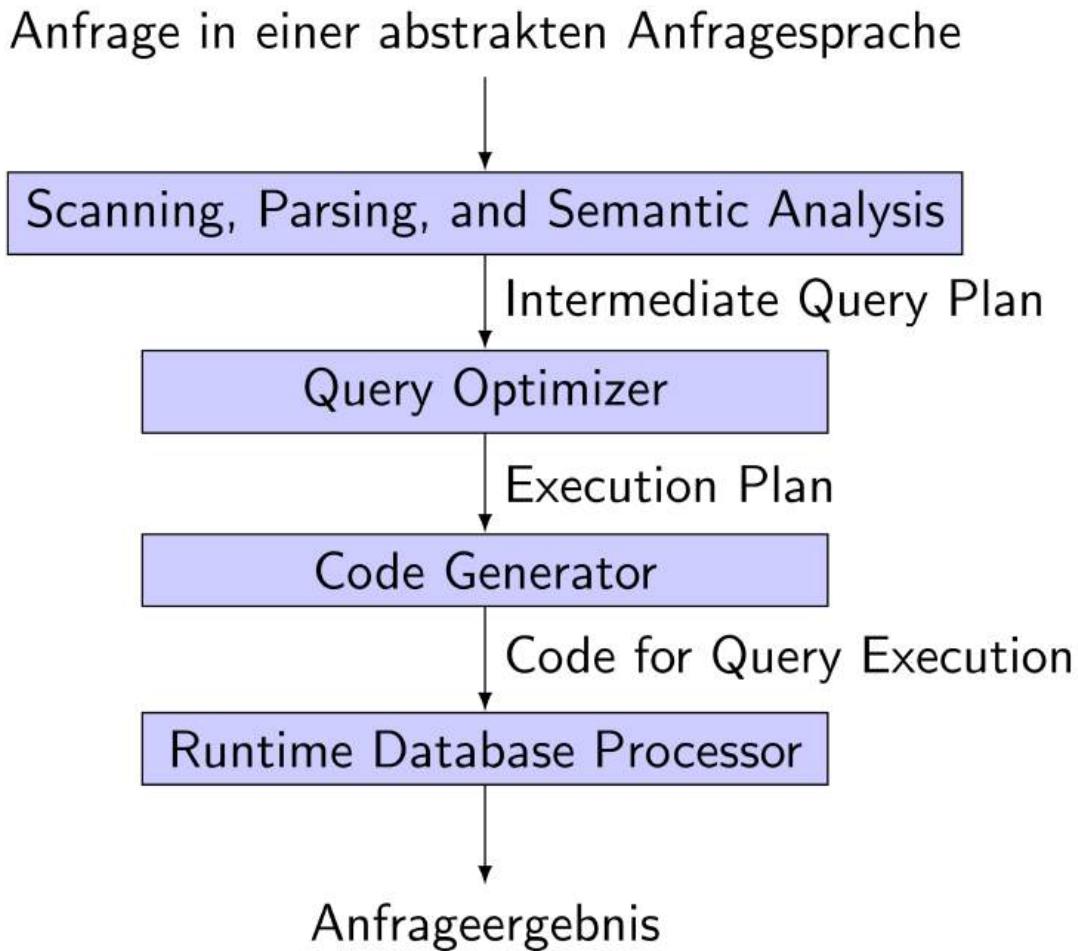
Die Anfrage wird jedoch in einer *anderen Reihenfolge* ausgeführt, als sie angegeben wird:

1. **Kartesisches Produkt** der Tabellen in der `FROM`-Klausel.
  - *Erklärung:* Es werden alle möglichen Kombinationen von Zeilen aus den angegebenen Tabellen gebildet. Wenn z.B. Tabelle A 3 Zeilen und Tabelle B 4 Zeilen hat, entstehen 12 Zeilen im kartesischen Produkt.
2. Anwendung der **Prädikate** in der `WHERE`-Klausel.
  - *Erklärung:* Hier werden Zeilen herausgefiltert, die die angegebene Bedingung nicht erfüllen.
3. Anwendung der `GROUP-BY`-Klausel.
  - *Erklärung:* Die verbleibenden Zeilen werden zu Gruppen zusammengefasst, basierend auf den Werten in den `grouping column(s)`.
4. Anwendung der **Prädikate** in der `HAVING`-Klausel (um Gruppen zu eliminieren).
  - *Erklärung:* Dies filtert Gruppen basierend auf einer Bedingung, die sich oft auf Aggregatfunktionen bezieht.
5. Berechnung der **Aggregatfunktionen** für jede verbleibende Gruppe.
  - *Erklärung:* Funktionen wie `SUM()`, `AVG()`, `COUNT()`, `MAX()`, `MIN()` werden auf die jeweiligen Gruppen angewendet.
6. **Projektion** auf Spalten der `SELECT`-Klausel.
  - *Erklärung:* Es werden nur die Spalten ausgewählt und angezeigt, die in der `SELECT`-Klausel spezifiziert sind.

- SQL ist eine **deklarative Sprache**. (mehr: [4. SQL](#))
  - Das bedeutet, man beschreibt *was* man möchte (das Ergebnis), aber nicht *wie* das Datenbankmanagementsystem (DBMS) es erreichen soll.
  - Das DBMS ist dafür verantwortlich, den effizientesten Weg zur Ausführung der Anfrage zu finden.

## Schritte der Anfragebearbeitung

Die Verarbeitung einer Anfrage in einem Datenbanksystem durchläuft mehrere Schritte:



Query Optimizer optimiert den Code für bessere Laufzeit

## Anfrageoptimierung

### Alternativen bei der Anfrageoptimierung

Der Anfrageoptimierer hat verschiedene Möglichkeiten, um einen optimalen Ausführungsplan zu finden:

- **Äquivalente Ausführungspläne:** Für dieselbe Anfrage gibt es oft mehrere Ausführungspläne, die dasselbe Ergebnis liefern. Der Optimierer wählt den kostengünstigsten.
- **Algorithmen zur Ausführung von Algebraoperatoren:** Für relationale Algebraoperatoren (z.B. Join, Selektion, Projektion) gibt es verschiedene Implementierungsalgorithmen (z.B. Nested-Loop Join, Hash Join, Sort-Merge Join). Der Optimierer wählt den passenden Algorithmus.
- **Methoden, um auf Relationen zuzugreifen (Indizes):** Es gibt verschiedene Möglichkeiten, auf Daten in Tabellen zuzugreifen. Ob ein Index verwendet wird oder ein vollständiger Tabellen-Scan, hängt von den Kosten ab.

*Bei gleichem Ergebnis können Ausführungskosten sehr unterschiedlich sein.*

## Theorie vs. Realität

- Es ist nicht die Aufgabe des Benutzers, "effiziente" Anfragen zu schreiben, sondern die Aufgabe der Anfrageoptimierung, effiziente Ausführungspläne zu finden!
- Aber in der Realität... Optimierer sind nicht perfekt.
  - Sie basieren auf Heuristiken und Statistiken, die nicht immer 100% genau sind.
  - In komplexen Szenarien kann es vorkommen, dass der Optimierer nicht den absolut besten Plan findet.

## Anfrageausführungskosten

### Kostenmodell

Die **Gesamte Zeit bis das Anfrageergebnis vorliegt** wird als **Response Time** (Antwortzeit) bezeichnet. Viele Faktoren tragen zu dieser bei:

- **Festplattenzugriff (I/O-Kosten):**
  - Dies ist oft der **dominierende Faktor** bei den Kosten.
  - Daten müssen von der Festplatte in den Hauptspeicher geladen werden.
  - **Block Access Time:**
    - **Seek Time:** Die Zeit, die der Schreib-/Lesekopf benötigt, um zu der richtigen Spur auf der Festplatte zu gelangen.
    - **Rotation Time:** Die Zeit, die der benötigte Sektor benötigt, um unter dem Schreib-/Lesekopf vorbeizudrehen.
- **CPU-Kosten:**
  - Kosten für die Verarbeitung von Daten im Hauptspeicher (z.B. Sortieren, Hashen, Vergleichen).
- **Netzwerkkommunikation:**

- Wenn Daten über ein Netzwerk abgerufen oder Ergebnisse an einen Client gesendet werden müssen.
- **Aktueller Query-Load:**
  - Die Anzahl und Art der gleichzeitig laufenden Anfragen kann die Performance beeinflussen (Konkurrenz um Ressourcen).
- **Parallelisierung:**
  - Wenn Operationen parallel ausgeführt werden können, kann dies die Antwortzeit verkürzen. Kosten für die Koordination.

## Logische Anfrageoptimierung

Die logische Anfrageoptimierung befasst sich mit der Umformung des Anfrageplans auf einer höheren, **abstrakteren Ebene**, bevor physikalische Details berücksichtigt werden.

- **Relationale Algebra:**
  - Der initiale Anfrageplan wird oft in Operatoren der relationalen Algebra dargestellt.
  - Die logische Optimierung manipuliert diesen Operatorbaum.
- **Äquivalenzhaltende Transformationsregeln:**
  - Dies sind Regeln, die es erlauben, einen relationalen Algebraausdruck in einen anderen umzuwandeln, ohne das Ergebnis der Anfrage zu verändern.
  - **Beispiel:** Das Vorziehen von Selektionen (Filtern) vor Joins ist oft effizienter, da die Datenmenge vor dem teuren Join reduziert wird.
- **Heuristische Optimierung:**
  - Dies sind "Faustregeln" oder Daumenregeln, die auf Erfahrungen basieren, um einen besseren Anfrageplan zu finden, ohne alle möglichen Pläne exakt zu kosten.
  - **Beispiel:** "Filter früh anwenden" ist eine typische Heuristik.

## Physische Anfrageoptimierung

Die physische Anfrageoptimierung befasst sich mit der Auswahl der **konkreten Implementierungsstrategien** für die Operatoren des Anfrageplans und der Berücksichtigung der tatsächlichen Kosten.

- **Algorithmen und Operatorimplementationen:**
  - Hier werden die spezifischen Algorithmen für die relationalen Algebraoperatoren ausgewählt (z.B. welcher Join-Algorithmus verwendet wird: Nested-Loop Join, Hash Join, Sort-Merge Join).
  - Auch die Zugriffspfade auf die Daten werden festgelegt (z.B. Tabellen-Scan oder Index-Scan).
- **Kostenmodell:**
  - Ein Kostenmodell wird verwendet, um die geschätzten Ausführungskosten für verschiedene physische Pläne zu berechnen.

- Dabei werden Faktoren wie CPU-Zeit, E/A-Operationen (Festplattenzugriffe) und Netzwerkkommunikation berücksichtigt.
- Der Plan mit den geringsten geschätzten Kosten wird ausgewählt.

## Logische (heuristische) Anfrageoptimierung

---

### Logische Anfrageoptimierung

#### Grundlagen der Logischen Anfrageoptimierung

- **Grundlage: Äquivalenzerhaltende Transformationsregeln**
  - Diese Regeln sind zentral für die logische Optimierung. Sie erlauben es, einen Ausdruck der relationalen Algebra in einen anderen umzuformen, ohne das Ergebnis der Anfrage zu verändern.
  - Sie bilden den **Suchraum** der möglichen Anfragepläne.
- **Algebraische Transformationen** bilden den Suchraum.
- **Gegeben sei ein initialer algebraischer Ausdruck:**
  - Verwende äquivalenzerhaltende Transformationsregeln, um neue Ausdrücke abzuleiten.
  - Der Optimierer erkundet diesen Suchraum, um alternative Pläne zu finden.

#### Was ist ein guter Plan?

- Eine **genaue Entscheidung ohne Kostenfunktion ist nicht möglich**. Die logische Optimierung trifft noch keine konkrete Kostenabschätzung.
- Logische Anfrageoptimierung basiert auf **Heuristiken**.
  - Diese "Faustregeln" helfen, den Suchraum einzuschränken und vielversprechende Pläne zu identifizieren.

#### Hauptziel der logischen Anfrageoptimierung

- **Größe von Zwischenergebnissen reduzieren!**
  - Dies ist entscheidend, da kleinere Zwischenergebnisse weniger E/A-Operationen und CPU-Ressourcen benötigen und somit die Gesamtausführungszeit erheblich reduzieren.

### Äquivalenzerhaltende Transformationsregeln

Diese Regeln ermöglichen es, relationale Algebraausdrücke umzuformen und dabei ihre Semantik (das Ergebnis) zu bewahren.

