IN0008 Grundlagen: Datenbanken

Kapital 01 - Einführung

Datenbasis: gespeicherte Daten, enthält die miteinander in Beziehung stehende Informationseinheiten, die zur Kontrolle & Steuerung eines Aufgabenbereichs notwendig sind

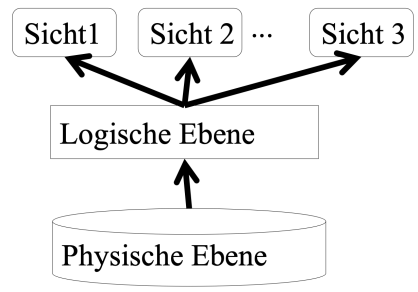
Datenbankverwaltungssystem (DBMS): Gesamtheit der Programme zum Zugriff auf die Datenbasis, zur Kontrolle der Konsistenz & zur Modifikation der Daten

Typische Probleme bei Informationsverarbeitung ohne DBMS

- Redundanz & Inkonsistenz
- Beschränkte Zugriffsmöglichkeiten
- Probleme beim Mehrbenutzerbetrieb
- Verlust von Daten
- Integritätsverletzung
- Sicherheitsprobleme
- hohe Entwicklungskosten für Anwendungsprogramme

Abstraktionsebenen in DBMS

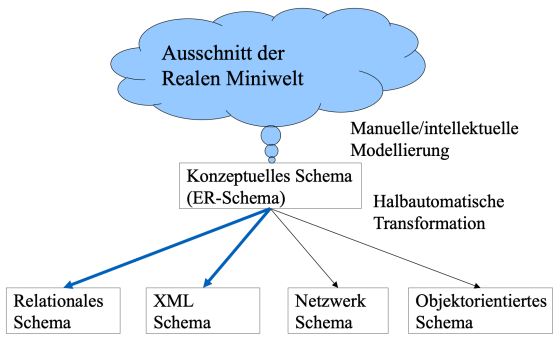
1. **physische Ebene:** wie Daten gespeichert werden
2. **logische Ebene:** in Datenbankschema festgelegt, welche Daten abgespeichert sind
3. **Sichten:** werden Teilmengen der Informationen bereitgestellt, auf Bedürfnisse der Nutzer zugeschnitten



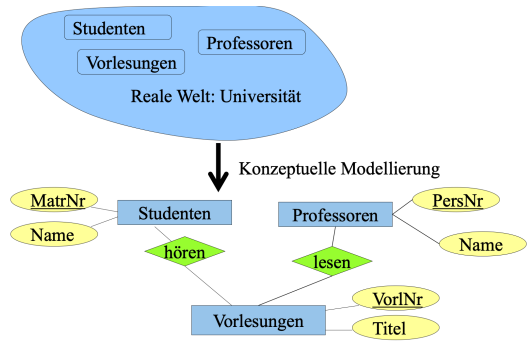
Datenunabhängigkeit

- **physische Unabhängigkeit:** nachträgliche Änderungen dürfen keinen Einfluss auf bereits existierende Anwendungen auf logischen Ebene haben
- **logische Datenunabhängigkeit:** in einer Sichtendefinition kann man keine Änderungen vor den Anwenden verbergen

Datenmodellierung



Beispiel-Modellierung



Das relationale Datenmodell

Studenten	
MatrNr	Name
26120	Fichte
25403	Jonas
...	...

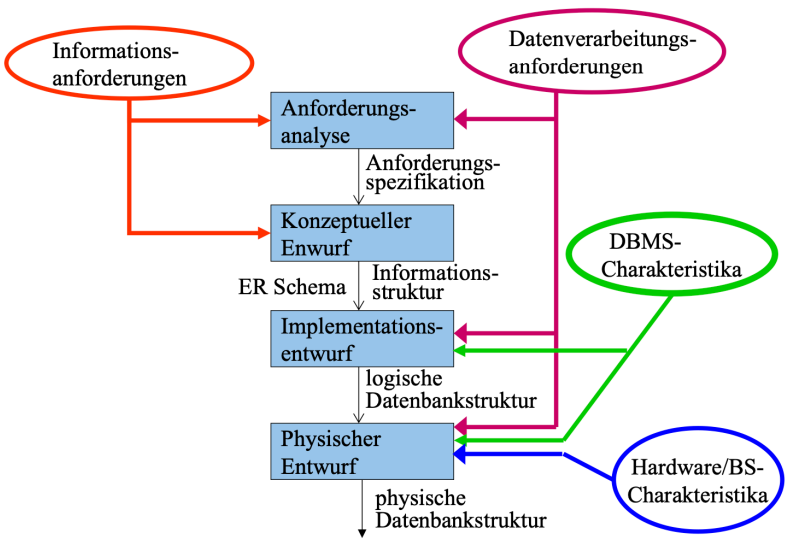
hören	
MatrNr	VorlNr
25403	5022
26120	5001
...	...

Vorlesungen	
VorlNr	Titel
5001	Grundzüge
5022	Glaube und Wissen
...	...

```
Select Name
From Studenten, hören, Vorlesungen
Where Studenten.MatrNr = hören.MatrNr and
      hören.VorlNr = Vorlesungen.VorlNr and
      Vorlesungen.Titel = `Grundzüge´;
```

```
update Vorlesungen
   set Titel = `Grundzüge der Logik´
 where VorlNr = 5001;
```

Kapital 02 - Konzeptuelle Modellierung



Anforderungsanalyse

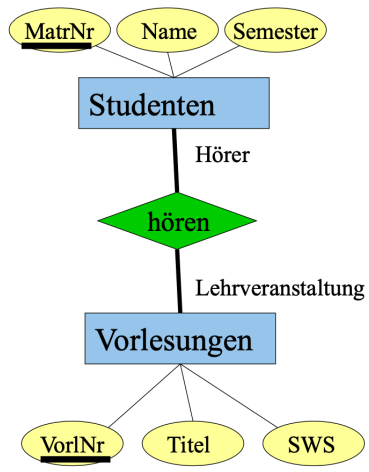
1. Identifikation von Organisationseinheiten - z.B. Studiengänge
2. Identifikation der zu unterstützenden Aufgaben
3. Anforderungs-Sammelplan (Ermittlung der zu befragenden Person)
4. Anforderungs-Sammlung (Phototyp, Gespräche mit Anwenden)
5. Filterung (eindeutige & verständliche Formulierung)
6. Satzklassifikationen (Informationen werden Objekten & Beziehungen zugeordnet)
7. Formalisierung (Pflichtenheft)

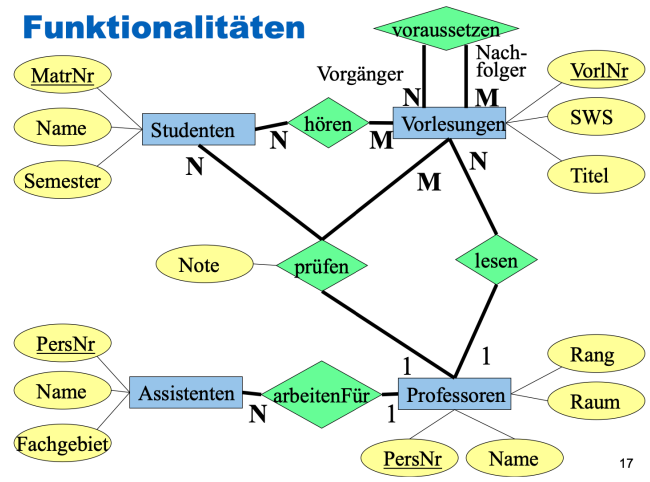
Objektbeschreibung: Typ, Länge, Wertebereich, Anzahl Wiederholungen, Definiertheit, Identifizierend
Beziehungsbeschreibung: Beteiligte Objekte, Attribute der Beziehung, Anzahl
Prozessbeschreibung: Häufigkeit, benötigte Daten, Priorität, zu verarbeitende Datenmenge
Spezifikation (Festlegung des „was")
- **Bestandteile:** Prozessbeschreibungen, Use cases, Benutzerschnittstelle, sonstige Schnittstellen (zu anderen Systemen)
- **Ideale Spezifikation:** eindeutig, vollständig, verständlich, redundanzfrei (in der Realität nicht zu erreichen)

Spezifikation-Methoden: Strukturierte Analyse, Sachbearbeiter-Methode, Objektorientierte Methoden

-> Entity/ Relationship-Modellierung

- **Entity (Gegenstandstyp)**
- **Relationship (Beziehungstyp)**
- **Attribut (Eigenschaft)**
- **Schlüssel (Identifikation)**

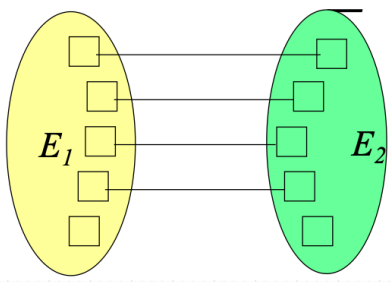




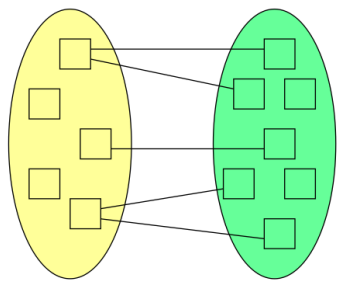
Funktionalitäten $R \subseteq E_1 \times E_2$



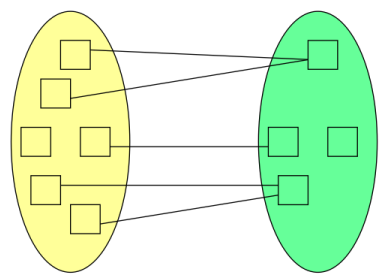
1:1



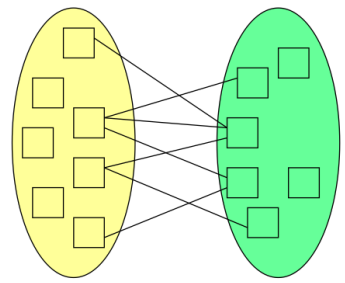
1:N

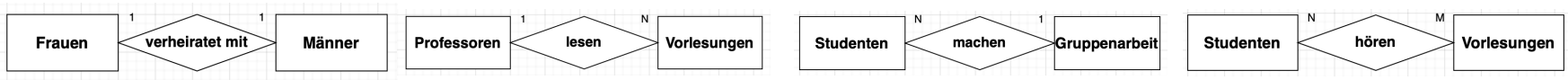


N:1



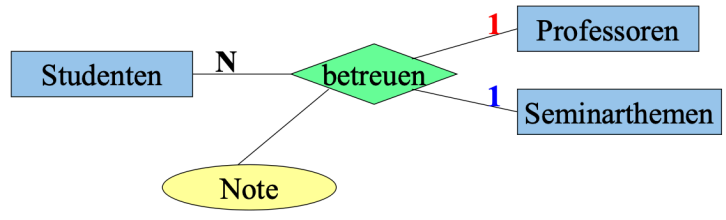
N:M





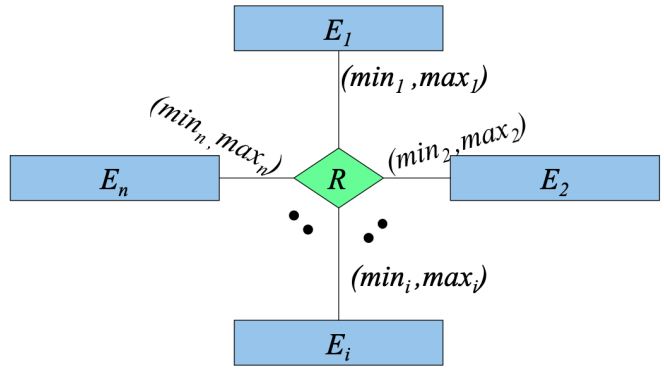
Beispiel-Beziehung: *betreuen*
- **betreuen : Professoren x Studenten -> Seminarthemen**
- **betreuen : Seminarthemen x Studenten -> Professoren**