## Aufbrechen von Konjunktionen in Selektionsprädikaten

$$\sigma_{c_1 \wedge c_2 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

$\sigma$  ist kommutativ

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

### $\pi$ -Kaskaden

If  $L_1 \subseteq L_2 \subseteq \dots \subseteq L_n$  dann gilt

$$\pi_{L_1}(\pi_{L_2}(\dots(\pi_{L_n}(R))\dots)) \equiv \pi_{L_1}(R)$$

### Vertauschen der Reihenfolge von $\sigma$ und $\pi$

Falls die Selektion sich nur auf Attribute  $A_1, \dots, A_n$  der Projektionsliste bezieht, können die beiden Operationen vertauscht werden:

$$\pi_{A_1, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, \dots, A_n}(R))$$

$\cup, \cap$  und  $\bowtie$  sind kommutativ

$$R \bowtie_c S \equiv S \bowtie_c R$$

### Vertauschen von $\sigma$ und $\bowtie$

Falls das Selektionsprädikat  $c$  nur auf Attribute der Relation  $R$  zugreift, kann man die beiden Operationen vertauschen:

$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j S$$

Falls das Selektionsprädikat  $c$  eine Konjunktion der Form  $c_1 \wedge c_2$  ist und  $c_1$  sich nur auf Attribute aus  $R$  und  $c_2$  sich nur auf Attribute aus  $S$  bezieht, gilt folgende Äquivalenz:

$$\sigma_c(R \bowtie_j S) \equiv \sigma_{c_1}(R) \bowtie_j \sigma_{c_2}(S)$$

### Vertauschen von $\pi$ und $\bowtie$

Gegeben sei Projektionsliste  $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$ , wobei  $A_i$  Attribute aus  $R$  und  $B_i$  Attribute aus  $S$  sind. Falls sich das Joinprädikat  $c$  nur auf Attribute aus  $L$  bezieht, gilt folgende Umformung:

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n}(R)) \bowtie_c (\pi_{B_1, \dots, B_m}(S))$$

### $\bowtie, \cap, \cup$ sind (jeweils einzeln betrachtet) assoziativ

Wenn also  $\Phi$  eine dieser Operationen bezeichnet, so gilt:

$$(R \Phi S) \Phi T \equiv R \Phi (S \Phi T)$$

### $\sigma$ ist distributiv mit $\cap, \cup, -$

Wenn also  $\Phi$  eine dieser Operationen bezeichnet, so gilt:

$$\sigma_c(R \Phi S) \equiv (\sigma_c(R)) \Phi (\sigma_c(S))$$

### $\pi$ ist distributiv mit $\cup$

$$\pi_c(R \cup S) \equiv (\pi_c(R)) \cup (\pi_c(S))$$

Join und/oder Selektionsprädikate können mittels de Morgan's Regeln umgeformt werden

$$\neg(c_1 \wedge c_2) \equiv (\neg c_1) \vee (\neg c_2)$$

$$\neg(c_1 \vee c_2) \equiv (\neg c_1) \wedge (\neg c_2)$$

### Kombination von Kartesischem Produkt und Selektion

Ein kartesisches Produkt, das von einer Selektionsoperation gefolgt wird, deren Selektionsprädikat Attribute aus beiden Operanden des kartesischen Produktes enthält, kann in eine Joinoperation umgeformt werden.

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

Ebenfalls relevant: die alternativen Ausdrücke für Operatoren der relationalen Algebra.

## Phasen der logischen Anfrageoptimierung

Die logische Anfrageoptimierung durchläuft typischerweise mehrere Phasen, um einen effizienten Anfrageplan zu generieren. Das Hauptziel ist dabei immer, die Größe der Zwischenergebnisse so früh wie möglich zu reduzieren.

### 1. Aufbrechen von Selektionen:

- Konjunktive Selektionsprädikate (Bedingungen, die mit AND verbunden sind) werden in einzelne Selektionsoperationen aufgeteilt.
- *Beispiel:* WHERE A > 10 AND B = 'x' wird zu zwei einzelnen Selektionen.

### 2. Verschieben der Selektionen so weit wie möglich nach unten (pushing selections):

- Dies ist eine der wichtigsten Heuristiken. Selektionen (Filteroperationen) sollten so früh wie möglich im Ausführungsplan angewendet werden.
- Je früher Daten gefiltert werden, desto kleiner werden die Zwischenergebnisse, was die Kosten für nachfolgende Operationen (wie Joins) erheblich reduziert.
- *Beispiel:* SELECT \* FROM R JOIN S ON R.id = S.id WHERE R.value > 10 ist effizienter, wenn R.value > 10 vor dem Join auf Tabelle R angewendet wird.

### 3. Joins einführen (Zusammenfassen von Selektionen und Kreuzprodukten):

- Kreuzprodukte (CROSS JOIN) gefolgt von Selektionen mit Gleichheitsbedingungen (WHERE R.id = S.id) werden in explizite Join-Operationen umgewandelt.
- Dies macht den Plan übersichtlicher und ermöglicht es dem Optimierer, spezifische Join-Algorithmen zu berücksichtigen.

### 4. Join-Reihenfolge bestimmen, so dass möglichst kleine Zwischenergebnisse entstehen:

- Die Reihenfolge, in der Joins ausgeführt werden, hat einen enormen Einfluss auf die Größe der Zwischenergebnisse und damit auf die Leistung der Anfrage.
- **Heuristik:** Joins mit Input von Selektionen vor anderen Joins auswerten.
  - Das bedeutet: Tabellen, die bereits durch Selektionen stark reduziert wurden, sollten bevorzugt gejoint werden, um die Datenmenge für die nachfolgenden Joins klein zu halten.

### 5. ggf. Einführen von Projektionen:

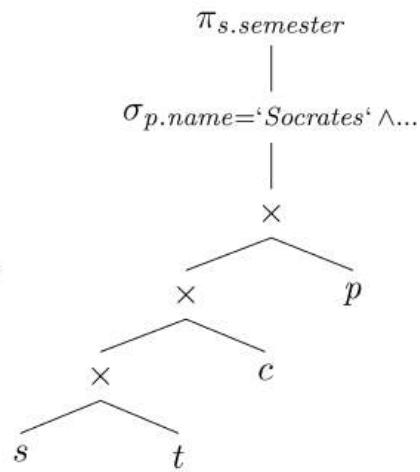
- Projektionen (SELECT-Klausel) wählen die benötigten Spalten aus. Manchmal müssen Projektionen eingefügt werden, um unnötige Spalten frühzeitig zu eliminieren.

### 6. Verschieben der Projektionen so weit wie möglich nach unten:

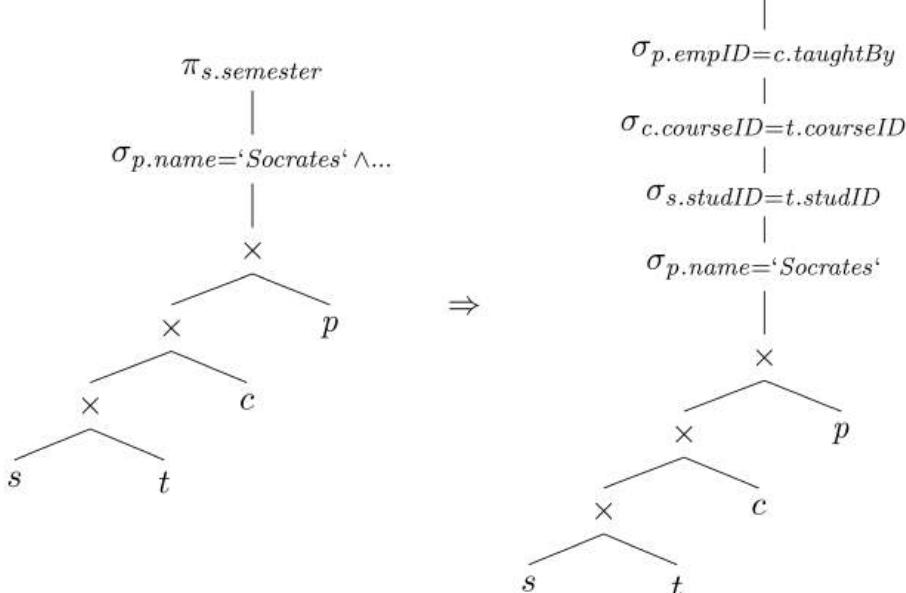
- Ähnlich wie bei Selektionen kann es vorteilhaft sein, Projektionen so früh wie möglich anzuwenden, um die Breite der Zwischenergebnisse zu reduzieren (d.h., weniger Spalten zu verarbeiten).
- **Nicht immer nötig:** Manchmal sind Projektionen nach Joins oder Aggregationen sinnvoller, insbesondere wenn die Projektionsspalten das Ergebnis von Aggregationen oder Join-Spalten sind, die erst später verfügbar werden. Die Regel aus der äquivalenzerhaltenden Transformation ( $\pi_{A_1, \dots, A_n}(\sigma_C(R)) = \sigma_C(\pi_{A_1, \dots, A_n}(R))$ ) ist hier relevant.

### ☰ Beispiel

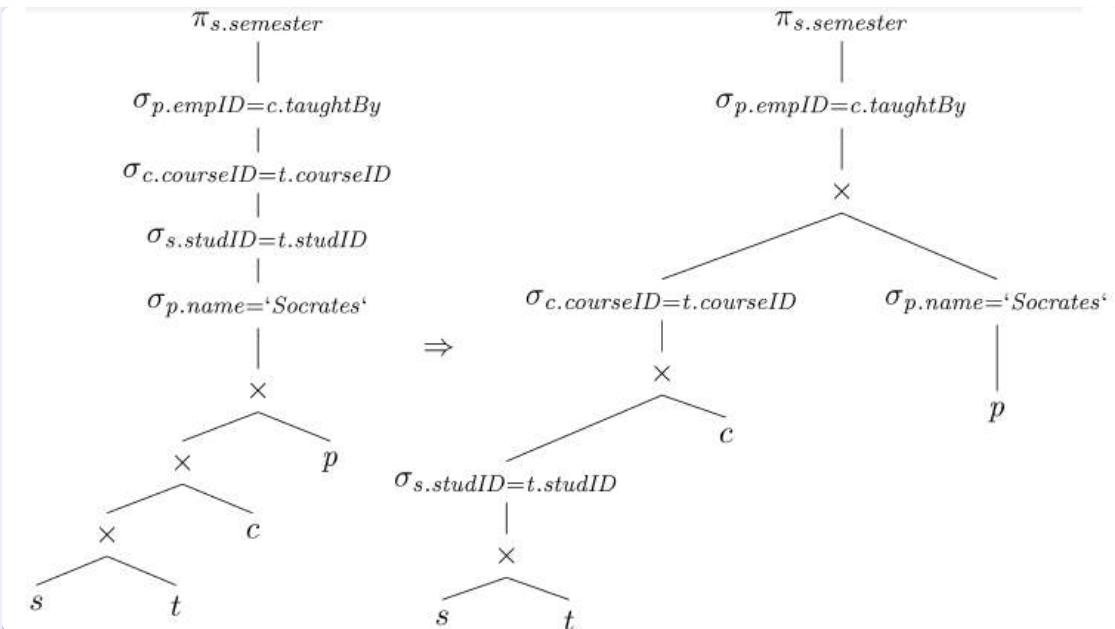
```
SELECT DISTINCT s.semester
FROM student s, takes t,
      course c, professor p
WHERE p.name='Socrates' AND
      c.taughtBy = p.empID AND
      c.courseID = t.courseID AND
      t.studID = s.studID;
```