-> **Dadurch erzwungene Konsistenzbedingungen:** Studenten dürfen bei dem(r)selben Professor/in dasselbe Seminarthema nur einmal bearbeiten

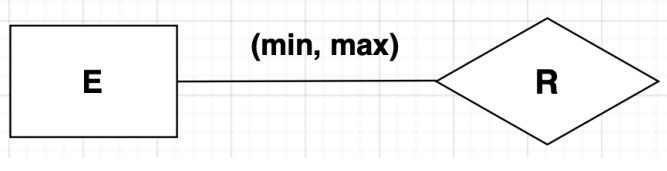


(min, max)-Notation $R \subseteq E_1 \times \ldots \times E_i \times \ldots \times E_n$

$$\begin{aligned} &\textit{Für jedes } e_i \in E_i \textit{ gibt es} \\ &- \textit{Mindestens } min_i \textit{ Tupel der Art } (\ldots, e_i, \ldots) \in R \\ &- \textit{Höchstens } max_i \textit{ viele Tupel der Art } (\ldots, e_i, \ldots) \in R \end{aligned}$$



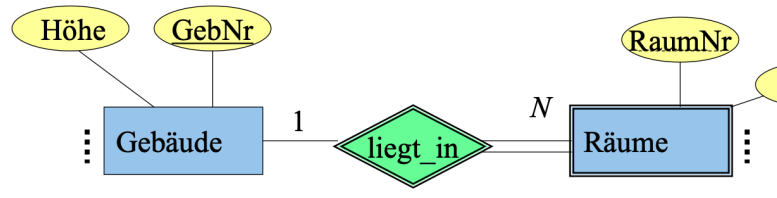
Jedes Entity dieses Typs steht mindestens min-mal & höchstens max-mal in Beziehung R
- Entity, die **gar nicht** an Beziehung teilnehmen -> **min = 0**
- Entities dürfen **beliebig oft** an Beziehung teilnehmen -> **max = ∗**



Existenzabhängige Entitytypen

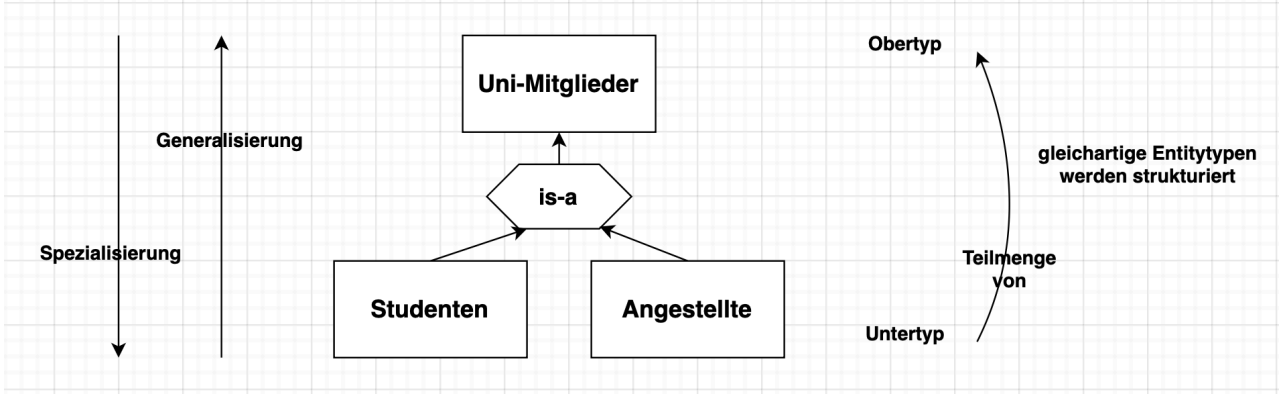
- **Beziehung zwischen "starken" und schwachem Typ ist fast immer 1:N**
  - Existenzabhängige Entity (schwach) oft nur in Kombination mit dem Schlüssel des übergeordneten Entities (stark) identifizierbar
- z.B. RaumNr ist **nur innerhalb eines Gebäudes (stark) eindeutig**

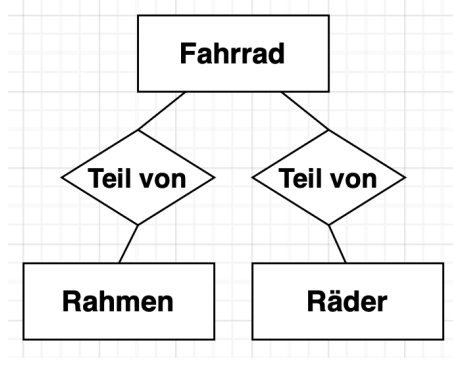
-> Schlüssel: GebNr und RaumNr



Aggregation: unterschiedliche Entitytypen, die in ihrer Gesamtheit einen strukturierten Objekttypen bilden, werden einander zugeordnet

Generalisierung: Attribute können vom Obertyp an Untertyp vererbt werden



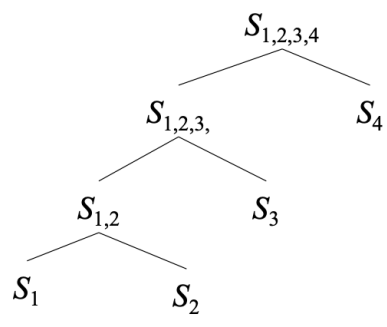


Konsolidierung von Sichtenintegration

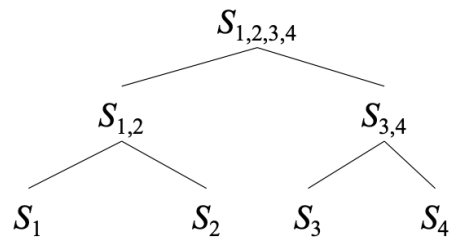
- Redundanzfrei
- Widerspruchsfrei
- Synonyme bereinigt
- Homonyme bereinigt

Möglicher Konsolidierungsbaum

- **maximal hoher Konsolidierungs-baum („links-tief")**



- **minimal hoher Konsolidierungs-baum (Balanciert)**



Kapital 03 - Das Relationale Datenmodell

Grundlagen des relationalen Modells

Relation = Teilmenge des Kreuzprodukts $R \subseteq D_1 \times \ldots \times D_n$

Ausprägung = aktueller Zustand der Datenbasis **Tuple = eine Zeile in einer Relation, hat Attribute**

Schlüssel = minimale Menge von Attributen, deren Werte ein Tupel eindeutig identifizieren

- **Primärschlüssel:** gewählte Schlüsselkandidaten, besondere Bedeutung bei Referenzierung für Tupel
- **Fremdschlüssel** (**Primärschlüssel** von einer anderen Tabelle): dienen dazu Tupel aus anderen Relation zu identifizieren, um dadurch die Zuordnung innerhalb der Beziehung R zu modellieren

Die Relationale Darstellung - Relation: {[Attribut: Datentyp, Schlüsselattribut: Datentyp]}

Verfeinern: Besitzen eine Entity und eine Beziehung den gleichen Primärschlüssel, kann die Beziehung als zusätzlicher Teil der Entity modelliert werden -> nur Relation mit gleichen Schlüssel zusammenfassen

VorlNr: {[VorlNr, Titel]} + **lesen**:{[VorlNr, PersNr]} -> **VorlNr**: {[VorlNr, Titel, gelesen von]}

- Beziehung wird Teil der Entität gegenüber der Funktionalität 1, da diese **auf einen eindeutigen Wert**, hier PersNr verweisen (bei 1:1 unproblematisch)
- Dabei sollten Null-Werte (leere Fremdschlüssel) vermieden werden (insbesondere bei 1:1 Beziehungen)

Die relationale Algebra

σ **Selektion: einzelne Tupel mit erfüllten Selektionsprädikant heraussortieren**

π **Pojektion: extrahiert einzelne Attribute (= Spalten)**

$\times$ **Kreuzprodukt (Kreuzprodukt): bildet alle möglichen Kombinationen**

ρ **Umbenennung (von Relationen)**

-- **Umbenennung von Relationen:** $\rho_{V1}(\text{voraussetzen})$ (alter Name der Relation → Neuer Name der Relation)

-- **Umbenennung von Attributen:** $\rho_{\text{Voraussetzung} \leftarrow \text{Vorgänger}}(\text{voraussetzen})$ (alter Name des Attributes → neuer Name des Attributes; Relation)

$-$ **Mengendifferenz:** $R - S$ **(oder** $R \setminus S$**) Menge der Tupel, die in** R **aber nicht in** S **vorkommen**

$\div$ **Division: Erhält diejenigen Tupel aus Divisor, welche in jeder Kombination mit Attributen aus Dividenden vorkommen (Dividend ÷ Divisor)**

$\cup$ **Vereinigung: zwei Relationen mit gleichem Schema (gleiche Attributnamen und -typen) zu einer Relation zusammenfassen**

$\cap$ **Mengendurchschnitt: erhält alle Tupel, die in beiden Argumentrelationen liegen**

-- Nur auf Argumentrelationen mit gleichem Schema anwendbar

-- Kann durch die Mengendifferenz ausgedrückt werden: R ∩ S = R − (R − S)

R	
M	V
m_1	v_1
m_1	v_2
m_1	v_3
m_2	v_2
m_2	v_3

÷

S
V
v_1
v_2

=

R ÷ S
M
m_1

$\bowtie$ **Join (Verbund = Kreuzprodukt und Selektion)** kommutativ und assoziativ

1. -- **Natürlicher Verbund/ Join: gibt die Schnittmenge an Tupeln der beiden Ursprungsmengen zurück mit gemeinsamen Attributen**
2. -- **Drei-Wege-Join: z.B.: (Studenten** $\bowtie$ **hören)** $\bowtie$ **Vorlesungen**
3. -- **Theta-Join (Allgemeiner Join): äquivalent zu Kreuzprodukt kombiniert mit Selektion** $R \bowtie_\theta S = \sigma_s(R \times S)$ (redundante Spalten werden nicht weggestrichen)
4. $\bowtie$ **linker äußerer Join: Tupel der linken Argumentrelation bleiben im jeden Fall erhalten** LEFT JOIN
5. $\bowtie$ **rechter äußerer Join: Tupel der rechten Argumentrelation bleiben im jeden Fall erhalten** RIGHT JOIN
6. $\bowtie$ **vollständiger äußerer Join: Tupel der beiden Argumentrelation bleiben im jeden Fall erhalten** FULL JOIN
7. $\ltimes$ **Semi-Join von** L **mit** R **: Ergebnis enthält alle Tupel aus** L **in unveränderten Form, die einen potenziellen Join-Partner in** R **haben**
8. $\ltimes$ **Semi-Join von** R **mit** L **: Ergebnis enthält alle Tupel aus** R **in unveränderten Form, die einen potenziellen Join-Partner in** L **haben**
9. $\triangleright$ **Anti-Semi Join von** L **mit** R **: erhält jeweils nur die Einträge aus** L**, die keinen Join-Partner gefunden haben**

Gruppierung und Aggregation

- Anzahl der Elemente der einzelnen Gruppe zählen

$\gamma_{\text{gelesenVon};\textbf{count}(*),\textbf{sum}(\text{SWS})}(\text{Vorlesungen})$ (Aggregation wird pro Gruppe durchgeführt; Gruppierungsattribute)

$\gamma_{\text{gelesenVon};\textbf{count}(*),\textbf{sum}(\text{SWS})}(\text{Vorlesungen})$		
gelesenVon	count(*)	sum(SWS)
2125	3	10
2126	3	8
2133	1	2
2134	1	2
2137	2	8

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

1.

R⋈S												
R − S				R ∩ S				S − R			
A_1	A_2	…	A_m	B_1	B_2	…	B_k	C_1	C_2	…	C_n
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

2.

(Studenten ⋈ hören) ⋈ Vorlesungen						
MatrNr	Name	Semester	VorlNr	Titel	SWS	gelesenVon
26120	Fichte	10	5001	Grundzüge	4	2137
27550	Jonas	12	5022	Glaube und Wissen	2	2134
28106	Carnap	3	4052	Wissenschftstheorie	3	2126
…	…	…	…	…	…	…

$R(A_1, \ldots, A_n)$ $S(B_1, \ldots, B_m)$

3.

$R \bowtie_\theta S$							
R				S			
A_1	A_2	…	A_n	B_1	B_2	…	B_m
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

7.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋉

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat		
A	B	C
a_1	b_1	c_1

8.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋊

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat		
C	D	E
c_1	d_1	e_1

1.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

4.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟕

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

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

5.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟖

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

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

6.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟗

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

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

9.

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

▷

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat		
A	B	C
a_2	b_2	c_2

Relationaler Tupelkalkül {t|P(t)}

- mit der Tupelvariable t (**Ergebnisspezifikation**) und dem **Prädikat** P(t), das erfüllt sein muss, damit t in das Ergebnis aufgenommen wird - z.B. C4-Professoren **{p| p ∈ Professoren ∧ p.Rang = 'C4'}**
- **Atome:** s.A Φ t.B oder s.A Φ c (Tupelvariablen s, t; Attributnamen A, B; Konstante c; Vergleichsperator Φ (=, ≠, ≤, ≥ ...)
- **Quantifizierung von Tupelvariablen**
  - **Allquantor** $\forall x \in X(P(x))$**: "für alle ..."** - Die Bedingung P muss für alle x gelten
  - **Existenzquantor** $\exists x \in X(P(x))$: **"es existiert ein ..."** - Die Bedingung P muss für mindestens ein x gelten
- **Sicherheit:** Eine Anfrage ist sicher, wenn das Ergebnis endlich ist
  - **Bedingung:** Es muss Ergebnis ⊆ Domäne der Formel sein (dann ist das Ergebnis automatisch endlich)
  - **Domäne der Formel:** alle Konstanten aus der Formel + alle Attributwerte von Relationen in der Formel
  - **unsichere Beispielanfrage:** **{p| ¬ (p ∈ Professoren)}**

Relationaler Domänenkalkül $\{[v_1, v_2, \ldots, v_n] \mid P(v_1, \ldots, v_n)\}$

- z.B. MatrNr und Namen der Prüflinge von Curie variable m → implizite Joinbedingung

$$\{[m, n] \mid \exists s\,([m, n, s] \in \textbf{Studenten} \wedge \exists v, p, g\,([m, v, p, g] \in \textbf{prüfen} \wedge \exists a, r, b\,([p, a, r, b] \in \textbf{Professoren} \wedge a = \text{'Curie'})))\}$$

- **Sicherheit:** Ergebnis muss endlich sein - wie bei Tupelkalkül
  - **Bedingung 1:** Konstante im Ergebnistupel muss in Domäne von P liegen
  - **Bedingung 2:** $\exists x\,(P(x))$ -> darf nur für $x \in$ Domäne von P erfüllbar sein
  - **Bedingung 3:** $\forall x\,(P(x))$ -> muss für alle $x \notin$ Domäne von P erfüllt sein

relationale Datenbanken Eingaben in SQL zunächst in relationale Algebra übersetzt, da relationale Algebra formal definiert, somit geschiet Optimierung dort

Kapital 04 - Relationale Anfragesprache SQL

SQL-Injektion mit Password "' OR 1=1 --"

Datentypen

relationales Datenbanksystem: Postgres, Hyper, Umbra, MSSQL, DB2, Oracle, MySQL, MariaDB, ...

- **character** (Exact n), **char** (Excat n) -> **festgelegt Länge** (kürzer als n dann mit Leerzeichen aufgefüllt)
- **character varying** (max n), **varchar** (max n) -> **variable Länge** (kürzer als n, wird nicht aufgefüllt)
- **numeric** (p,s) -> p = the **total number of digits** in the value; s = number of digits **after the decimal point**
- **blob** oder **raw** für sehr große binäre Daten
- **clob** für sehr große String-Attribute
- **date** für Datumsangaben -> oft in Form "YYYY-MM-DD"
- **xml** für XML-Dokumente

● Anlegen von Tabellen/ definieren

```
create table Professoren
                (PersNr integer not null,
                 Name varchar (30) not null,
                 Rang character (2) );
```

● Einfügen

```
insert into hören
       select MatrNr, VorlNr
       from Studenten, Vorlesungen
       where Titel= `Logik' ;

insert into Studenten (MatrNr, Name)
       values (28121, `Archimedes');
```

● einfache Abfragen

SELECT attribut1, attribut2,... $\Pi_{attribut1,attribut2,\ldots}$
FROM rel1, rel2,... $rel1 \times rel2 \times \cdots$
WHERE Bedingung; $\sigma_{Bedingung}$

SELECT attribut **FROM** relation R $\rho_R(relation)$

SELECT attribut **AS** a **FROM** relation $\rho_{a \leftarrow attribut}(relation)$
neu Name neu Name

● Löschen von Tupeln

```
delete Studenten
       where Semester > 13;
```

● Verändern von Tupeln

```
update Studenten
       set Semester= Semester + 1;
```

Änderungsoperation abhängig von Reihenfolge der Tupel

● Duplikateliminierung

SELECT DISTINCT attribut **FROM** relation

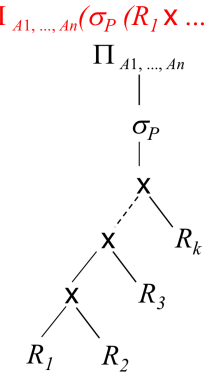
● Sortierung (**ASC** aufsteigend**; DESC** absteigend)

SELECT attr1, attr2
FROM relation
ORDER BY attr1 **DESC,** attr2 **ASC**

Kanonische Übersetzung in die relationale Algebra

SELECT $A_1, \ldots, A_n$
FROM $R_1, \ldots, R_k$ ->
WHERE P;

$\Pi_{A1,\ldots,An}(\sigma_P\,(R_1 \times \ldots \times R_k))$



UNION eliminiert Duplikate -> aufwändig

UNION ALL eliminiert Duplikate nicht -> effizienter, aber kann zu Endlosschleifen führen

- **Mengenoperationen**

∪ **Vereinigung:** (**SELECT** name **FROM** assistenten) **UNION** (**SELECT** name **FROM** professoren)

∩ **Schnitt:** (**SELECT** name **FROM** assistenten) **INTERSECT** (**SELECT** name **FROM** professoren)

\ Differenz: (**SELECT** name **FROM** assistenten) **MINUS** (**SELECT** name **FROM** professoren)

Mengendifferenz: Alle Elemente, die in der linken Seite enthalten sind, aber nicht in der rechten

- **Existenzquantor EXISTS**

SELECT attribute
FROM relation
WHERE NOT EXISTS (subquery);

- **Mengenvergleich**

SELECT attribute
FROM relation
WHERE attribute **IN** (subquery / set of values);

-> Sowohl **EXISTS** als auch **IN** können mit **NOT** kombiniert werden **Anti-Join = not exists = not in**

- **Gruppierung & Aggregatfunktionen - Aggregatfunktionen: avg, max, min, count, sum**

SELECT attr1, attr2, sum(attr3)
FROM relation
optional: **WHERE** …
GROUP BY attr1, attr2
optional: **HAVING** bedingung

Beispiel:

SELECT gelesenVon, sum(sws)
FROM vorlesungen
GROUP BY gelesenVon
HAVING count(*) > 1

-> Alle **nicht-aggregierten** Attribute, die **SELECT**-Klausel enthalten sind, müssen in **GROUP BY** vorkommen

Aggregation in WHERE verboten - z.B. WHERE count(*) >= 1

- **Geschachtelte Anfragen**

Grundsätzlich - In einer Anfrage können noch andere Anfragen enthalten sein

- in **SELECT:** **SELECT** matrnr, note, (**SELECT** s.name **FROM** studenten s **WHERE** s.matrnr = p.matrnr) AS name **FROM** pruefen p
- In **FROM**: **SELECT** * **FROM** (**SELECT** s.matrnr, s.name **FROM** studenten s) **WHERE** matrnr < 2700
- in **WHERE**: **SELECT** * **FROM** pruefen **WHERE** note < (**SELECT** avg(note) **FROM** pruefen)

- **Korrelierte Unteranfragen**

SELECT p.Name
FROM professoren p
WHERE NOT EXISTS (**SELECT** *
FROM vorlesungen v
WHERE v.gelesenVon = p.persNr);

Problem: p.persNr ist für jedes Tupel in p anders -> die Unteranfrage **muss für jedes Tupel neu ausgeführt** werden

- **Unkorrelierte Formulierung**

SELECT p.Name
FROM professoren p
WHERE p.persNr **NOT IN** (**SELECT** v.gelesenVon
FROM vorlesungen v);

Vorteil: Unteranfrage **unabhängig** von p -> muss nur **einmal ausgewertet** werden

- **Join**

SELECT *
FROM rel1
[CROSS | INNER | LEFT/RIGHT/FULL OUTER] JOIN
rel2 **ON** rel1.A = rel2.B

Achtung
In SQL: **Keine Semi- und Anti-Joins** müssen umgeschrieben werden
A × B (CROSS JOIN) -> hier **kein ON** (es gibt keine Join-Bedingung)

- **Entschachtelung korrelierter Unteranfragen durch Join**

SELECT a.*
FROM Assistenten a
WHERE EXISTS
(**SELECT** p.*
FROM Professoren p
WHERE a.Boss = p.PersNr **and** p.GebDatum>a.GebDatum);

->

SELECT a.*
FROM Assistenten a, Professoren p
WHERE a.Boss=p.PersNr **AND** p.GebDatum > a.GebDatum;

- **Casting der Integer zu Decimal** **cast** (h.AnzProVorl **as decimal(6,2)**) / g.GesamtAnz **as** Marktanteil

precision after decimal pointer garantieren: count * 1.000 ≡ cast (count as decimal (10,3))

- **WITH -> Anfragen mit Zwischenergebnissen vereinfachen**

CTE: Common Table Expression

Zwischenergebnisse: CTE 1

```
WITH studenten_3_semester AS (
        SELECT * FROM studenten WHERE semester = 3),
    noten_3_semester AS (
        SELECT s.*, p.note
        FROM studenten_3_semester s, pruefen p
        WHERE s.matrnr = p.matrnr)
```

CTE 2

select all students from CTE 1 and their corresponding grades from "prüfen" table

→

Eigentliche Anfrage:

```
SELECT name, avg(note)
FROM noten_3_semester
GROUP BY name
```

calculates the average grade for each of those groups

group all the rows with the same "name" value into single rows in the result set, one for each unique name

$$\forall t(P(t)) = \forall t(A \to \exists t) = \neg\exists t(\neg(A \to \exists t)) = \neg\exists t(\neg(\neg A \vee \exists t)) = \neg\exists t(A \wedge \neg\exists t)$$

常用于寻找满足"alle" "jede"条件的数据行 where not existists (A and not exists ...)