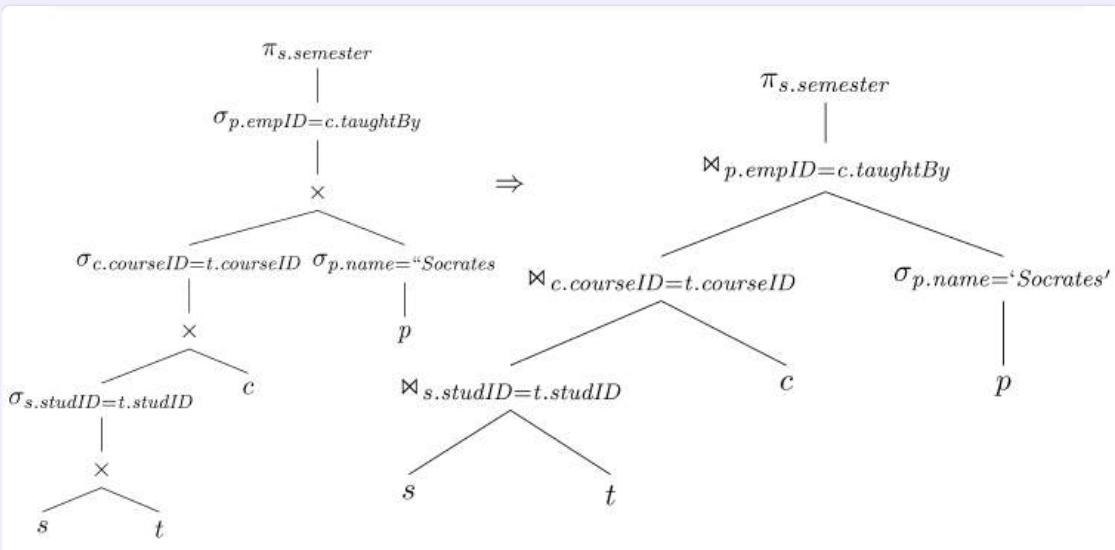
### Aufbrechen von Selektionen



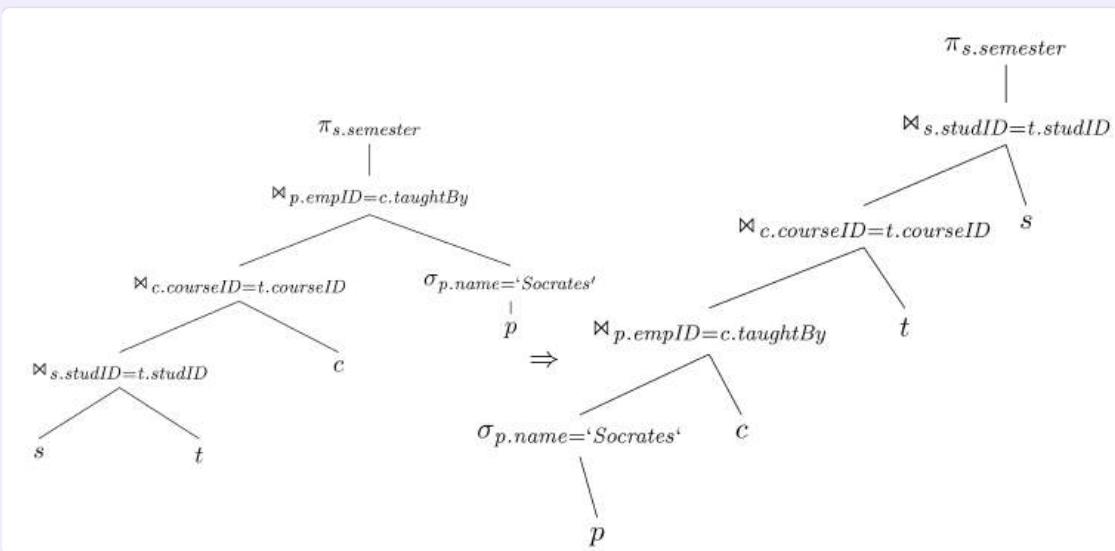
### Verschieben von Selektionen



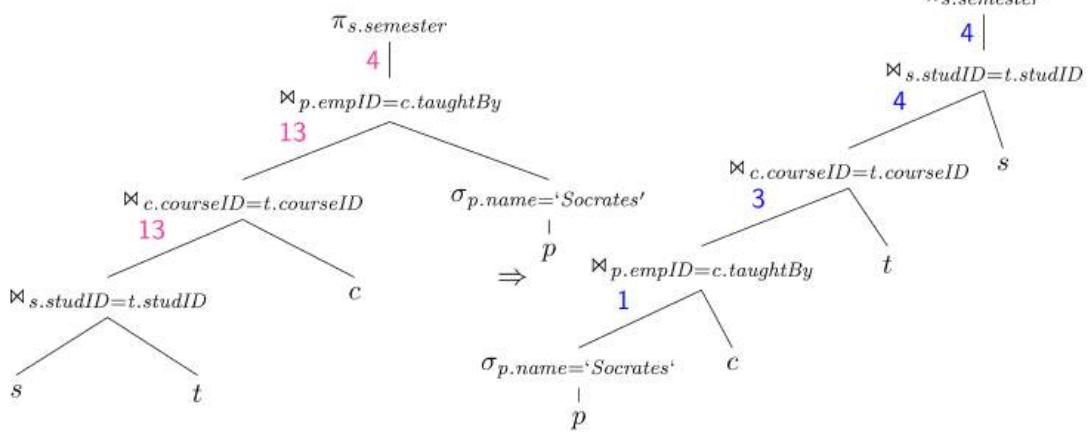
Joins einführen



Joinreihenfolge bestimmen

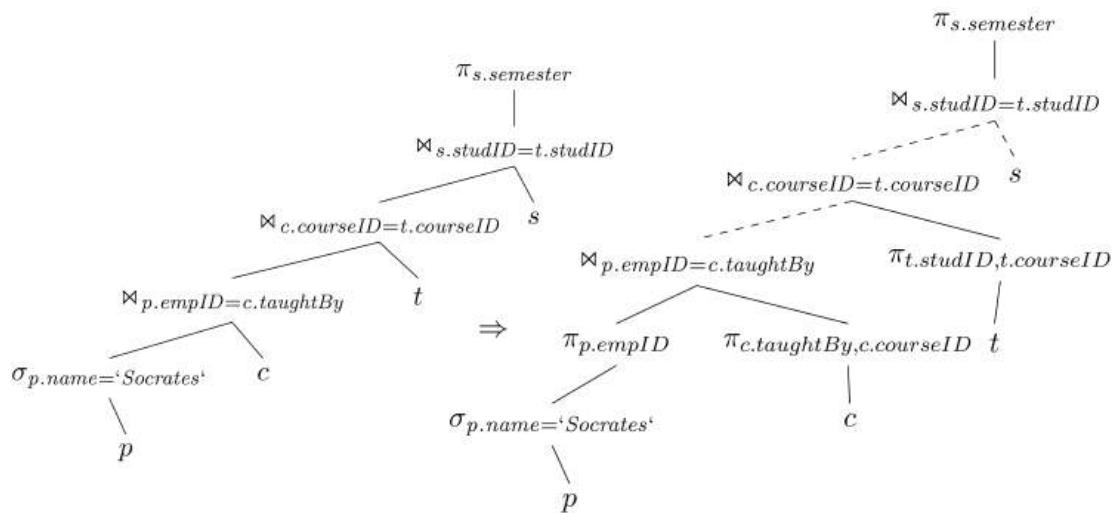


Effekt: Verkleinerung der Zwischenergebnisse



Schätzung der Zwischenergebnisgröße nur mit Statistiken möglich  
 → Kostenmodelle

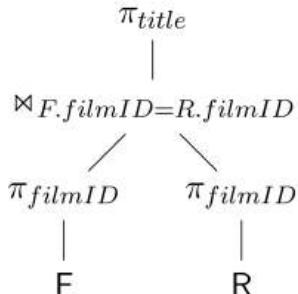
### Einführen und verschieben von Projektionen



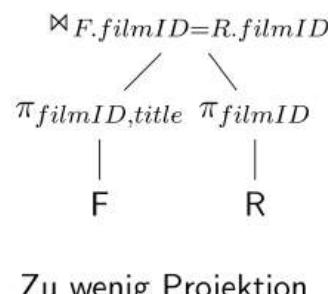
### ☰ Weitere Beispiele

## Finde die Titel von reservierten Filmen

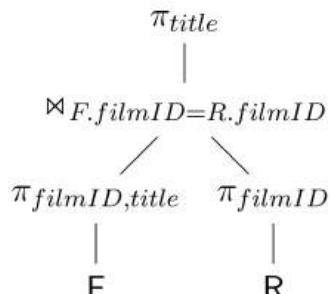
```
SELECT DISTINCT title  
FROM film F, reserved R  
WHERE F.filmID = R.filmID
```



### Zu viel Projektion



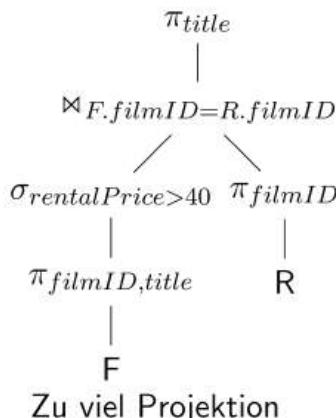
Zu wenig Projektion



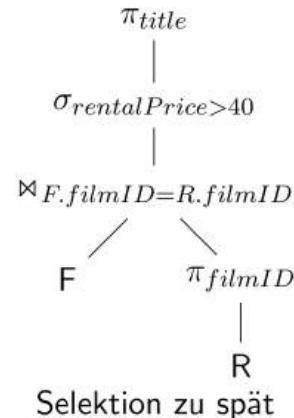
Korrekt

Finde die Titel von teuren reservierten Filmen

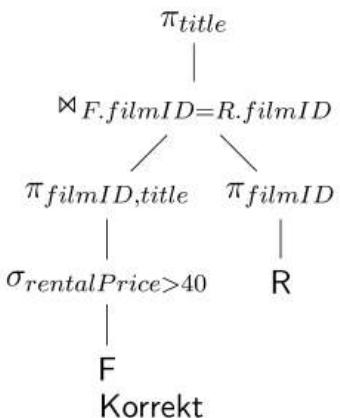
```
SELECT DISTINCT title  
FROM film F, reserved R  
WHERE F.filmID = R.filmID AND F.rentalPrice > 40
```



### Zu viel Projektion



## Selektion zu spät



Korrekt

## Zusammenfassung: heuristische Anfrageoptimierung

Die heuristische Anfrageoptimierung konzentriert sich auf bewährte "Faustregeln", um effiziente Anfragepläne zu erstellen, ohne eine vollständige Kostenanalyse für alle möglichen Pläne durchzuführen.

# Erfahrungsregeln

Die wichtigsten Heuristiken (Erfahrungsregeln) der logischen Anfrageoptimierung sind:

- Selektionen so früh wie möglich ausführen:

- Das Anwenden von Filtern ( WHERE -Klausel) so früh wie möglich im Ausführungsplan reduziert die Anzahl der Zeilen in Zwischenergebnissen drastisch.
- Kleinere Zwischenergebnisse führen zu geringeren E/A- und CPU-Kosten für nachfolgende Operationen wie Joins.
- **Projektionen so früh wie möglich ausführen:**
  - Das Entfernen von unnötigen Spalten ( SELECT -Klausel) so früh wie möglich im Ausführungsplan reduziert die Breite (Anzahl der Attribute) der Zwischenergebnisse.
  - Dies spart Speicherplatz und reduziert die Datenmenge, die zwischen den Operatoren bewegt werden muss.

## Optimierungsprozess

Der allgemeine Prozess der heuristischen Anfrageoptimierung umfasst zwei Schritte:

1. **Erstelle initialen Plan aus der SQL-Anfrage:**

- Die ursprüngliche SQL-Anfrage wird in einen ersten, meist naiven Operatorbaum der relationalen Algebra übersetzt (z.B. ein Parsen in einen Baum).

2. **Modifiziere den Plan, um ihn in einen effizienteren zu überführen:**

- Äquivalenzerhaltende Transformationsregeln und die oben genannten Heuristiken werden angewendet, um den initialen Plan schrittweise in einen effizienteren Ausführungsplan umzuwandeln.

## Hinweis

- **Anfrageergebnisse werden durch einen einzigen Plan berechnet:**
  - Obwohl es viele äquivalente Ausführungspläne geben kann, wählt der Optimierer letztendlich *einen* Plan aus, der dann zur Berechnung des Anfrageergebnisses verwendet wird.

## Implementierung von Operatoren

---

### Selektion (Access Paths)

#### Verschiedene Verwendungen von Select

- Primary Key, Punkt

$$\sigma_{filmID=2}(film)$$

- Punkt

$$\sigma_{title='Terminator'}(film)$$

- Bereich

$$\sigma_{1 < rentalPrice < 4}(film)$$

- Konjunktion (logisches und)

$$\sigma_{kind='F' \wedge rentalPrice=4}(film)$$

- Disjunktion (logisches oder)

$$\sigma_{rentalPrice < 2 \vee kind='D'}(film)$$

## Selektion - Punkt-/Bereichsanfragen

Bei der Selektion (dem Filtern von Daten) gibt es verschiedene grundlegende Suchstrategien, die je nach Datenstruktur und Art der Anfrage angewendet werden können:

- **Linear Search (Sequenzielle Suche):**

- **Aufwändig:** Jedes Tupel (jede Zeile) der Relation muss gelesen und geprüft werden.
- **Funktioniert aber immer:** Unabhängig davon, ob die Daten sortiert sind oder Indizes existieren.