- **Allquantifizierung durch Umformen**

Elimination von ⇒ und ∀: $A \Rightarrow B \equiv (\neg A \vee B)$ und $\forall X(P(x)) \equiv \neg\exists x(\neg P(x))$

Beispiel

-> **ALL** in SQL kann nur Werte vergleichen, nicht allgemein Bedingungen prüfen -> kein vollwertiger Allquantor

äquivalent < alle Studenten, die alle vierstündigen Vorlesungen hören
es existiert keine vierstündigen Vorlesungen, für die es in der Relation "hören" keinen Eintrag für den Studenten gibt

- **Alternative: Allquantifizierung durch count-Aggregation**

- mithilfe einer count-Aggregation zählen, wie viele Tuple die Bedingung erfüllen und wie viele es gibt

```
SELECT h.matrNr
FROM hoeren h
GROUP BY h.matrNr
HAVING count(*) = (SELECT count(*) FROM vorlesungen)
```

die Anzahl der qualifizierte Tupel der Relation mit der Anzahl der aller möglichen qualifizierten Tupel vergleichen

-> Entsprechende korrekte „SQL-Übersetzung"

{[h.matrNr] | h ∈ hoeren ∧ ∀v ∈ Vorlesungen (∃h2 ∈ hoeren(h2.vorlNr = v.vorlNr))}

- **Nullwerte** - unbekannte Wert („Lücken" in der Datenbank)
  - **Entstehen** im Zuge der Anfrageauswertung mit fehlende Informationen- z.B. outer joins
  - **Führen** manchmal zu überraschende Anfrageergebnisse
  - **arithmetische Ausdrücke:** x + null = null, y * null = null
  - **SQL-Ergebniswerte:** true, false und **unknown** (-> wenn mindestens eines ihrer Argumente **null** ist)
    - Im Ergebnis landen nur die Tupel, bei denen die WHERE-Bedingung zu true auswertet -> insbesondere werden Tupel, für die die Bedingung zu unknown auswertet, nicht ins Ergebnis aufgenommen -> Tupel mit null-Werten gehen verloren
  - **SELECT COALESCE** (val1, val2, ..., val_n): return the first non-null value in a list of arguments
  - **In logischen Ausdrücken**

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

- **Spezielle Sprachkonstrukte ("syntaktischer Zucker")**
  - **AND: SELECT * FROM** Studenten **WHERE** Semester > = 1 **AND** Semester < = 4;
  - **BETWEEN ... AND...: SELECT * FROM** Studenten **WHERE** Semester **BETWEEN** 1 **AND** 4**;**
  - **IN(...): SELECT * FROM** Studenten **WHERE** Semester **in (**1,2,3,4**);**

- **String-Vergleiche - LIKE mit Platzhalter**
  - **"%"** steht für **beliebig viele (oder gar kein) Zeichen**
  - **"_"** steht für **genau ein Zeichen**

z.B. **SELECT * FROM** Studenten **WHERE** Name **like** `T%eophrastos`;

- **Das case-Konstrukt** (-> Die **erste** qualifizierende **WHEN**-Klausel wird ausgeführt)

CASE WHEN bedingung1 **THEN** ausführung1 **WHEN** bedingung2 **THEN** ausführung1 ... **ELSE** ausführung_n **END**

SELECT UNIQUE gibt es nicht

Rekursion: Definition eines Ergebnisses, in der das Ergebnis selbst vorkommt

```
WITH RECURSIVE TransVorl (Vorg, Nachf) AS (
        SELECT Vorgänger, Nachfolger
        FROM voraussetzen
        WHERE Nachfolger = 5259
        UNION/ UNION ALL
        SELECT t.Vorg, v.Nachfolger
        FROM TransVorl t, voraussetzen v
        WHERE t.Nachf= v.Vorgänger)
)
SELECT * FROM TransVorl
```

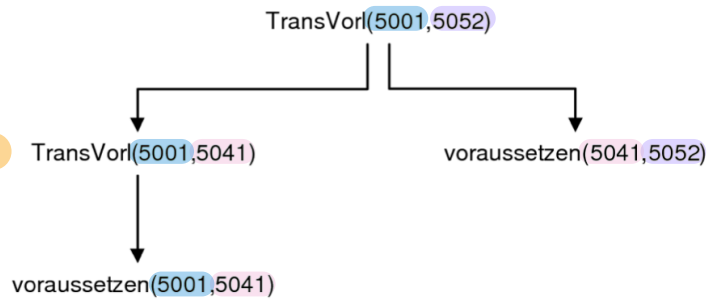
Rekursionsbasis

Rekursionsschritt

Transitive Hülle der Relation voraussetzen

TransVorl(V,N): voraussetzen(V,N)

TransVorl(V,N): TransVorl(V,Z), voraussetzen(Z,N)



Sichten: Oft benutzte Anfragen permanent als Sicht gespeichert

- **Initialisierung: CREATE VIEW** view_name **AS** (subquery)
- **Verwendung: SELECT * FROM** view_name
- **Verändern/ überschrieben: CREATE OR REPLACE VIEW** view_name **AS** (subquery)

Sichten zur Modellierung von Generalisierung

- **Untertypen als Sicht**

1. Physische Relation für den **Obertyp** und seine Attribute: **CREATE TABLE** obertyp_name (attr1, attr2, …)
2. Physische Relation für die **Untertypen** und spezifische Attribute: **CREATE TABLE** untertyp_daten (attr1, attr2, …)
3. Sichten für die **Untertypen: CREATE VIEW** untertyp_name **AS** (select … from … where …)
   - mit einem Join die physischen Relationen des Ober- und Untertyps verknüpfen

- **Obertypen als Sicht**

1. Physische Relation für die **Untertypen** und alle ihre Attributen: **CREATE TABLE** untertyp_name (attr1, attr2, …)
2. Sicht für den **Obertyp: CREATE VIEW** obertyp_name **AS** (select … from …) **UNION** (select … from …) **UNION** (…)
   - Relationen der Untertypen mit union vereinigt - nur die Attribute des Obertyps in der select-Klausel auftauchen

Änderbarkeit von Sichten

- Nur eine Basisrelation
- Der **Primärschlüssel** muss vorhanden
- keine Aggregatfunktion, Gruppierung oder Duplikateleminierung enthalten

Embedded SQL

- Oft will man SQL-Anfragen in seine C/ C++ Programme einbauen
- Embedded SQL als Möglichkeit, im Programmcode spezielle „SQL-Abschnitte" zu markieren
  - die dann ausgeführt werden können und das Ergebnis in Variablen speichern und wie gewohnt verwendet
- Details hängen von der Programmiersprache bzw. verwendeten DBMS ab

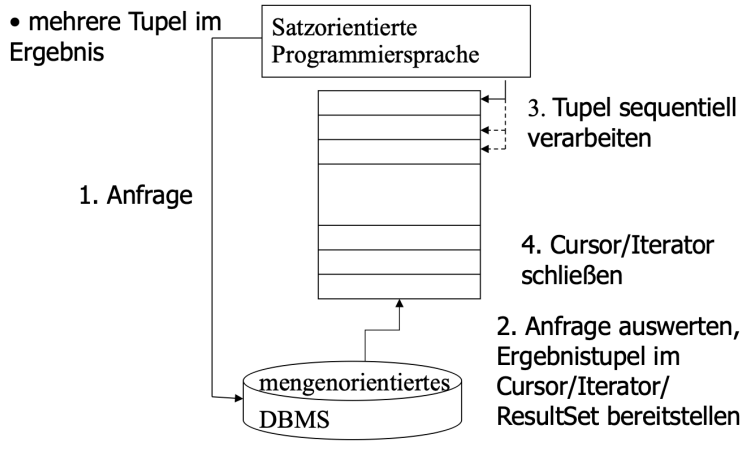
Cursor-Schnittstelle in SQL

```
exec sql declare c4profs cursor for
                select Name, Raum
                from Professoren
                where Rang='C4';
exec sql open c4profs;
exec sql fetch c4profs into :pname, :praum;
exec sql close c4profs;
```

PreparedStatement verhindert SQL-Injection-Angriffe

und bei wiederholten Anfragen leistungsfähiger

Anfragen in Anwendungsprogrammen



Kapital 05 - Datenintegrität

- **Integitätsbedingungen:** Schlüssel, Beziehungskardinalitäten, Attributdomänen, Inklusion bei Generalisierung
- **statische Integritätsbedingungen:** Bedingungen an den Zustand der Datenbasis
- **dynamische Integritätsbedingungen:** Bedingungen an Zustandsübergänge
- **Temporale Daten in SQL:** System-versioniert, Anwendung-versioniert

Referentielle Integrität: Fremdschlüssel müssen auf existierende Tupel verweisen oder einen Nullwert enthalten

Referentielle Integrität in SQL

- **Kandidatenschlüssel: unique** (zusätzliche Schlüssel, aber nicht Primärschlüssel z.B. Raum)
- **Primärschlüssel: primary key** (müssen Wert haben, nicht NULL)
- **Fremdschlüssel: foreign key** (können auch NULL sein)

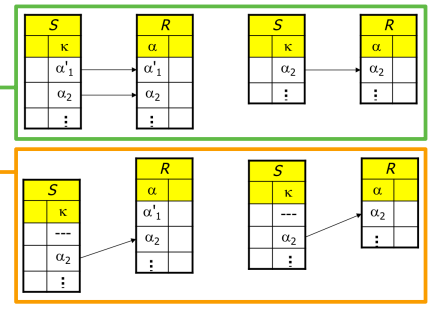
create table R (α **integer primary key**, ...);

create table S (..., $\varkappa$ **integer references** R);

create table prüfen (
MatrNr **integer references** Studenten **on delete cascade**,
VorlNr **integer references** Vorlesungen,
foreign key (PersNr) **references** Professoren **on delete set null**,
Note **numeric**(2,1) **check** (Note **between** 0.7 **and** 5.0));

Änderung von referenzierten Daten

1. **Default (wenn nichts drin steht):** Zurückweisen der Änderungsoperation
2. **Propagieren der Änderungen: [on delete | on update] cascade**
   - Änderung wird auch in referenzierter Tabelle S übernommen
3. **Verweise auf Nullwert setzen: [on delete | on update] set null**



Einfache statische Integritätsbedingungen

- **Wertebereichseinschränkungen: check** attr1 **between** grenz_1 **and** grenz_2
- **Aufzählungstypen: check** attr2 **in** (wert1, wert2, …)

Komplexe Konsistenzbedingungen

- **constraint** name **check** (condition) -> kann Unteranfragen (select … from … where …) beinhalten
- Bei jeder Änderung und Einfügung wird die **check-Klausel** ausgewertet (nur wenn der check **true** ergibt)

Trigger

- automatische Ausführung der bestimmten Routinen, wenn Einträge in Datenbank geändert oder gelöscht
- Verhindern falsche/ sinnlose Werte einzutragen
- Routine ausgeführt, falls **when**-Klausel erfüllt

Auslöser-Definition

create trigger <name>
[before | after] [delete | insert | update of <attr>] on <table>
referencing old as <old_name> **new as** <neu_name>
for each row
when (bedingung)
<routine>

Prozedur-Definition

set semester = **case**
when semester **is null then** 1
when semester < 1 **then** 1
else semester
end;

Temporale Daten

- Tupel können eine Gültigkeitsdauer zugewiesen bekommen
- In Anfragen wird dann automatisch die **aktuelle/ explizit angegebene system_time** mit dem Zeitraum verglichen, für das ein Tupel gültig ist
- Tupel mit falschem Zeitraum werden ignoriert (tauchen nicht im Ergebnis auf)

create table mindestlohn
(LandID **integer primary key**,
Lohn **numeric**(2,1) **not null**,
Beginn **date not null generated always as row start**,
Ende **date not null generated always as row end**,
period for system_time (Beginn, Ende)

```
system_time as of date('2011.04.01')
system_time between ... and ...
```

Contains: Überprüft, ob ein Zeitintervall vollständig innerhalb eines anderen liegt
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 **CONTAINS** Zeitintervall2;
Precedes: Überprüft, ob ein Zeitintervall vor einem anderen endet
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 ENDS BEFORE Zeitintervall2 STARTS;
Succeeds: Überprüft, ob ein Zeitintervall nach einem anderen beginnt
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 STARTS AFTER Zeitintervall2 ENDS;
Immediately Precedes: Überprüft, ob ein Zeitintervall unmittelbar vor einem anderen endet, ohne Zeitlücke
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 ENDS IMMEDIATELY BEFORE Zeitintervall2 STARTS;
Immediately Succeeds: Überprüft, ob ein Zeitintervall unmittelbar nach einem anderen beginnt, ohne Zeitlücke
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 STARTS IMMEDIATELY AFTER Zeitintervall2 ENDS;
Overlaps: Überprüft, ob zwei Zeitintervalle sich überschneiden
SELECT * **FROM** Tabelle **WHERE** Zeitintervall1 **OVERLAPS** Zeitintervall2;

Kapital 06 - Relationale Entwurfstheorie

= konzeptionelle Feinabstimmung des erstellten relationalen Schemas auf Grundlage formaler Methoden, Qualität zu bewerten und zu verbessern (indem Redundanz minimiert und Konsistenzbedingungen eingehalten)

Wenn A ein Superschüssel ist, dann auch jede Menge, die A enthält, ein Superschlüssel