- **Binary Search (Binäre Suche):**

- **Nur wenn die Datei entsprechend sortiert ist:** Setzt voraus, dass die Daten physisch nach dem Suchkriterium sortiert sind.

- **Primary Hash Index:**

- **Single Record Retrieval** – Funktioniert sehr effizient für den direkten Zugriff auf einzelne Datensätze über einen exakten Schlüsselwert (Gleichheitsanfragen).
- **Funktioniert nicht für Bereichsanfragen:** Da Hash-Funktionen keine Ordnung herstellen.

- **Primary/Clustering Index:**

- Mehrere Records für jeden Wert.
- Pointer zum Block mit dem ersten Record (Daten sind physisch nach Index sortiert).
- **Vorteil:** Ideal für Bereichsanfragen, da zugehörige Datenblöcke sequenziell gelesen werden können.

- **Secondary Index:**

- Jeder Record hat einen eigenen Pointer.
- **Kann teuer werden:** Insbesondere bei Bereichsanfragen, die viele Tupel zurückgeben, da dies zu vielen **zufälligen** E/A-Zugriffen auf der Festplatte führen kann.

## Strategien für konjunktive Anfragen

### Selektion (Access Paths)

- Beispiel einer SQL-Anfrage mit konjunktiven Bedingungen:

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
AND state = 'Arizona'
```

- Nutzung von Indizes bei konjunktiven Bedingungen:

- Kann ein Index auf (*name*) oder (*street*) benutzt werden? **Ja** (für die jeweilige Bedingung)
- Kann ein Index auf (*name, street, state*) benutzt werden? **Ja** (präfix-basiert, also wenn die ersten Spalten im Index vorkommen)
- Kann ein Index auf (*name, street*) benutzt werden? **Ja** (präfix-basiert)
- Kann ein Index auf (*name, street, city*) benutzt werden? **Ja** (hier wird der Index bis (*name, street*) genutzt, die *city*-Spalte im Index ist für diese Anfrage nicht relevant)
- Kann ein Index auf (*city, name, street*) benutzt werden? **Nein** (da die Abfragebedingungen *name, street, state* sind und keine Präfix-Übereinstimmung mit *city* vorliegt, kann dieser Index nicht **effektiv** genutzt werden)

### Optimierung von konjunktiven Anfragen

- Indizes bieten gute Möglichkeiten, die Performance von Anfragen zu verbessern. (Insbesondere bei konjunktiven Bedingungen, da sie den Zugriff auf die Daten beschleunigen können).

## Strategien für konjunktive Anfragen (Fortsetzung)

- Existierende Indizes verwenden:
  - **Idealfall:** Es existiert ein Index, der alle Attribute abdeckt, die in der Anfrage benötigt werden.
  - **Falls es mehrere Indizes gibt:**
    - Der Datenbankoptimierer wählt den Index, der am selektivsten ist (d.h. der die Anzahl der potenziellen Ergebnisse am stärksten reduziert).
    - Die verbleibenden Bedingungen werden anschließend auf die durch den Index gefilterten Ergebnisse angewendet (ausgewertet).
- **Überschneidung von Pointern ausnutzen (Index Merge / Intersection):**
  - Diese Strategie wird angewendet, wenn mehrere Indizes auf verschiedene Attribute in einer konjunktiven Anfrage anwendbar sind.
  - **Schritte:**

1. **Index Lookups:** Für jede Bedingung, für die ein Index existiert, werden die relevanten Pointer (Verweise auf die Datenblöcke/Tupel) ermittelt.
2. **Überschneidung der Pointer:** Die Schnittmenge (Konjunktion) dieser Pointer wird gebildet. Das bedeutet, es werden nur die Pointer behalten, die in *allen* relevanten Index-Lookups vorkommen. (Also nur die Tupel, die *alle* Bedingungen erfüllen.)
3. **Records/Tupel lesen:** Basierend auf den ermittelten, überschneidenden Pointern werden die zugehörigen Records/Tupel aus der Tabelle gelesen.

## Disjunktive Anfragen (mit OR-Verknüpfungen)

- Disjunktive Anfragen bieten **wenig Gelegenheit zur Optimierung** durch Indizes im Vergleich zu konjunktiven Anfragen. (Oft muss hier ein Full Table Scan durchgeführt werden, es sei denn, die Indizes decken sehr spezifische Fälle ab oder es gibt eine Möglichkeit, die OR-Bedingungen in eine Reihe von UNION-Operationen umzuwandeln, die Indizes nutzen können.)

## Wichtigkeit von Tuning

- **Tuning und das Anlegen von Indizes ist wichtig!** (Eine gute Indexstrategie ist entscheidend für die Performance von Datenbankabfragen.)
- 

## Join Algorithmen

### Algorithmen

- **Nested Loop Join (Geschachtelter Schleifen-Join):** Ein grundlegender Join-Algorithmus, bei dem für jedes Tupel der äußeren Relation die innere Relation durchlaufen wird.
- **Index-based Join (Index-basierter Join):** Nutzt Indizes auf einer der Relationen, um den Join-Prozess zu beschleunigen.
- **Sort-Merge Join (Sortier-Misch-Join):** Beide Relationen werden nach dem Join-Attribut sortiert und anschließend in einem Merge-Schritt zusammengeführt.
- **Hash Join (Hash-Join):** Erstellt Hash-Tabellen für eine oder beide Relationen, um Tupel mit übereinstimmenden Join-Attributen schnell zu finden.

### Strategien basieren auf Blöcken (nicht Tupeln) als Basis

- Bei der Kostenabschätzung von Join-Algorithmen geht man davon aus, dass Daten in Blöcken (Seiten) und nicht in einzelnen Tupeln gelesen werden.
- **Schätze I/Os (Block Retrievals):** Die Kosten eines Join-Algorithmus werden primär durch die Anzahl der benötigten Ein-/Ausgabeoperationen (I/Os), d.h. das Lesen von Blöcken vom Speicher (z.B. Festplatte) in den Hauptspeicher, bestimmt.

- **Benutze Puffer (Buffer) im Hauptspeicher:** Ein Puffer im Hauptspeicher wird genutzt, um die gelesenen Blöcke temporär zu speichern und die Anzahl der teuren Plattenzugriffe zu minimieren.

## Tabellengröße und Join-Selektivität bestimmen die Kosten

- **Selektivität einer Anfrage (sel):**

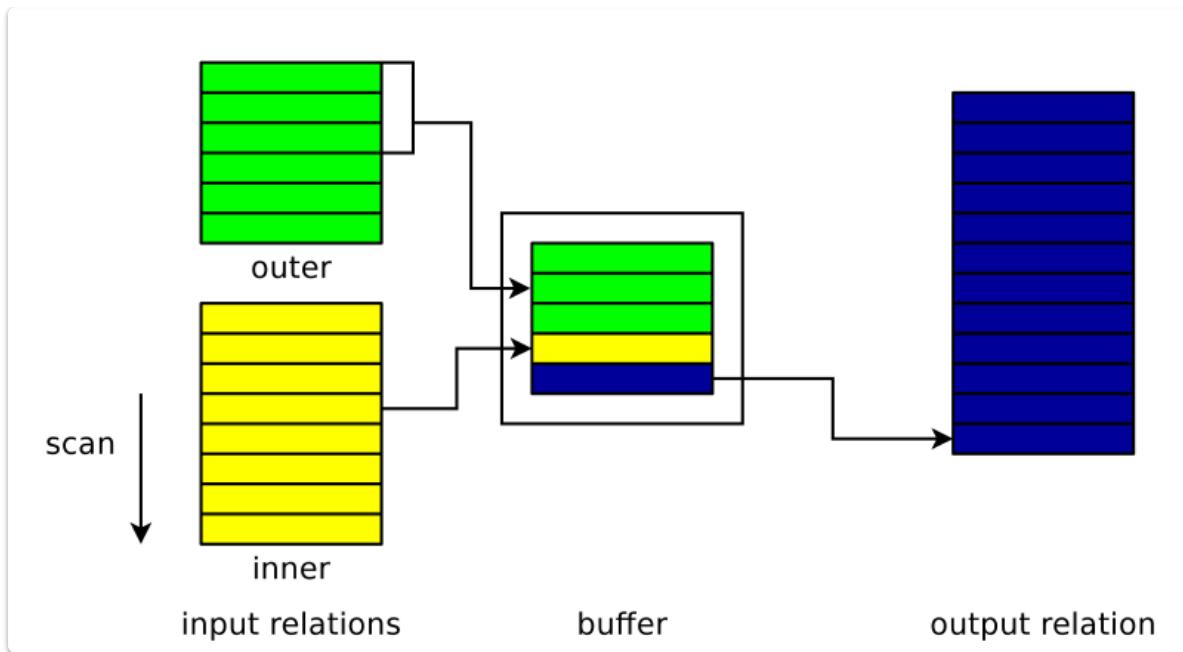
$$sel = \frac{\#\text{tuples in result}}{\#\text{candidates}}$$

- Beschreibt das Verhältnis der Anzahl der Tupel im Ergebnis zur Anzahl der potenziellen Tupel.
- Eine höhere Selektivität (näher an 1) bedeutet, dass ein größerer Anteil der Kandidaten in das Ergebnis aufgenommen wird. Eine geringere Selektivität (näher an 0) bedeutet, dass ein kleinerer Anteil der Kandidaten die Bedingung erfüllt.
- Für einen Join entspricht  $\#\text{candidates}$  der Größe des Kartesischen Produkts:
  - Das Kartesische Produkt zweier Relationen  $R$  und  $S$  ( $R \times S$ ) ist die Menge aller möglichen Tupelpaare, die durch die Kombination jedes Tupels aus  $R$  mit jedem Tupel aus  $S$  entstehen.
  - Die Anzahl der Kandidaten für einen Join ist somit  $|R| \times |S|$ , also die Produkt der Anzahlen der Tupel der beiden Relationen.
  - Die Join-Operation filtert dann dieses Kartesische Produkt basierend auf den Join-Bedingungen.

## Nested Loop Join

Sehr umfangreiches Beispiel von DBS-9, p.37 bis DBS-9, p.37

- **Brute-Force Strategie**
  - Sehr teuer, da **alle Tupel** (Datenzeilen) miteinander verglichen werden müssen.
  - **Kein Preprocessing** (Vorverarbeitung) der Input-Relationen (Tabellen) nötig.
  - **Kein Index** (Hilfsstruktur zur Beschleunigung von Datenzugriffen) erforderlich, da **alle Joinbedingungen** (Verknüpfungsregeln zwischen Tabellen) unterstützt werden.



- **Problemstellung:** Nicht alle Blöcke beider Relationen passen gleichzeitig in den Hauptspeicher. (Deshalb muss blockweise gelesen und verglichen werden.)
- **Algorithmus (Pseudocode):**