Funktionale Abhängigkeiten (FD)

Kandidatenschlüssel ist die minimale Untermenge eines Superschlüssels

- **Definition:** Sei R = {A_1, A_2, … } das Schema einer Relation, R ihre Ausprägung
  Dann gilt für $\alpha \subseteq R$, $\beta \subseteq R$: $\alpha \rightarrow \beta$ **genau dann wenn** ∀ r, s ∈ R mit r.α = s.α —> r.β = s.β
- **Schlüssel:** Menge von Attributen, die ein Tupel eindeutig identifizieren
  - **Super-Schlüssel** $\alpha \subseteq R$, falls $\alpha \rightarrow R$ (α bestimmt alle anderen Attributwerte innerhalb der Relation R)
  - **Kandidaten-Schlüssel** $\alpha \subseteq R$, falls $\alpha \rightarrow\!\!\!\cdot\, R$ (R ist voll funktional abhängig von α) (z.B. Primärschlüssel)
- β ist **voll funktional abhängig** von α (Schreibweise $\alpha \rightarrow\!\!\!\cdot\, \beta$), genau dann wenn
  1) $\alpha \rightarrow \beta$ gilt; **2)** $\alpha \rightarrow \beta$ und α kann nicht mehr verkleinert (d.h. keines der Attribute in α weggelassen) werden

∅ -> A bedeutet: A unabhängig von anderen Attribute, immer den selben Wert hat

Bestimmung funktionaler Abhängigkeiten

Armstrong-Axiome zur FDs-Herleitung (sei α, β, γ, $\delta \subseteq R$)

- **Reflexivität:** wenn $\beta \subseteq \alpha$ gilt ⇒ $\alpha \rightarrow \beta$
- **Verstärkung:** Wenn $\alpha \rightarrow \beta$ gilt, dann auch $\alpha\gamma \rightarrow \beta\gamma$
- **Transitivität:** Wenn $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ gelten, dann gilt auch $\alpha \rightarrow \gamma$

Bonusregeln

- **Vereinigungsregel:** Wenn $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$ gelten, dann auch $\alpha \rightarrow \beta\gamma$
- **Dekompositionsregel:** Wenn $\alpha \rightarrow \beta\gamma$ gilt, dann auch $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$
- **Pseudotransitivitätsregel:** Wenn $\alpha \rightarrow \beta$ und $\beta\gamma \rightarrow \delta$, dann gilt auch $\alpha\gamma \rightarrow \delta$

Bestimmung der Attributhülle a+ von Menge F

- **Eingabe:** F (Menge von funktionalen Abhängigkeiten) und a (Menge von Attributen)
- **Prozess 1 - Initialisierung:** Setze Ergebnismenge = a
- **Prozess 2 - Iteration** (**solange Änderungen** an Ergebnismenge gibt): Für jede funktionale Abhängigkeit b → g in F überprüfen, ob b eine **Teilmenge** von Ergebnismenge ist. Wenn ja, dann g zu Ergebnismenge **hinzufügen**
- **Ausgabe:** Attributhülle a+ (endgültige Ergebnismenge, nachdem keine weiteren Attribute hinzugefügt werden)

Kanonische Überdeckung Fc - kleinstmögliche Menge an FDs mit folgenden Eigenschaften:

- **Fc ≡ F bzw. Fc+ = F+** (FC bestimmt die **gleichen Attribute** wie ursprüngliche Menge)
- **Keine der FDs enthält überflüssige Attribute**
- **Jede linke Seite** einer FD in Fc ist **einzigartig** (sonst Vereinigungsregel anwenden)

Bestimmen kanonische Überdeckung Fc

1. **Linksreduktion** auf allen FDs $\alpha \rightarrow \beta \in F$ durchführen: Wenn es auf der linken Seite ein Attribut gibt, das man weglassen kann (d.h. auch ohne das Attribut kann ganz β erzeugt werden), wird es entfernt
2. **Rechtsreduktion** auf alle verbliebene FDs $\alpha \rightarrow \beta \in F$ durchführen: Attribute auf der rechten Seite, die schon durch andere FDs erzeugt werden, können weggelassen werden
3. **Entfernen** die entstandene FDs in Form $\alpha \rightarrow \varnothing$
4. FDs mit gleicher linker Seite **zusammenfassen**: $\alpha \rightarrow \beta_1$, … , $\alpha \rightarrow \beta_n$ wird $\alpha \rightarrow (\beta_1 \cup \ldots \cup \beta_n)$

Die kanonische Überdeckung von R ist normalerweise nicht eindeutig, nur eindeutig wenn die Linksreduktion und Rechtsreduktion immer zum selben Endergebnis führen (egal in welcher Reihenfolge man sie ausführt)

Zerlegung (Dekomposition) von Relationen:

Relationenschema R in Relationenschema R_1 , … , R_n zerlegt

- **Verlustlosigkeit: 1)** Alle Informationen aus der ursprünglichen Relation sind in den neuen **wiederzufinden**
  2) wenn $R = R_1 \bowtie R_2$ gilt; **3) hinreichende Bedingung:** wenn $R_1 \cap R_2 \rightarrow R_1$ oder $R_1 \cap R_2 \rightarrow R_2$ gilt
- **Abhängigkeitserhaltung:** Alle FDs aus der Ursprungsrelation sind auf neue Relationen **übertragbar**

Anomalien bei Verletzung der Korrektheitskriterien

- **Update-Anomalien** - führen zu erhöhtem Speicherbedarf und Leistungseinbußen bei Änderungen (Redundanz)
- **Einfüge-Anomalien** - führen zu großen Mengen an Null-Werten
- **Löschanomalien** - können zu ungewolltem Datenverlust führen

Normalform

höhere Normalform -> weniger Anomalien bei insert und update

- Höhere Normalform hat eine bessere Datenstruktur (reduzierter Datenredundanz und erhöhender Integrität)
- Höhere Normalform erfüllt Kriterien der niedrigeren Normalformen - Schema in 3NF automatisch in 2NF und 1NF

- **Erste Normalform (1NF)**: alle Attribute haben atomare Werte (Attributwerte dürfen nicht weiter zerlegbar sein)
- **Zweite Normalform (2NF):** jedes Nicht-Schlüsselattribut voll (d.h. nicht nur von Teilen) funktional abhängig von jedem Kandidatenschlüssel der Relation **falls irgendeine Attribute nicht in FDs enthalten, dann 2NF automatisch verletzt**
- **Dritte Normalform (3NF):** Kein Nichtschlüsselattribut hängt transitiv von einem Schlüsselkandidaten ab
  - R ist in 3NF, wenn für jede funktionale Abhängigkeit $\alpha \rightarrow \beta$ mindestens eine von drei Bedingungen gilt
    1. $\beta \subseteq \alpha$ (die FD ist trivial) (rechte Seite β ist Teil der linken Seite α)
    2. β ist Prime-Attribute (d.h. alle Attribute von β sind in einem Kandidatenschlüssel von R enthalten)
    3. α ist Superschlüssel (α bestimmt alle anderen Attributwerte innerhalb der Relation R)
  - **3NF-Verschärfung = Boyce-Codd-Normalform (BCNF)** -> wenn mindestens eine von Bedingungen 1) und 3) gilt
    - Kein Teil eines Kandidatenschlüssels ist funktional abhängig von einem Teil eines anderen Kandidatenschlüssels
      -> Wenn es nur einen einzigen Kandidatenschlüssel gibt, ist die Relation automatisch in BCNF
- **Vierte Normalform (4NF):** alle MVDs α ->-> β müssen **1)** MVD ist trivial oder **2)** α ist ein Superschlüssel von R

Eine Relation R (falls bereits in BCNF) mit nur FDs -> dann automatisch in 4. Normalform

3NF-Synthesealgorithmus: ermittelt zum Relationenschema eine Zerlegung, die folgenden Kriterien erfüllt
1) verlustlos (d.h. alle Informationen bleiben **erhalten**); **2) abhängigkeitserhaltend** (d.h. alle FDs gelten immer noch); **3)** Die Relationen im Ergebnis sind **alle in 3NF**

- **Schritt 1:** Bestimmen **Kanonische Überdeckung** der FDs (Links- und Rechtsreduktion, Entfernen, Zusammenfassen)
- **Schritt 2:** Für jede FD $\alpha \rightarrow \beta$ wird eine Relation $R_\alpha = \alpha \cup \beta$ erstellt, alle passenden FDs werden R_α zugeordnet
- **Schritt 3:** Überprüfen, ob eine der in Schritt 2 entworfenen Relationen einen Kandidatenschlüssel der ursprünglichen Relation enthält. Wenn nicht, wird eine neue Relation erstellt, die nur den (beliebigen) Kandidatenschlüssel enthält
- **Schritt 4:** Eliminiere diejenigen redundante Schemata, die bereits in anderen enthalten sind

BCNF-Dekompositionsalgorithmus: ermittelt zum Relationenschema eine Zerlegung, die folgenden Kriterien erfüllt
1) verlustlos (d.h. alle Informationen bleiben erhalten); **3)** Die Relationen im Ergebnis sind **alle in BCNF**;
2) ABER: die Zerlegung **muss nicht abhängigkeitserhaltend** sein

- **Schritt 1:** Starte mit $Z = \{R\}$
- **Schritt 2:** Solange es ein Relationenschema $R_i \in Z$ gibt, das nicht in BCNF ist: bei BCNF: MVD wird vernachlässigt
  1. Finde eine FD $\alpha \rightarrow \beta$ sodass **1)** $\alpha \cap \beta = \varnothing$ (nicht trivial) und **2)** $\neg(\alpha \rightarrow R_i)$ (α kein Superschlüssel)
  2. Zerlege R_i in $R_{i1} := \alpha \cup \beta$ und $R_{i2} := R_i \setminus \beta$
  3. Entferne R_i aus Z und füge R_{i1} und R_{i2} ein, also $Z_{neu} = (Z_{alt} \setminus \{R_i\}) \cup \{R_{i1}\} \cup \{R_{i2}\}$

4NF-Dekompositionsalgorithmus - analog zu dem von BCNF

- **Schritt 1:** Starte mit $Z = \{R\}$
- **Schritt 2:** Solange es ein Relationenschema $R_i \in Z$ gibt, das nicht in 4NF ist:

bei 4NF: FD und MVD werden betrachtet weil FD ist auch Teil von MVD

  1. Finde eine MVD α ->-> β sodass **1)** $\alpha \cap \beta = \varnothing$ (nicht trivial) und **2)** $\neg(\alpha \rightarrow R_i)$ (α kein Superschlüssel)
  2. Zerlege R_i in $R_{i1} := \alpha \cup \beta$ und $R_{i2} := R_i \setminus \beta$
  3. Entferne R_i aus Z und füge R_{i1} und R_{i2} ein, also $Z_{neu} = (Z_{alt} \setminus \{R_i\}) \cup \{R_{i1}\} \cup \{R_{i2}\}$

wenn wir eine Relation mit einer MVD erstellen, sollen wir jede deren Attribute als Teil des Kandidatenschlüssels anzeigen