```

repeat
  lese  $n_B - 2$  Blöcke der äußeren Relation
  repeat
    lese 1 Block der inneren Relation
    Vergleiche enthaltene Tupel
  until innere Relation ist vollständig durchlaufen
  until äußere Relation ist vollständig durchlaufen

```

- **Parameter:**
  - $b_{inner}, b_{outer}$ : Anzahl der Blöcke der inneren bzw. äußeren Relation.
  - $n_B$ : Größe des Puffers im Hauptspeicher (Anzahl der Blöcke, die gleichzeitig in den Hauptspeicher geladen werden können).
- **Kostenschätzung (Anzahl der Block-Transfers / I/Os):**  
Die geschätzten Kosten in Form von Block-Transfers für den Block Nested Loop Join sind:

$$b_{outer} + \left( \left\lceil \frac{b_{outer}}{(n_B - 2)} \right\rceil \cdot b_{inner} \right)$$

- **Erklärung der Formel:**
  - $b_{outer}$ : Dies sind die Kosten für das *einmalige* Lesen der äußeren Relation.
  - $\left\lceil \frac{b_{outer}}{(n_B - 2)} \right\rceil$ : Dies ist die Anzahl der Iterationen der äußeren Schleife. In jeder Iteration werden  $n_B - 2$  Blöcke der äußeren Relation gelesen. Wir ziehen 2 ab, da ein Block für die innere Relation und ein weiterer Block für die Ausgabe des Join-Ergebnisses benötigt werden. Die Ceil-Funktion ( $\lceil \dots \rceil$ ) stellt sicher, dass auch der letzte, möglicherweise unvollständige Block-Batch berücksichtigt wird.
  - $b_{inner}$ : In jeder Iteration der äußeren Schleife wird die innere Relation *vollständig* gelesen.

- **Berechnung der Berechnungszeit:**

Wenn weitere Systemparameter wie Block-Transfer-Zeiten, Disk Seek-Zeiten, CPU-Geschwindigkeit, etc. sowie die Größen der Relationen bekannt sind, können wir die gesamte Berechnungszeit detaillierter abschätzen.

### ☰ Beispiel

- **Beispiel:** Join von *reserved*  $\bowtie$  *customer*
- **Anzahl der Blöcke:**
  - $b_{\text{reserved}} = 2.000$  (Anzahl der Blöcke der Relation *reserved*)
  - $b_{\text{customer}} = 10$  (Anzahl der Blöcke der Relation *customer*)
- **Größe des Buffers im Hauptspeicher:**
  - $n_B = 6$  (Anzahl der Blöcke, die gleichzeitig im Hauptspeicher gehalten werden können)
- **Kostenschätzung (Block Transfers) - Wiederholung der Formel:**

$$b_{\text{outer}} + \left( \left\lceil \frac{b_{\text{outer}}}{(n_B - 2)} \right\rceil \cdot b_{\text{inner}} \right)$$
- **Kostenberechnung für das Beispiel:**
  - **1. Fall:** *reserved* als äußere Relation ( $b_{\text{outer}} = b_{\text{reserved}}$ ,  $b_{\text{inner}} = b_{\text{customer}}$ )
    - $n_B - 2 = 6 - 2 = 4$
    - Kosten =  $2.000 + (\lceil 2.000/4 \rceil) \cdot 10$
    - Kosten =  $2.000 + (500) \cdot 10$
    - Kosten =  $2.000 + 5.000 = 7.000$  Block-Transfers
  - **2. Fall:** *customer* als äußere Relation ( $b_{\text{outer}} = b_{\text{customer}}$ ,  $b_{\text{inner}} = b_{\text{reserved}}$ )
    - $n_B - 2 = 6 - 2 = 4$
    - Kosten =  $10 + (\lceil 10/4 \rceil) \cdot 2.000$
    - Kosten =  $10 + (\lceil 2.5 \rceil) \cdot 2.000$
    - Kosten =  $10 + (3) \cdot 2.000$
    - Kosten =  $10 + 6.000 = 6.010$  Block-Transfers
- **Ergebnis des Beispiels:** Es ist effizienter, die kleinere Relation (*customer*) als äußere Relation zu wählen, um die Anzahl der Block-Transfers zu minimieren.

## Index-based Nested Loop Join

- **Gleiches Prinzip wie beim Standard Nested Loop Join:** Es gibt eine äußere und eine innere Relation.
- **Äußere Relation:** Wird sequenziell oder blockweise durchlaufen.
- **Innere Relation:** Der **File Scan** (komplettes Durchsuchen der Datei) der inneren Relation kann durch **Index Lookups** ersetzt werden. Das bedeutet: Für jedes Tupel der äußeren Relation wird der Index der inneren Relation verwendet, um passende Tupel schnell zu

finden, anstatt die gesamte innere Relation erneut zu scannen. Dies ist besonders effizient, wenn die innere Relation auf dem Join-Attribut indiziert ist.

## Merge Join

Ausnutzen der sortierten Reihenfolge

R				S	
	A	←	→	B	
...	0			5	...
...	7			6	...
...	7			7	...
...	8			8	...
...	8			8	...
...	10			11	...
...	...			...	...

Annahme:

Beide Input-Relationen sind sortiert

## Umfangreiches Beispiel

In den Slides von DBS-9, p.43|Seite 126 bis DBS-9, p.43|Seite 136

## Kosten von Merge Join

### Parameter

- $b_1, b_2$ : Anzahl der Blöcke der beiden zu joinenden Relationen.

### Kostenschätzung (Block Transfers)

- Wenn beide Relationen bereits nach dem Join-Attribut sortiert vorliegen, sind die Kosten für den Merge Join sehr gering:

$$b_1 + b_2$$

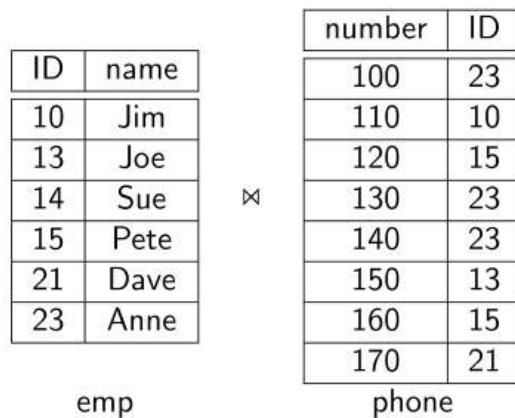
- Diese Kosten entstehen, weil jede Relation einmal sequenziell gelesen werden muss, um die Tupel zu mergen.

### Erweiterungen (Zusätzliche Überlegungen)

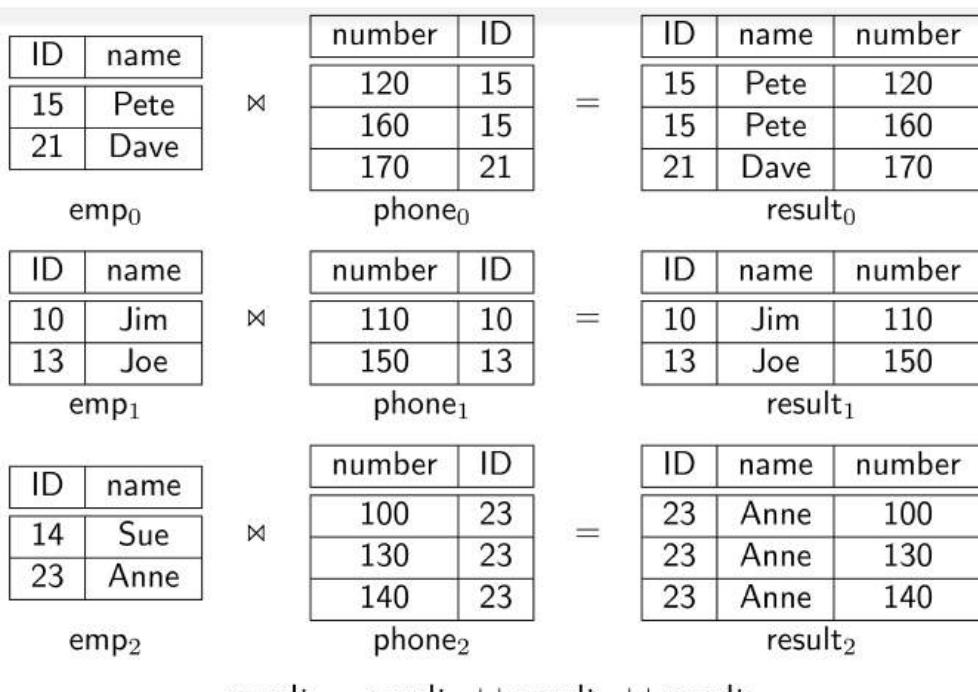
- **Kombination mit Sortierung, wenn Input-Relationen nicht sortiert vorliegen:**
  - Der Merge Join setzt voraus, dass die Input-Relationen nach dem Join-Attribut sortiert sind.

- Falls sie es nicht sind, müssen sie zuerst sortiert werden. Die Kosten für diese Sortierung (oft externer Sortieralgorithmus) müssen dann zu den  $b_1 + b_2$  Kosten hinzuaddiert werden. Diese Sortiekosten können erheblich sein.
  - **Nicht genügend Hauptspeicher:**
    - Wenn die Relationen zu groß sind, um vollständig in den Hauptspeicher geladen zu werden (was oft der Fall ist), müssen externe Sortier- und Merge-Verfahren angewendet werden. Dies erhöht die Komplexität und die I/O-Kosten des Algorithmus.

## Hash Join

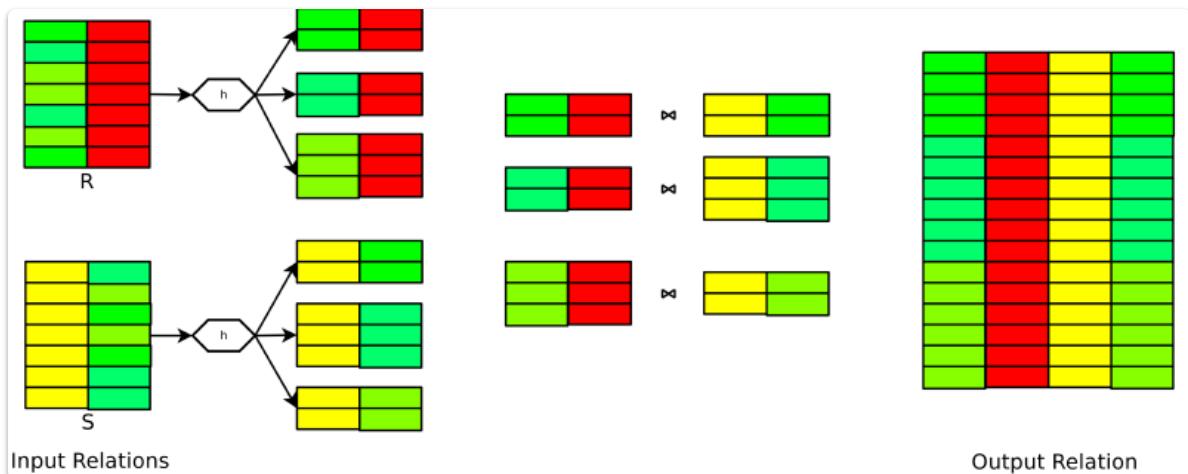


Wende Hashfunktion auf die Join-Attribute an  
→ Partitioniere Tupel in Buckets



- Jede Relation mit Hashfunktion partitionieren:

- Die Input-Relationen  $R$  und  $S$  werden mithilfe einer Hashfunktion  $h$  (auf das Join-Attribut angewendet) in mehrere Buckets aufgeteilt.
- **Jedes Bucket muss klein genug sein, um in den Hauptspeicher zu passen:**
  - Die erzeugten Buckets sollen idealerweise in den Hauptspeicher passen, um I/O-Operationen zu minimieren.
- **Join die "passenden" Buckets miteinander:**
  - Nur Buckets, die denselben Hash-Wert aufweisen (d.h. von derselben Partition stammen), müssen miteinander gejoined werden. (Z.B. Bucket 1 von  $R$  mit Bucket 1 von  $S$ , Bucket 2 von  $R$  mit Bucket 2 von  $S$ , etc.)



## Parameter

- $b_1, b_2$ : Anzahl der Blöcke der Relationen  $R_1$  und  $R_2$ .

### Schritte

#### 1. Partitioniere Relation $R_1$ mit $h_1$ in Buckets $r_1$ :

- Lese  $R_1$  komplett ein (`read all`).
- Schreibe  $R_1$  nach Hashing in Buckets auf die Platte (`write all`).
- Kosten:  $2 \times b_1$  (einmal lesen, einmal schreiben)

#### 2. Partitioniere Relation $R_2$ mit $h_1$ in Buckets $r_2$ :

- Lese  $R_2$  komplett ein (`read all`).
- Schreibe  $R_2$  nach Hashing in Buckets auf die Platte (`write all`).
- Kosten:  $2 \times b_2$  (einmal lesen, einmal schreiben)

#### 3. Build-Phase:

- Für jedes Bucket von  $r_1$  (aus der kleineren Relation):
  - Benutze eine Hashfunktion  $h_2$  (oft eine einfache interne Hash-Tabelle) zur Erstellung eines In-Memory Hash Index.
  - Lese das Bucket  $r_1$  (`read all`).
- Kosten:  $b_1$  (einmaliges Lesen der partitionierten  $R_1$  Buckets).