Mehrwertige Abhängigkeiten (MVD) FD ⊆ MVD

- **Definition:** Für ein relationales Schema R, mit $\alpha \subseteq R$, $\beta \subseteq R$ und $\gamma = R \setminus (\alpha \cup \beta)$, gilt α ->-> β (β ist mehrwertig abhängig von α, aber β ist unabhängig von sich selber) genau dann wenn
  - für Tupel mit gleichen α-Werten, kann die β-Werte vertauscht werden und die resultierende Tupel auch in R vorhanden
  - d.h. wenn Tupel (α_1, β_1, γ_1) und (α_2, β_2, γ_2) in R enthalten, dann Tupel (α_1, β_2, γ_1) und (α_1, β_1, γ_2) auch in R enthalten
- **Verlustlose Zerlegung** bei MVDs: Eine Zerlegung von R in R_1 und R_2 ist genau dann verlustlos, wenn
  1) $R = R_1 \cup R_2$ **und 2)** mindestens eine der zwei MVDs gilt: $(R_1 \cap R_2)$ ->-> R_1 oder $(R_1 \cap R_2)$ ->-> R_2
- **Jede FD ist auch eine MVD:** Wenn für eine Relation $\alpha \rightarrow \beta$ gilt, dann gilt auch α ->-> β
  - Wenn nur ein einziger β-Wert zu jedem α vorkommt, dann nichts zu tauschen, sodass Bedingung automatisch erfüllt
- **Komplementregel:** Wenn α ->-> β gilt, dann auch immer α ->-> $R \setminus (\alpha \cup \beta)$
  - Wenn die β-Werte sich beliebig mit den γ-Werten kombinieren lassen, dann auch andersherum
- **Eine MVD ist trivial** - wenn eine der folgenden beiden Bedingungen erfüllt ist
  - $\beta \subseteq \alpha$ (dann gilt automatisch auch $\alpha \rightarrow \beta$ und damit α ->-> β)

jede FD ist auch eine MVD

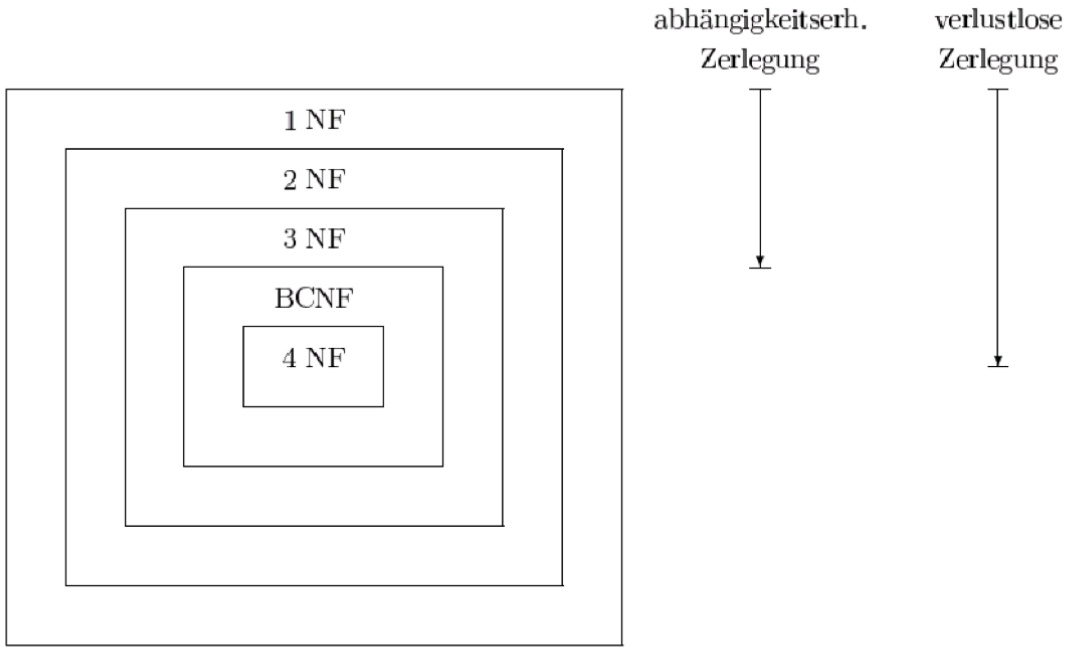
R		
A	B	C
a	b	c
a	bb	cc
a	bb	c
a	b	cc

Inferenzregeln für MVDs

- **Komplement:** wenn $\alpha \twoheadrightarrow \beta$ gilt, dann gilt $\alpha \twoheadrightarrow \mathcal{R} - \beta - \alpha$
- **Mehrwertige Verstärkung:** Sei $\alpha \twoheadrightarrow \beta$ und $\delta \subseteq \gamma$, dann gilt $\gamma\alpha \rightarrow \delta\beta$
- **Mehrwertige Transitivität:** Sei $\alpha \twoheadrightarrow \beta$ und $\beta \twoheadrightarrow \gamma$, dann gilt $\alpha \twoheadrightarrow \gamma - \beta$
- **Verallgemeinerung:** Sei $\alpha \rightarrow \beta$, dann gilt $\alpha \twoheadrightarrow \beta$
- **Koaleszenz:** Sei $\alpha \twoheadrightarrow \beta$ und $\gamma \subseteq \beta$, existiert ein $\delta \subseteq \mathcal{R}$, sodass $\delta \cap \beta = \emptyset$ und $\delta \rightarrow \gamma$, gilt $\alpha \rightarrow \gamma$
- **Mehrwertige Vereinigung:** Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$, dann gilt $\alpha \twoheadrightarrow \gamma\beta$
- **Schnittmenge:** Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$, dann gilt $\alpha \twoheadrightarrow \beta \cap \gamma$
- **Differenz:** Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$, dann gilt $\alpha \twoheadrightarrow \beta - \gamma$ und $\alpha \twoheadrightarrow \gamma - \beta$

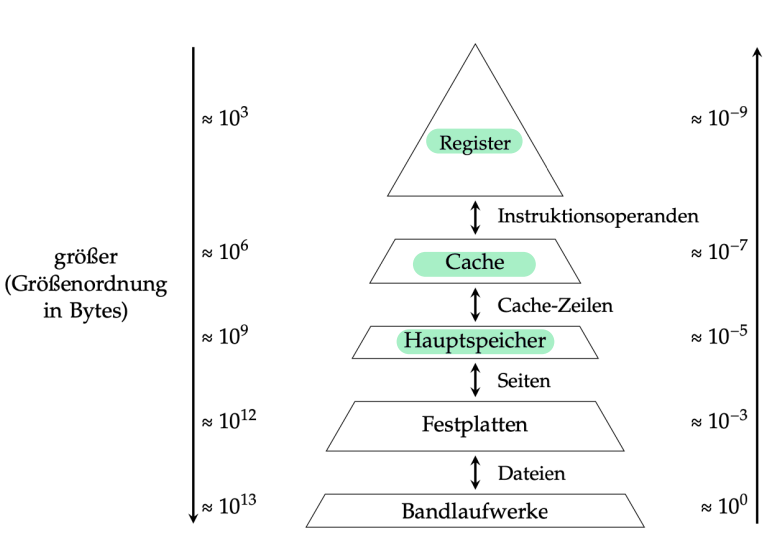
Zusammenfassung

- Verlustlosigkeit ist für alle Zerlegungsalgorithmen in alle Normalformen garantiert
- Abhängigkeitserhaltung kann nur bis zur dritten Normalform garantiert werden



Kapitel 07 - Physische Datenorganisation

Speicherhierarchie (schneller = besser; langsamer = billiger)



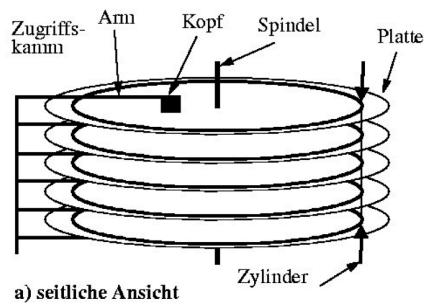
hoch

Durchsatz

Typ	Größenordnung	Zugriffszeit
Register	Byte	1-10 ns
Cache	KB bis wenige MB	10-100 ns
(Primär-) Hauptspeicher	GB bis wenige TB	10-1000 ns
(Sekundär-) Plattenspeicher	TB	10 ms
Archivspeicher	(viele) TB	Sekunden bis Minuten

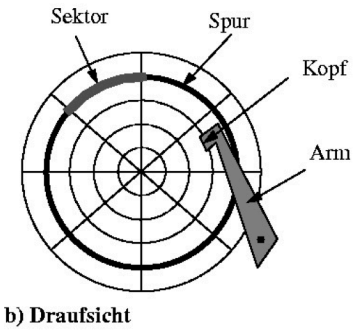
Zugriffslücke: Zugriffe auf Festplatten dauern viel länger als Hauptspeicher etc. (10.000 bis 100.000 mal langsamer!)
Daher sollten alle regelmäßig genutzten Daten immer mindestens auf Hauptspeicher zur Verfügung stehen

Speichermedien - **Magnetplattenspeicher** für Plattenspeicher genutzt



- **Aufbau**
  - **Zylinder** besteht aus mehreren gleichförmig rotierenden Platten und einem **Schreib-/ Lesekopf** pro Platte
  - **Platten** sind eingeteilt in **Spuren**
  - **Spulen** sind eingeteilt in **Sektoren** (kleinste Einheit auf Platte, 1-8 KB)

-> Adressierung über Zylindernummer, Spurnummer und Sektornummer



- **Zugriff**
  1. Kopf muss in die richtige Spule platziert werden: **Seek Time**, ca. 5ms
  2. Platte muss rotieren, bis sich der gesuchte Sektor unterhalb des Kopfes befindet: **Latenzzeit**, ca. 3ms (1/2 Plattenumdrehung)
  3. **Transfer** von der Platte zum Hauptspeicher ca. 15 MB/s

Random IO: jedes Mal Arm neu positioniert

Chained IO: einmal positioniert und nimmt Informationen nacheinander von Platte (schneller)

RAID (Redundant Arrays of Independent Disks)

Problem: Von Festplatten zu lesen ist langsam, von mehreren mithilfe Disk-Array-Controller gleichzeitig zu lesen schneller; Wenn das System ausfällt (z.B. Festplatte geht kaputt) gehen Daten verloren

Lösung: Redundant Arrays of Independent Disks (RAID)

- Redundante Platten mit Replikationen der Daten und zusätzlichen Error-Correcting-Codes (ECCs)
- **Stripingbreite** = Platten-Anzahl; **Stripinggranulität** = Datenblöcke-Größe

RAID 0 (Striping): Datenblöcke gleichmäßig auf mehrere Platten verteilt - „kein echtes RAID"

✅ Zugriffe schneller: Wenn A1 – A4 geladen werden soll geht das hier doppelt so schnell

❌ Datenverlust, wenn eine der Platten ausfällt

RAID 1 (Mirroring): Datenblöcke werden auf allen Platten komplett gespeichert -> mehrere Kopien

- Kann von irgendeiner Platte lesen, aber müssen auf allen Platten Kopien schreiben

✅ Hohe Datensicherheit (Kein Verlust, wenn eine Platte ausfällt); von mehreren Platten gleichzeitig lesen -> gleich schnell

❌ allerdings auch hoher Speicherbedarf

sodass man verwendet hier keine Festplatten mit unterschiedlichen Geschwindigkeiten

RAID 0+1 (Striping + Mirroring): Kombination von RAID 0 & RAID1 (erst als RAID 0 verteilt und als RAID 1 gespiegelt)

✅ noch höhere Bandbreite möglich (Wie RAID 1)

❌ Doppelter Speicherbedarf (Wie RAID 1)

RAID 0 -> vervielfacht sich die Gesamtkapazität
RAID 1 -> Gesamtkapazität = Kapazität von einer der RAID1-Arrays, unabhängig von deren Anzahl, da die sind nur Kopien voneinander, verbessert Performanz aber erhöht Gesamtkapazität nicht

RAID 2 (Striping auf Bit-Ebene): Daten werden in Bits/ Bytes aufgeteilt; zusätzliche Platte speichert HammingCode zur Fehlererkennung (Praktisch nicht eingesetzt - da Platten sowieso schon Fehlererkennungscodes verwalten)

✅ Eigentlich keine Festplatten speichern ohnehin Fehlererkennungscodes)

❌ Zusätzliche Platten, die keine Information sondern nur Fehlererkennungscodes speichern

Alle Platten müssen synchron arbeiten

RAID 3 (Striping auf Bit-Ebene mit zusätzliche Platte für Paritätsinfo) **Alle Platten müssen synchron arbeiten**

- Daten werden in Bits/ Bytes aufgeteilt, zusätzliche Platte speichert Parität (bit-weise xor ⊕) der anderen Platten

✅ Hohe Datensicherheit (Kein Verlust, wenn eine Platte ausfällt)

❌ Lesen eines Blocks erfordert Zugriff auf alle Platten (Verschwendung von Schreib/ Leseköpfen; Alle marschieren synchron) -> immer nur eine Anfrage gleichzeitig