#### 4. Probe-Phase:

- Für jedes Bucket von  $r_2$ :
  - Benutze den In-Memory Hash Index, der in der Build-Phase erstellt wurde, um Joinpartner zu finden.
  - Lese das Bucket  $r_2$  (`read all`).
- Kosten:  $b_2$  (einmaliges Lesen der partitionierten  $R_2$  Buckets).

## Kostenschätzung (Block Transfers)

- Die Gesamtkosten für den Hash Join sind:

$$2 \cdot b_1 + 2 \cdot b_2 + b_1 + b_2 = 3 \cdot b_1 + 3 \cdot b_2$$

- Hier sind die Kosten für (unvollständig gefüllte Blöcke) mit  $\epsilon$  nicht explizit aufgeführt, aber die Formel  $3 \cdot b_1 + 3 \cdot b_2$  repräsentiert die Summe der Lese- und Schreiboperationen während der Partitionierungs- und Join-Phasen.

## Kosten und Anwendung von Join-Algorithmen

### Nested Loop Join

- Kann für alle Join-Typen verwendet werden. (Sehr flexibel, aber oft ineffizient).
- Kann sehr teuer werden. (Besonders bei großen Relationen, da viele I/O-Operationen anfallen können).

### Merge Join

- Daten müssen auf Joinattributen sortiert sein. (Wenn nicht, fallen zusätzliche Sortierkosten an).
- Sortierung kann für den Join vorgelagert vorgenommen werden. (D.h., wenn Daten bereits für andere Operationen sortiert wurden, können diese Sortierkosten für den Join wiederverwendet werden).
- Kann Indexe verwenden. (Wenn ein Index auf dem Join-Attribut existiert, kann dieser für die Sortierung genutzt werden oder um eine bereits sortierte Reihenfolge zu gewährleisten).

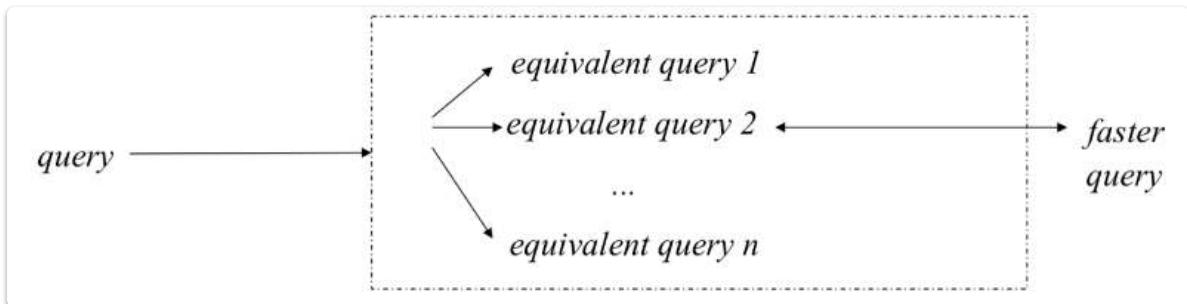
### Hash Join

- Gute Hashfunktionen sind die Grundlage. (Eine gleichmäßige Verteilung der Tupel auf die Buckets ist entscheidend für die Performance).
- Beste Performance, wenn die kleinere Relation in den Hauptspeicher passt. (Idealfall: Die kleinere Relation kann komplett in den Hauptspeicher geladen und dort eine Hash-Tabelle aufgebaut werden, was die I/O-Kosten stark reduziert).

# Kostenbasierte (physische) Anfrageoptimierung

## Ziel

Für eine gegebene Anfrage den besten Ausführungsplan finden.



## Optimierung

- **Heuristische (logische) Optimierung:**
  - Anfrageplan (relationale Algebra)
- **Kostenbasierte (physische) Optimierung:**
  - Anfrageausführungsplan (konkrete Algorithmen und Zugriffspfade)

## Physische Anfrageoptimierung

1. **Erstelle alternative Ausführungspläne:** Für eine Anfrage gibt es oft mehrere Wege, um das Ergebnis zu erzielen. Hier werden diese verschiedenen Möglichkeiten gesucht.
2. **Wähle Algorithmen und Zugriffspfade (Access Paths):** Für jeden Schritt in einem Ausführungsplan werden spezifische Algorithmen (z.B. für Join-Operationen) und Zugriffspfade (wie man auf die Daten zugreift, z.B. Index-Scan, Tabellen-Scan) ausgewählt.
3. **Berechne Kosten:** Für jeden erstellten Ausführungsplan werden die geschätzten Kosten berechnet. Diese Kosten können zum Beispiel CPU-Zeit, I/O-Operationen oder Netzwerknutzung umfassen.
4. **Wähle den günstigsten Ausführungsplan:** Der Ausführungsplan mit den geringsten geschätzten Kosten wird ausgewählt.

## Voraussetzung für die Kostenberechnung

- **Kostenmodell:** Ein Modell, das festlegt, wie die Kosten für verschiedene Operationen berechnet werden.
- **Statistiken für die Input-Relationen:**
  - Statistiken für Blattrelationen: Werden im Systemkatalog (System Catalog) gespeichert. (Blattrelationen sind die Tabellen, die direkt in der Datenbank existieren.)
  - Statistiken für Zwischenergebnisse müssen geschätzt werden (Kardinalität). (Kardinalität ist die Anzahl der Tupel (Zeilen) in einer Relation.)

## Selektivität und Kardinalität

### Statistiken pro Relation

Für eine Relation  $r$  werden folgende Statistiken erfasst:

- **Anzahl Tupel (Records):**  $n_r$  (Gesamtzahl der Zeilen in der Relation).
- **Größe der Tupel in Relation:**  $l_r$  (Durchschnittliche oder maximale Länge einer Zeile).
- **Füllgrad (Load/Fill Factor):** prozentuale Nutzung des Platzes je Block. (Zeigt an, wie voll die Datenblöcke sind. Ein hoher Füllgrad bedeutet, dass Blöcke effizient genutzt werden, aber auch, dass bei Einfügungen schneller neue Blöcke benötigt werden.)
- **Blocking Factor:** Anzahl Tupel pro Block. (Gibt an, wie viele Zeilen einer Relation in einen Datenblock passen.)
- **Größe der Relation in Blöcken:**  $b_r$  (Gesamtzahl der Blöcke, die die Relation belegt).
- **Organisation der Relation:** Informationen darüber, wie die Daten physisch gespeichert sind (z.B. Heap, Hash, Indexes, Sortierung).
- **Anzahl Overflow-Buckets:** Relevant bei bestimmten Speicherorganisationen (z.B. Hashing) für Daten, die nicht in ihren primären Speicherbereich passen.

### Statistiken pro Attribut

Für ein Attribut  $A$  in einer Relation  $r$  werden folgende Statistiken erfasst:

- **Größe und Typ:** Datentyp und Speichergröße des Attributs (z.B. INTEGER, VARCHAR(255)).
- **Anzahl "distinct" Werte für Attribut  $A$ :**  $V(A, r)$  (Anzahl der einzigartigen Werte, die dieses Attribut in der Relation annimmt).
  - Entspricht der Kardinalität der Projektion  $\pi_A(r)$ . ( $\pi_A(r)$  projiziert die Relation  $r$  auf das Attribut  $A$ , wodurch nur die eindeutigen Werte von  $A$  übrig bleiben.)
- **Kardinalität einer Selektion  $S(A, r)$  für beliebigen Wert  $a$ :** (Anzahl der Tupel, die das Attribut  $A$  mit einem spezifischen Wert  $a$  enthalten).
  - Entspricht der Kardinalität der Selektion  $\sigma_{A=a}(r)$ . ( $\sigma_{A=a}(r)$  wählt alle Tupel aus der Relation  $r$  aus, bei denen das Attribut  $A$  den Wert  $a$  hat.)
- **Wahrscheinlichkeitsverteilung der Werte:** Beschreibt, wie die Werte des Attributs verteilt sind.
  - Alternativ: **Gleichverteilung annehmen.** (Wenn keine genauen Verteilungsstatistiken vorliegen, wird oft angenommen, dass alle Werte gleich häufig vorkommen.)

**Wichtig:** Statistiken müssen aktualisiert werden, wenn eine Tabelle aktualisiert wird!

### Statistiken pro Index

Für einen Index werden folgende Statistiken erfasst:

- **Basisrelation:** Die Relation, auf der der Index aufgebaut ist.

- **Indexierte Attribute:** Die Attribute der Basisrelation, die im Index enthalten sind.
- **Organisation:** Die Struktur des Indexes, z.B. B<sup>+</sup>-Baum, Hash.
- **Clustering Index?**: Gibt an, ob es sich um einen Clustering-Index handelt (d.h., die physische Reihenfolge der Daten entspricht der Indexreihenfolge).
- **Schlüsselattribut(e)?**: Gibt an, ob der Index auf Schlüsselattributen basiert.
- **Sparse oder dense?**:
  - **Sparse Index:** Enthält nur Einträge für eine Untermenge der Daten (z.B. nur für Block-Pointer).
  - **Dense Index:** Enthält einen Eintrag für jeden einzelnen Datensatz.
- **Anzahl der Ebenen:** Die Anzahl der Hierarchieebenen im Index (z.B. bei einem B<sup>+</sup>-Baum).
- **Anzahl Blätter:** Die Anzahl der Blattknoten im Index (dort, wo die tatsächlichen Daten-Pointer oder Daten selbst gespeichert sind).

## Kostenschätzung

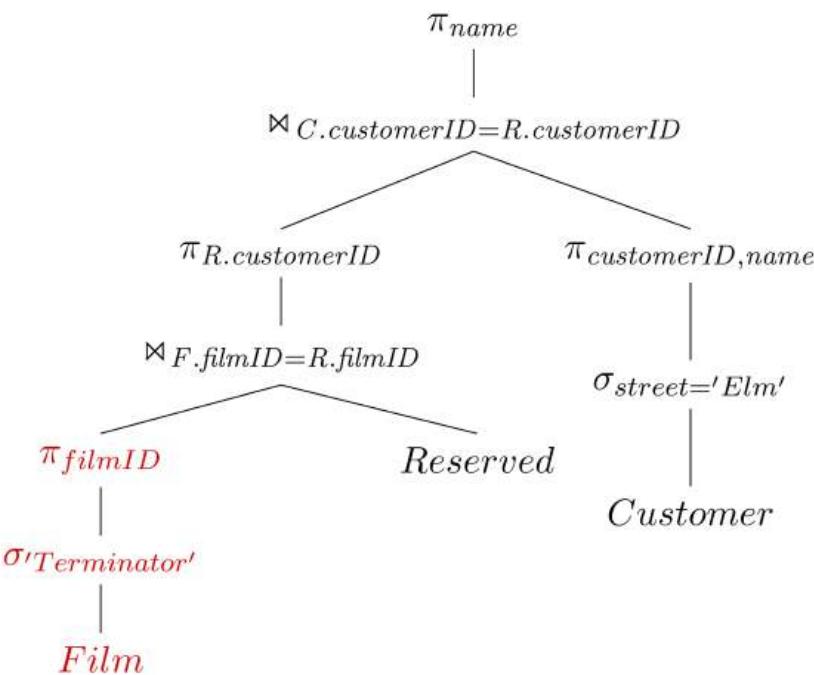
### Beispiel

#### Anfrage

Welche Kunden wohnen in der Elm Street und haben den Film "Terminator" reserviert?

```
SELECT name
FROM customer C, reserved R, Film F
WHERE C.customerID = R.customerID
  AND F.filmID = R.filmID
  AND F.title = 'Terminator'
  AND C.street = 'Elm';
```

#### Anfragebaum (Beispiel)



### Beispielhafte Teilanfrage: Projektion der Film-IDs für 'Terminator'

Ziel:

$$\pi_{filmID}(\sigma_{title='Terminator'}(Film))$$

### Statistiken für diese Teilanfrage

- Statistiken für Relation `Film`:
  - Anzahl Tupel:  $n_{Film} = 5000$
  - Größe der Relation in Blöcken:  $b_{Film} = 50$
- Statistiken für Attribute:
  - Kardinalität der Selektion  $S(title, Film) = 1$  (Es gibt nur einen Film mit dem Titel 'Terminator').
- Statistiken für Index:
  - Hash Index für Attribut „title“.

### Ausführung

- Benutze Index und suche „Terminator“:
  - Kosten für den Disk-Zugriff:

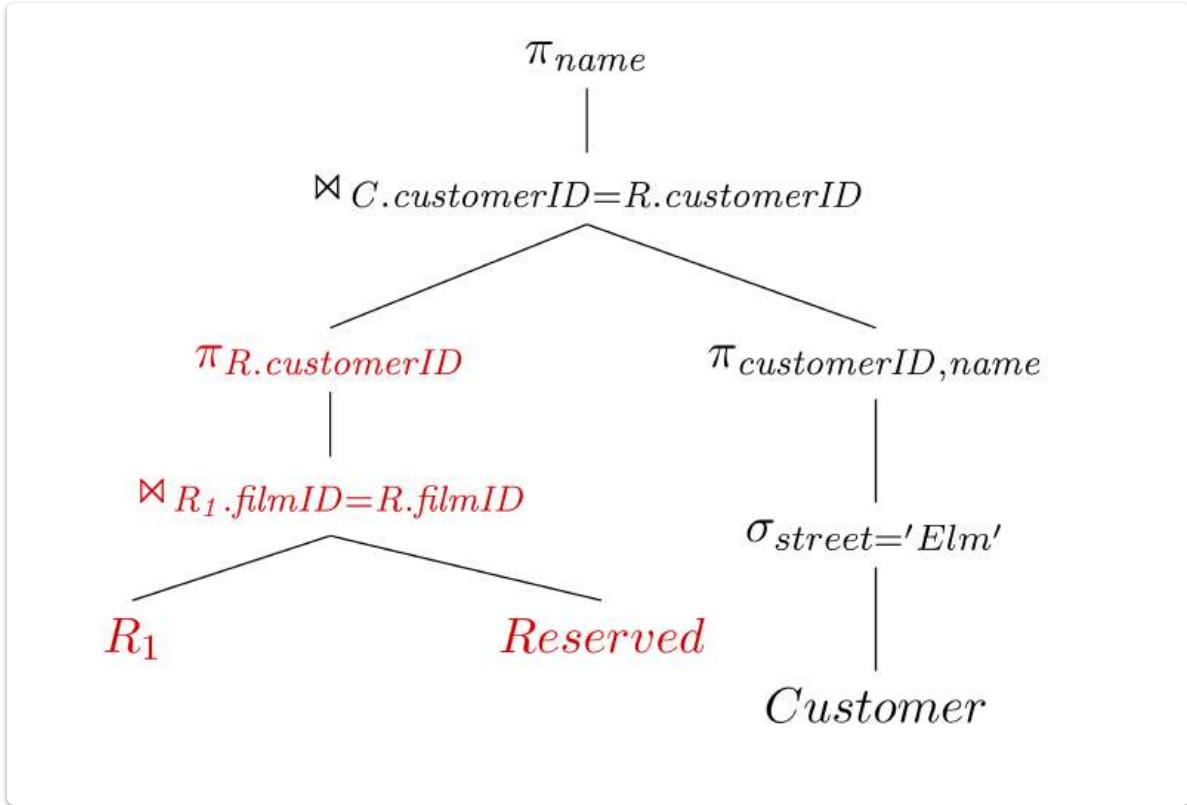
$$cost_{disk\_access} = 1$$

(Da ein Hash-Index auf 'title' existiert und die Kardinalität der Selektion 1 ist, wird angenommen, dass nur ein einziger Disk-Zugriff benötigt wird, um den entsprechenden Eintrag zu finden.)

- Projektion auf `filmID`: Das Ergebnis der Selektion wird auf das Attribut `filmID` projiziert.
- Ergebnisgröße: 1 Tupel (Da nur ein Film den Titel "Terminator" hat und dessen Film-ID projiziert wird).

- **Speicherung des Ergebnisses:** Das Ergebnis wird im Hauptspeicher gelassen (benötigt 1 Block).

## Anfragebaum (Fortsetzung)



**Beispielhafte Teilanfrage:** Join der Ergebnisse der Film-Selektion ( $R_1$ ) mit der **Reserved**-Relation

Ziel:

$$\pi_{R.customerID}(R_1 \bowtie_{R.filmID=F.filmID} \text{Reserved})$$

Dabei ist  $R_1$  das Ergebnis der vorherigen Teilanfrage ( $\pi_{filmID}(\sigma_{title='Terminator'}(Film))$ ), also das Tupel mit der `filmID` von 'Terminator'.

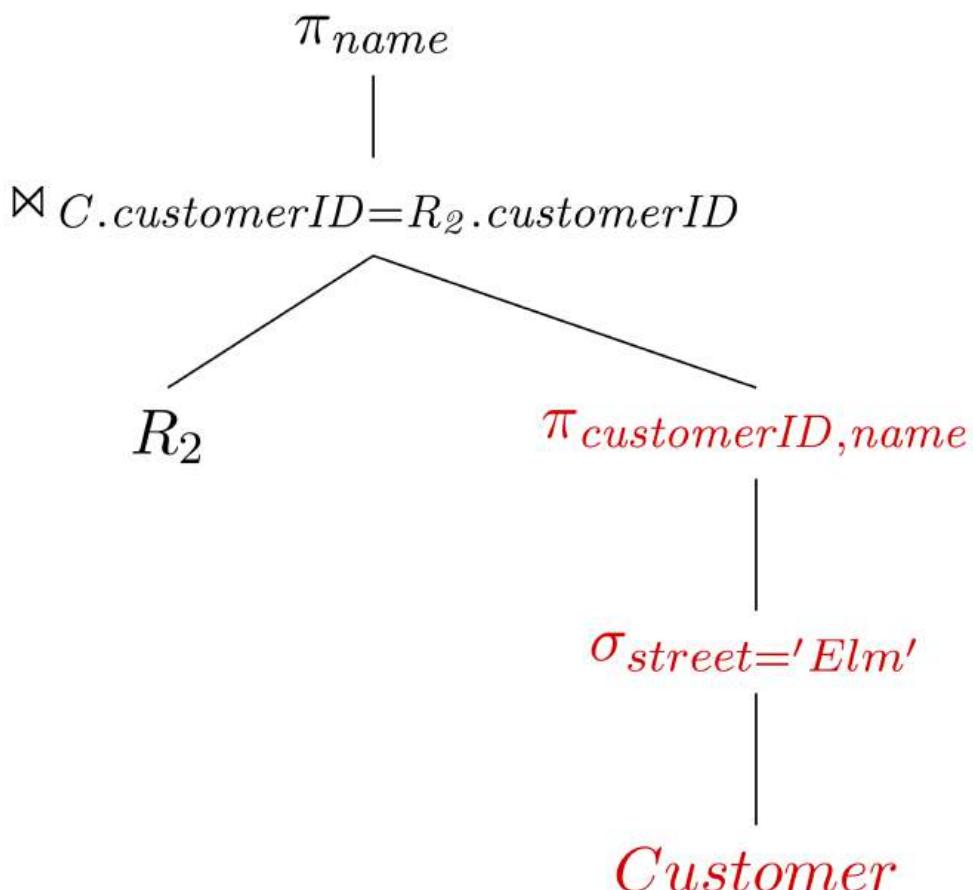
### Statistiken für diese Teilanfrage

- **Statistiken für Relation `Reserved`:**
  - Anzahl Tupel:  $n_{\text{Reserved}} = 40000$
  - Größe der Relation in Blöcken:  $b_{\text{Reserved}} = 2000$
- **Statistiken für Attribute:**
  - Kardinalität der Selektion  $S(filmID, Reserved) = 8$  (Dies bedeutet, dass es 8 Reservierungen für den Film 'Terminator' gibt.)
- **Statistiken für Index:**
  - Primary B<sup>+</sup>-Baum Index für `Reserved` auf `filmID` mit 3 Ebenen.

### Ausführung der Teilanfrage

- Index Join unter Verwendung des  $B^+$ -Baums:
  - Kosten für Disk-Zugriff:  $cost_{disk\_access} = 2$ 
    - (Diese Kosten resultieren aus dem letzten Indexebene auf der Platte, um den Datensatz im Index zu finden, und einem weiteren Zugriff, um das Tupel in der Relation `Reserved` zu lesen. Der  $B^+$ -Baum hat 3 Ebenen, d.h. der Zugriff auf den Blattknoten des Index kostet ca. 1 Leseoperation, und der Zugriff auf den Datenblock der Relation `Reserved` kostet eine weitere Leseoperation. Also  $1 + 1 = 2$ .)
  - Ergebnisgröße: 8 Tupel (Basierend auf der Kardinalität der Selektion  $S(filmID, Reserved) = 8$ ).
  - Projektion auf `customerID`: Die `customerID` aus den 8 Tupeln der `Reserved`-Relation wird projiziert.
  - Lasse Ergebnis im Hauptspeicher (1 Block): Das Ergebnis von 8 Tupeln wird im Hauptspeicher gehalten.

### Anfragebaum (Fortsetzung)



**Beispielhafte Teilanfrage: Selektion der Kunden aus 'Elm Street'**

Ziel:

$$\pi_{customerID, name}(\sigma_{street='Elm'}(Customer))$$

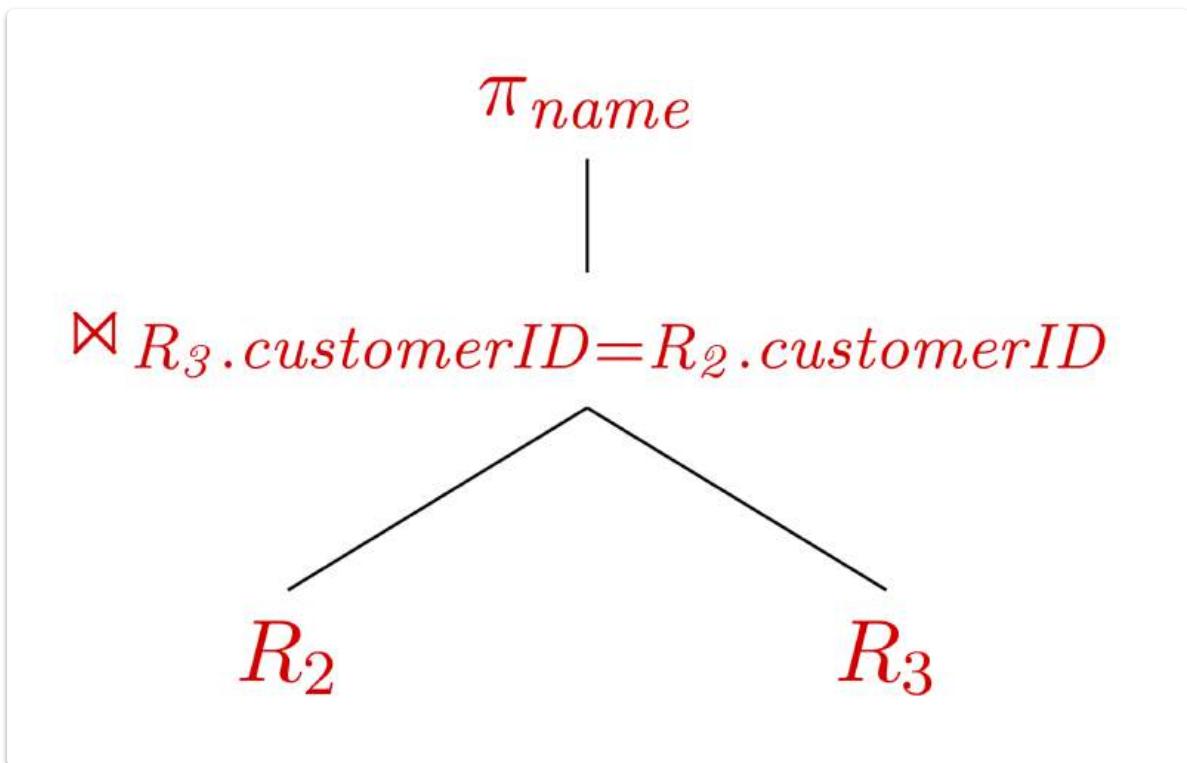
## Statistiken für diese Teilanfrage

- Statistiken für Relation `Customer`:
  - Anzahl Tupel:  $n_{Customer} = 200$
  - Größe der Relation in Blöcken:  $b_{Customer} = 10$
- Statistiken für Attribute:
  - Kardinalität der Selektion  $S(street, Customer) = 10$  (Es gibt 10 Kunden in der Elm Street).
- Statistiken für Index:
  - Kein Index auf „street“.

## Ausführung der Teilanfrage

- Lineare Suche in `Customer`: Da kein Index auf dem Attribut `street` vorhanden ist, muss die gesamte `Customer`-Relation durchsucht werden.
  - Kosten für Disk-Zugriff:  $cost_{disk\_access} = 10$  (entspricht  $b_{Customer}$ , da alle 10 Blöcke der `Customer`-Relation gelesen werden müssen).
- Projektion auf `customerID, name`: Die `customerID` und `name` der selektierten Tupel werden projiziert.
  - Ergebnisgröße: 10 Tupel (entspricht der Kardinalität der Selektion).
- Lass Ergebnis im Hauptspeicher (1 Block): Das Ergebnis wird im Hauptspeicher gehalten.

## Anfragebaum (Abschluss)



## Beispielhafte Teilanfrage: Finaler Join und Projektion

Ziel:  $\pi_{name}(R_2 \bowtie_{R_3.\text{customerID}=R_2.\text{customerID}} R_3)$

Dabei ist:

- $R_2$ : Ergebnis der Selektion und Projektion der `Reserved`-Tabelle auf `customerID` (8 Tupel).
- $R_3$ : Ergebnis der Selektion und Projektion der `Customer`-Tabelle auf `customerID` und `name` (10 Tupel).

## Ausführung der Teilanfrage

- **Join im Hauptspeicher:** Da  $R_2$  (8 Tupel) und  $R_3$  (10 Tupel) klein genug sind, um im Hauptspeicher zu liegen, kann der Join direkt dort durchgeführt werden, ohne weitere Disk-Zugriffe.
  - Kosten für Disk-Zugriff: 0 (keine neuen Plattenzugriffe für diese Operation).

## Gesamtkosten (Total Costs)

Die Gesamtkosten für Disk-Zugriffe ergeben sich aus der Summe der Kosten der einzelnen Teilschritte:

$$cost_{disk\_access} = 1 + 2 + 10 + 0 = 14$$

Dabei stehen die einzelnen Summanden für:

- 1: Kosten für den Zugriff auf den `Film`-Index (für 'Terminator').
- 2: Kosten für den Join mit `Reserved` (Indexzugriff und Datensatzlesen).
- 10: Kosten für die lineare Suche in `Customer` (alle Blöcke lesen).
- 0: Kosten für den finalen Join im Hauptspeicher.

## Kostenmodell

Kostenmodelle erfassen mehr als nur Festplattenzugriffe. Sie berücksichtigen unter anderem:

- **CPU Time** (Prozessorzeit)
- **Communication Time** (Kommunikationszeit, z.B. bei verteilten Datenbanken)
- **Main Memory Usage** (Hauptspeichernutzung)
- ... (Weitere Faktoren)

## Größen von Input/Output für jede Operation abschätzen

- **Statistiken für Relationen:** Im System Catalog gespeichert.
- **Statistiken für Zwischenergebnisse:** Müssen geschätzt werden (insbesondere Kardinalität).

## Weitere Aspekte der Anfrageoptimierung

- **Suchraum aufspannen:** Methoden wie Dynamic Programming, Exhaustive Search, etc. werden verwendet, um alle möglichen Ausführungspläne zu finden oder zu generieren.
- **Bushy vs. Left-Deep Join-Trees:**
  - **Bushy Join-Trees:** Erlauben, dass beide Operanden eines Joins das Ergebnis eines anderen Joins sein können. Bieten mehr Parallelisierungsmöglichkeiten (Parallelität).
  - **Left-Deep Join-Trees:** Der rechte Operand eines Joins ist immer eine Basisrelation. Gut für Pipelining.
- **Multiquery-Optimization (Shared Scans...):** Optimierung, die mehrere Anfragen gleichzeitig betrachtet, um gemeinsame Operationen (z.B. Scans) zu identifizieren und nur einmal auszuführen.

## Heuristische vs. Kostenbasierte Anfrageoptimierung

### Heuristisch

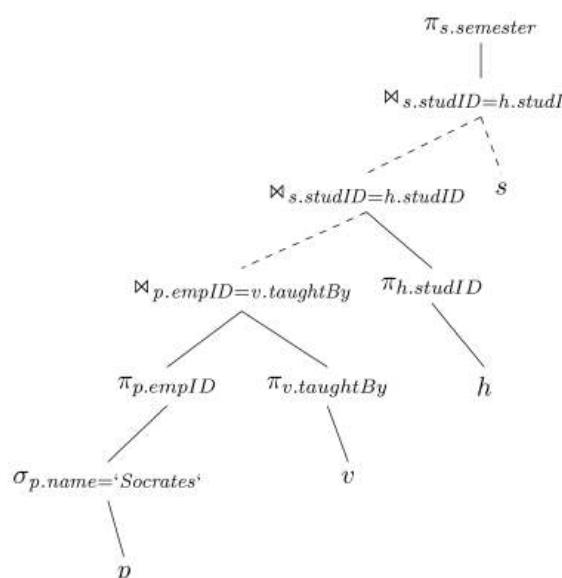
- Kann immer verwendet werden.
- Anfragepläne werden sequenziell generiert.
- Jeder Plan ist (wahrscheinlich) effizienter als der vorherige.
- Suche ist linear.

### Kostenbasiert

- Kann nur verwendet werden, wenn Statistiken vorliegen.
- Mehrere Pläne werden generiert.
- Kosten für jeden Plan werden berechnet, um den günstigsten zu wählen.
- Suche ist mehrdimensional.

## Postgre SQL