RAID 4 (Striping von Blöcken mit zusätzliche Platte für Paritätsinfo) **Platten müssen nicht synchron arbeiten**

- Daten werden in Blöcke aufgeteilt, zusätzliche Platte speichert Parität (xor ⊕) der anderen Platten
- Neue Berechnung bei Modifikation von Block A zu A': $P'_{A\text{-}D} := P_{A\text{-}D} \oplus A \oplus A'$

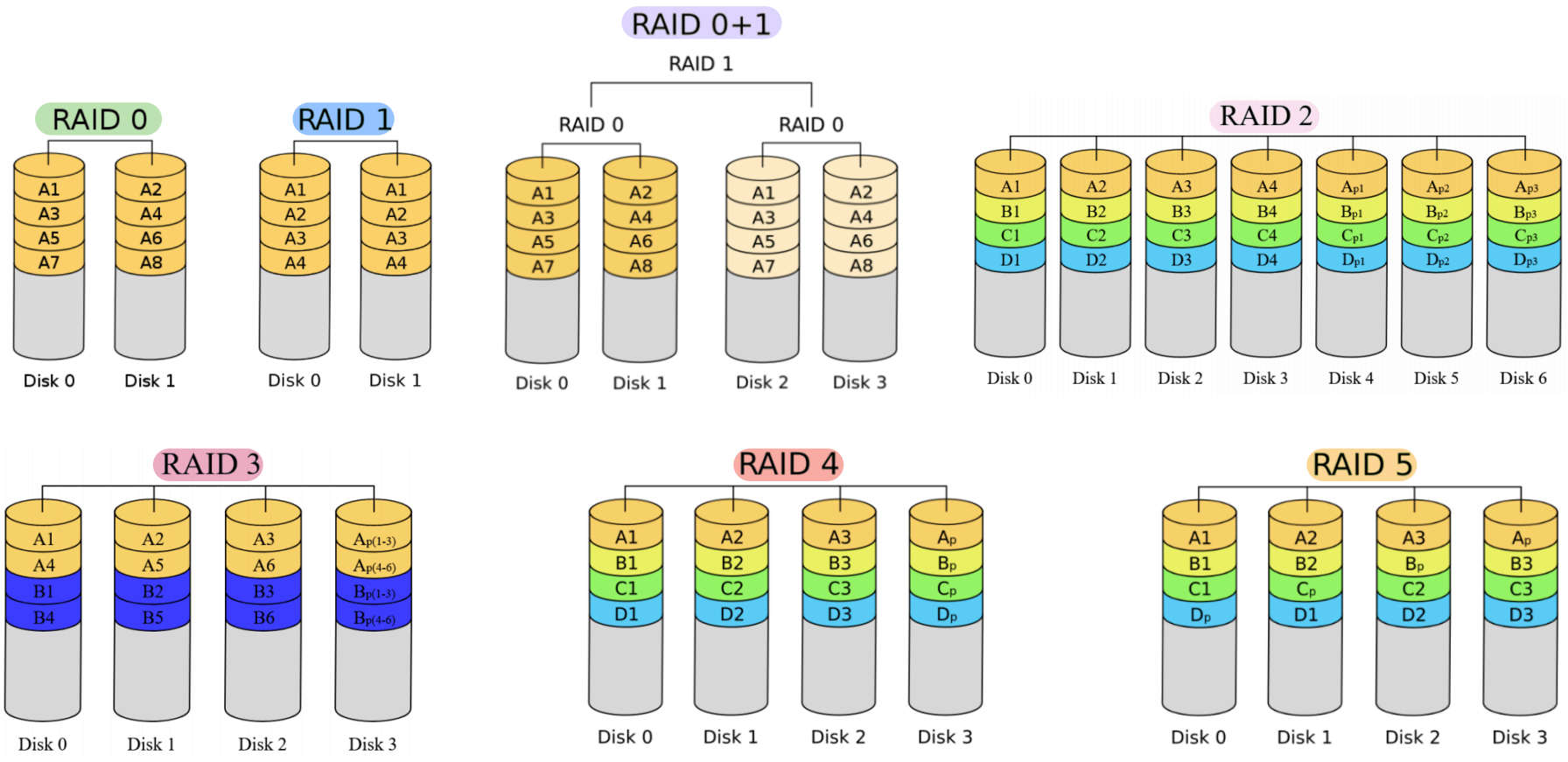
✅ Hohe Datensicherheit (alles gut wenn 1 Platte ausfällt); Bessere Performance (mehrere Anfragen gleichzeitig möglich)

❌ Flaschenhals bei Paritätsplatte: Es kann zwar von mehreren Platten gleichzeitig gelesen werden, aber nur auf eine geschrieben (jede Änderung betrifft auch die Paritätsplatte)

RAID 5 (Striping von Blöcken und Verteilung der Paritätsblöcke)

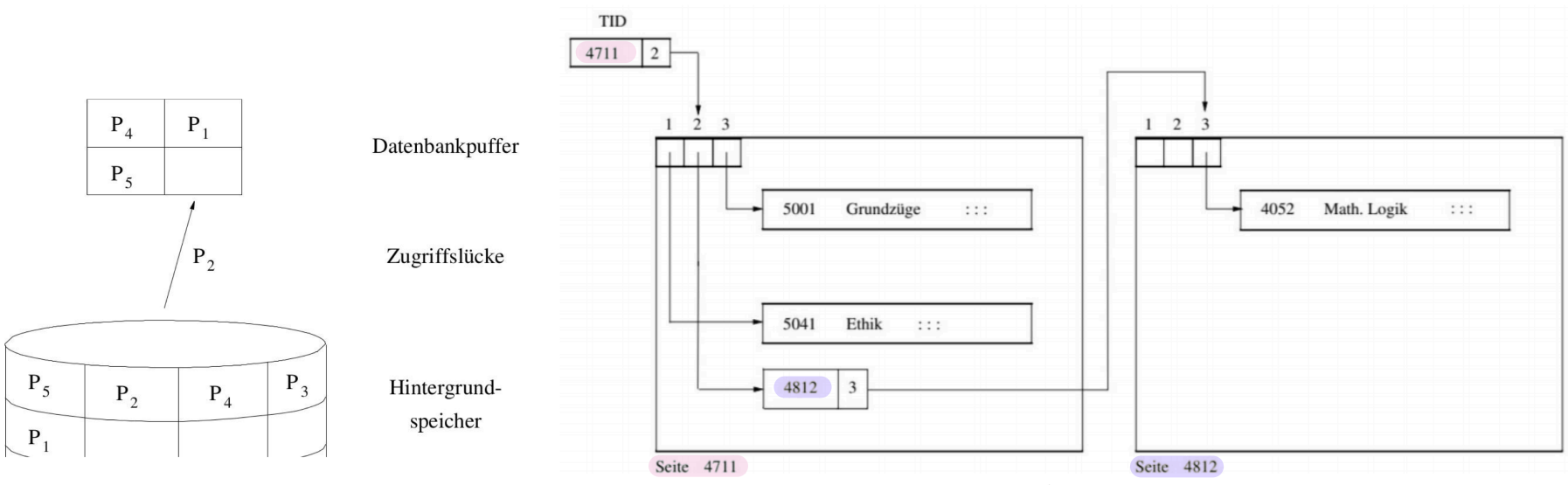
- Daten werden in Blöcke aufgeteilt, Parität (xor ⊕) der anderen Platten wird verteilt gespeichert

✅ Hohe Datensicherheit (Kein Verlust, wenn eine Platte ausfällt); bessere Lastbalancierung (Paritätsplatte hat kein Flaschenhals mehr); Guter Ausgleich zwischen Platzbedarf und Leistungsfähigkeit ; höhere Datendurchsatz



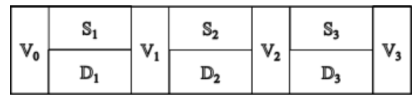
Systempuffer - Speichern von Relationen

- Systemspeicher ist in **Seiten gleicher Größe** (z.B. 4 KB) aufgeteilt
- Tupel werden auf evtl. mehreren Seiten P_n gespeichert und bei **Zugriff in den Puffer** (z.B. Arbeitsspeicher) geladen
- Puffer **voll**, werden sie auf **Hintergrundspeicher** ausgelagert **-> Zugriffslücke!**
- Einzelne Tupel werden durch **Tupel-ID (TID)** referenziert: TID = | *NrSeite* | *NrEintrag* |
- **"Forward":** bei Verschiebung von einer Seite auf eine andere bleibt ein Verweis auf neuen Ort an ursprünglichen Stelle
  - Bei **erneutem** Verschieben wird das Forward **aktualisiert**



B-Baum vom Grad k

- Besteht aus Knoten mit **minimal k** und **maximal 2k** sortierten Einträgen (Ausnahme: Wurzel kann weniger haben)
- für **balancierten** Baum: Jeder Weg von der Wurzel zu einem Blatt hat **die gleiche Länge**
- Innere Knoten mit n Einträgen haben **n + 1 Kinder**
- **1 Seite = 1 Knote** (**Schlüssel S_i** (bzw. $S_1,\dots,S_n$); **Daten D_i**; **Verweisen V_i** (bzw. $V_0,\dots,V_n$) auf n+1 Kindknoten)
  - V_0 verweist auf den Teilbaum mit Schlüsseln kleiner als S_1
  - V_i (i = 1,…,n-1) verweist auf den Teilbaum, dessen Schlüssel zwischen S_i und S_{i+1} liegen
  - V_n verweist auf den Teilbaum mit Schlüssel größer als S_n
  - In Blattknoten sind die Zeiger nicht definiert



✅ **gewährleisten eine effiziente und geordnete Durchquerung der Daten, was insbesondere für Bereichsabfragen und sortierte Zugriffe nützlich**
❌ **weniger effizient bei Punkt- oder exakten Abfragen (im Vergleich zu erweiterbarem Hashing), da Hashing direkteren und schnelleren Zugriff auf einzelne Datenelemente ermöglicht**

● B-Bäume - Einfügen (Insert)

Schritt 1: Suche Wert im Baum
- Existiert schon: Fertig (oder Fehler)
- Existiert nicht: finde Einfügestelle

Schritt 2: Füge Wert ein

Schritt 3: Falls Knoten überfüllt, aufteilen
- Neuer Knoten mit allen Werten rechts vom mittleren Eintrag
- Neuer Knoten mit allen Werten links vom mittleren Eintrag
- Mittleren Eintrag in Vaterknoten einfügen
- Verweise in Vaterknoten korrekt setzen

Schritt 4: Falls Vaterknoten überfüllt
- Ist Vaterknoten Wurzel ? Ja, Mittleren Eintrag von Vaterknoten in eine neue Wurzel anlegen
- Sonst: wiederholen Schritt 3

● B-Bäume - Löschen (Remove)

Schritt 1: Suche Wert im Baum
- Wert im Blattknoten? Entfernen Schlüssel und Daten
- Sonst: Mit nächstgrößerem Eintrag (aus dem ganzen Baum!) ersetzen

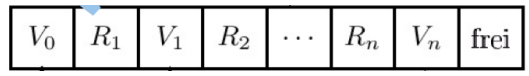
Schritt 2: Falls Unterlauf entsteht (ein Knoten hat jetzt weniger als k Einträge)
- **Durch Ausgleich mit einem Nachbarn mit mehr als k Einträgen**, d.h. der Vaterknoten wird in aktuellen Knoten hineingezogen und ein Eintrag aus einem Nachbarknoten ersetzt den Vaterknoten
- **Durch Verschmelzen von zwei Nachbarknoten** (nur möglich, wenn beide Knoten vorher minimalbelegt waren)

Anmerkung: Wenn in der Aufgabe nicht angegeben ist, ob zuerst der linke oder rechte Nachbar betrachtet werden soll, kann man ihn sich selbst aussuchen

B⁺-Bäume: in inneren knoten nur Referenzschlüssel gespeichert; Daten in verketteten Blattknoten gespeichert