```
SELECT DISTINCT s.semester
FROM student s, takes h,
      course v, professor p
WHERE p.name='Socrates' AND
      v.taughtBy = p.empID AND
      v.courseID = h.courseID AND
      h.studID = s.studID;
```



## Explain

```
EXPLAIN SELECT DISTINCT s.semester
FROM student s, takes h,
    course v, professor p
WHERE p.name='Socrates' AND
      v.taughtBy = p.empID AND
      v.courseID = h.courseID AND
      h.studID = s.studID;
```

### EXPLAIN

Zeige den Ausführungsplan, den PostgreSQL für das Statement generiert

#### QUERY PLAN text

Unique (cost=4.61..4.62 rows=2 width=4)
-> Sort (cost=4.61..4.62 rows=2 width=4)
Sort Key: s.semester
-> Hash Join (cost=3.47..4.60 rows=2 width=4)
Hash Cond: (s.studid = h.studid)
-> Seq Scan on student s (cost=0.00..1.08 rows=8 width=8)
-> Hash (cost=3.45..3.45 rows=2 width=4)
-> Hash Join (cost=2.26..3.45 rows=2 width=4)
Hash Cond: (h.courseid = v.courseid)
-> Seq Scan on takes h (cost=0.00..1.13 rows=13 width=8)
-> Hash (cost=2.25..2.25 rows=1 width=4)
-> Hash Join (cost=1.10..2.25 rows=1 width=4)
Hash Cond: (v.taughtby = p.empid)
-> Seq Scan on course v (cost=0.00..1.10 rows=10 width=8)
-> Hash (cost=1.09..1.09 rows=1 width=4)
-> Seq Scan on professor p (cost=0.00..1.09 rows=1 width=4)
Filter: ((name)::text = 'Socrates'::text)

## Explain Analyze

```
EXPLAIN ANALYZE SELECT DISTINCT s.semester
FROM student s, takes h,
     course v, professor p
WHERE p.name='Socrates' AND
      v.taughtBy = p.empID AND
      v.courseID = h.courseID AND
      h.studID = s.studID;
```

## EXPLAIN ANALYZE

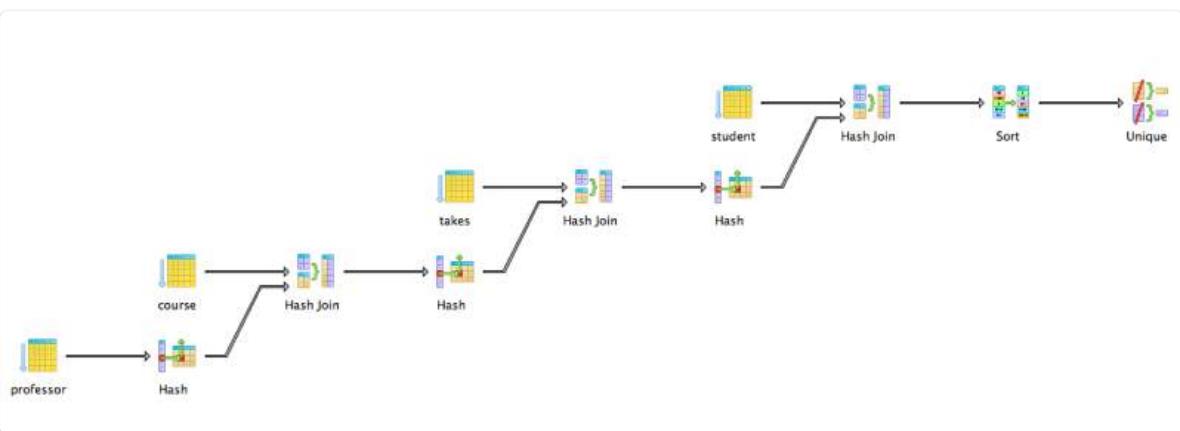
Die zusätzliche ANALYZE-Option sorgt dafür, dass die Anfrage zusätzlich ausgeführt wird.

## ANALYZE

ANALYZE sammelt Statistiken zu den Inhalten der Tabellen.

```
QUERY PLAN
text
Unique (cost=4.61..4.62 rows=2 width=4) (actual time=0.087..0.091 rows=3 loops=1)
-> Sort (cost=4.61..4.62 rows=2 width=4) (actual time=0.087..0.089 rows=4 loops=1)
  Sort Key: s.semester
  Sort Method: quicksort  Memory: 25kB
-> Hash Join (cost=3.47..4.60 rows=2 width=4) (actual time=0.071..0.075 rows=4 loops=1)
  Hash Cond: (s.studid = h.studid)
    -> Seq Scan on student s (cost=0.00..1.08 rows=8 width=8) (actual time=0.004..0.005 rows=8 loops=1)
    -> Hash (cost=3.45..3.45 rows=2 width=4) (actual time=0.054..0.054 rows=4 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 1kB
        -> Hash Join (cost=2.26..3.45 rows=2 width=4) (actual time=0.043..0.053 rows=4 loops=1)
          Hash Cond: (h.courseid = v.courseid)
            -> Seq Scan on takes h (cost=0.00..1.13 rows=13 width=8) (actual time=0.002..0.006 rows=13 loops=1)
            -> Hash (cost=2.25..2.25 rows=1 width=4) (actual time=0.032..0.032 rows=3 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 1kB
                -> Hash Join (cost=1.10..2.25 rows=1 width=4) (actual time=0.022..0.029 rows=3 loops=1)
                  Hash Cond: (v.taughtby = p.empid)
                    -> Seq Scan on course v (cost=0.00..1.10 rows=10 width=8) (actual time=0.001..0.003 rows=10 loops=1)
                    -> Hash (cost=1.09..1.09 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=1)
                      Buckets: 1024  Batches: 1  Memory Usage: 1kB
                        -> Seq Scan on professor p (cost=0.00..1.09 rows=1 width=4) (actual time=0.006..0.010 rows=1 loops=1)
                          Filter: ((name)::text = 'Socrates'::text)
Total runtime: 0.185 ms
```

Also Analyze führt aus und vergleicht Schätzung und tatsächlichen Wert



## Sequential Scans vs. Indexe

Ob ein Index „nützlich“ ist oder nicht, hängt ab von:

- **Wie viele Daten sind für die Anfrage relevant:** Wenn nur wenige Daten benötigt werden, ist ein Index oft nützlich.
- **Größe der Relation:** Bei sehr kleinen Relationen ist ein Sequential Scan oft effizienter als der Overhead eines Index.
- **Eigenschaften des Index:**
  - **Clustered:** Ein Clustering-Index kann sehr effizient sein, da die Daten physisch sortiert sind.
  - **Multiple Columns:** Indexe über mehrere Spalten können bei Anfragen, die diese Spalten kombinieren, sehr nützlich sein.
  - ... (Weitere Eigenschaften wie Unique, Non-Unique, etc.)
- **Welche Algorithmen benötigen die Daten als Input:** Manche Algorithmen profitieren stärker von indexierten Daten als andere.
- ... (Weitere Faktoren)

**Wichtig:** Bis die Anfrageoptimierung perfektioniert wird, ist das **Tuning** die Hauptaufgabe eines Datenbankadministrators (Indexe erstellen, etc.).