Eigenschaften: grundsätzlich wie bei B-Baum, aber statt Grad k haben B⁺-Bäume (k, k*)

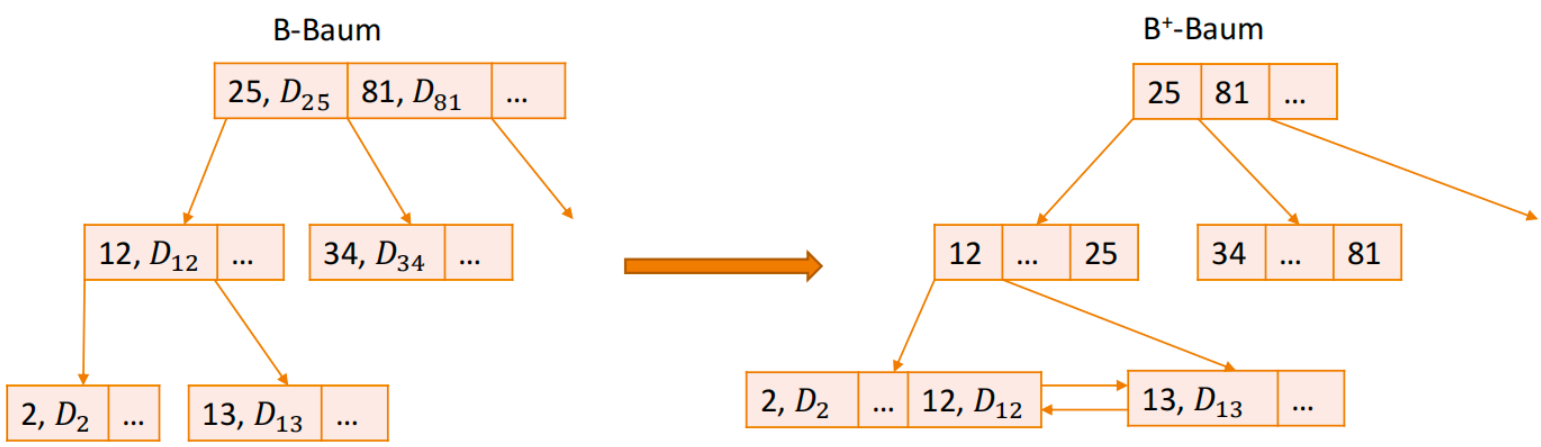
- Innere Knoten haben $k \leq n \leq 2k$ Einträge
- Blattknoten haben $k^* \leq n \leq 2k^*$ Einträge



- **1 innere Knote** (Referenz**schlüssel R_i** (bzw. R_1,…,R_n); **Verweisen V_i** (bzw. V_0,…,V_n) auf n+1 Kindknoten)
  - V_0 verweist auf den Teilbaum mit Schlüsseln kleiner oder gleich R_1
  - V_i (i = 1,…,n-1) verweist auf den Teilbaum, dessen Schlüssel zwischen R_i und R_{i+1} liegen (einschließlich R_{i+1})
  - V_n verweist auf den Teilbaum mit Schlüssel größer als R_n

O(log(n)) und erlaubt Bereichsanfrage -> häufig als standardmäßige Indexstruktur verwendet

-> Zusätzlich: Verketten der Blattknoten ermöglicht schnelle sequentielle Suche



Hashing: Abbilden einer beliebig großen Schlüsselmenge auf ein Bucket mithilfe einer Hashfunktion

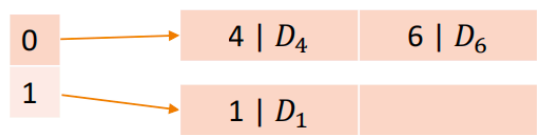
- **Statisches Hashing:** Abbildung von Tupel-ID x direkt auf Seitennummer (Tupel dort gespeichert) - z.B. h(x) = x mod 3
  - Bei **Kollisionen** (Seite bereits voll): Verweis auf andere Seite
  - **Problem:** Kann zu linearer Suche werden -> möglicherweise sehr ineffizient
- **Dynamisches Hashing** übers Directory**:** jedem Tupel hat einen Bitstring (Zeiger auf ein Bucket) durch h(x) zugeordnet
  - **Dynamisches Wachsen und Schrumpfen möglich:** Buckets haben vorher festgelegte Kapazität -> falls voll, ein neuer Bucket wird hinzufügt; falls leer, dies Bucket wird entfernet
  - **Globale Tiefe t** = #Bits d im Bitstring; **Lokale Tiefe t'** = #Bits um Bucket zu finden; p = Bitstring außer Bits d
  - **Bucket-Overflow: t' < t** -> aktuelles Bucket in zwei Buckets aufteilen, t' um 1 erhöhen ; **t' = t** -> t um 1 erhöhen

Es gibt 2^(t−t') Einträge im Verzeichnis, die auf denselben Behälter mit lokaler Tiefe t' verweisen

Nr	Daten	$h(x)$
1	D_1	1000...0
4	D_4	0010...0
6	D_6	0110...0
10	D_{10}	0101...0

Globale Tiefe (Bits im Bitstring)

$t = 1$



Lokale Tiefe (Bits um Bucket zu finden)

$t' = 1$

$t' = 1$

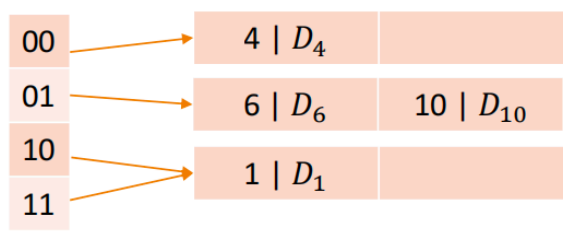
Problem: Bucket voll -> 10 kann nicht eingefügt werden

Lösung: t'=t muss t erhöhen -> dürfen Buckets wieder aufteilen

Nr	Daten	$h(x)$
1	D_1	1000...0
4	D_4	0010...0
6	D_6	0110...0
10	D_{10}	0101...0

Globale Tiefe (Bits im Bitstring)

$t = 2$



Lokale Tiefe (Bits um Bucket zu finden)

$t' = 2$

$t' = 2$

$t' = 1$

✅ **Hashtabellen haben die Laufzeitkomplexität O(1) bei der Suche, wobei B+Bäume O(log n)**
❌ **Auf Hashtabellen können Bereichsanfragen nicht effizient ausgeführt werden**

Mehrdimensionale Datenstrukturen: für wertbasierten Zugriff auf Grundlage mehrerer Attribute (z.B. Gehalt und Alter)

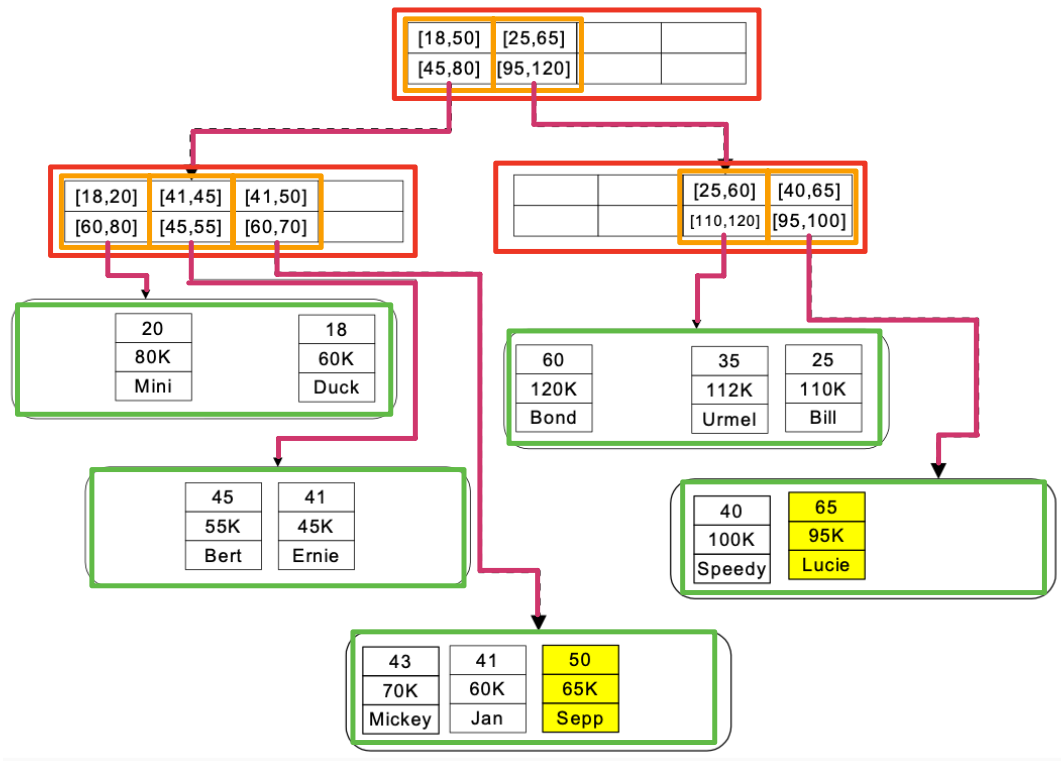
- **Grundlagen mehrdimensionaler Datenstrukturen: Wertebereich $D_0,\ldots, D_{k-1}$** mit k Dimensionen und einem entsprechenden **Datenraum $D = D_0 \times \ldots \times D_{k-1}$**, wobei ein k-dimensionaler Schlüssel einem Punkt p im Datenraum
- **Verschiedene Anfragen möglich**
  - **Exact Match Query:** spezifiziert Suchwert für alle D_i
  - **Partial Match Query:** spezifiziert Suchwert für einige D_i
  - **Range Query:** spezifiziert ein Suchintervall für alle D_i
  - **Partial Range Query:** spezifiziert ein Suchintervall für einige D_i

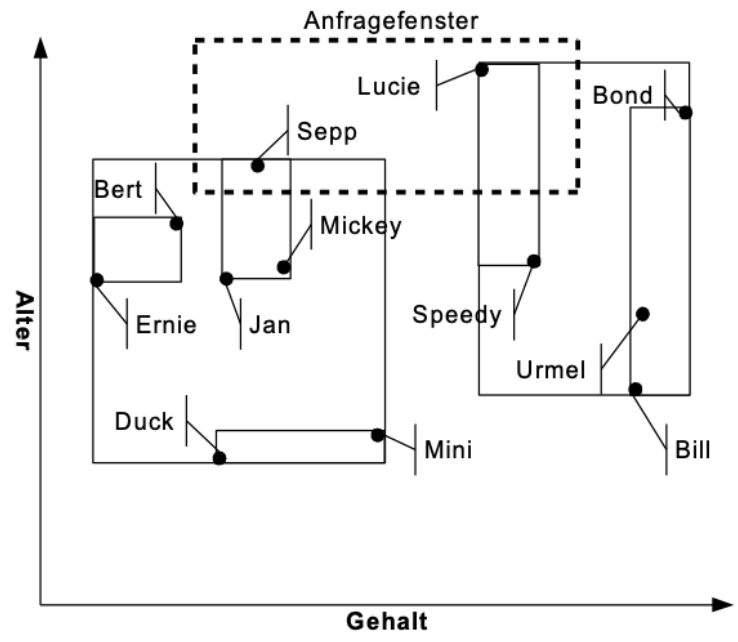
R-Baum: besteht aus unsortierten Blattknoten mit Daten und inneren Knoten mit Bounding Box und einem Verweis auf einen Nachfolger-Inneren-Knoten. Bounding Box beschreibt jeweils den minimalen Raum, in den jeweilige Daten hineinfallen

Overflow (wenn die Kapazität einer Blattknotens überschritten wird)

- **Partitionierung:** Teilen der Bounding Box in zwei möglichst kleine Boxen (Box in Box) mit möglichst wenig Überlappung
- **R-Baum wächst nach oben:** Teilen des Blattknotens und hinzufügen eines Eintrags in den Vorgänger-Inneren-Knoten

Bei Bereichsanfragen: jede Bounding Box wird überprüft, ob sich mit dem Anfragefenster überlappt (ggf. mehrere Wege)

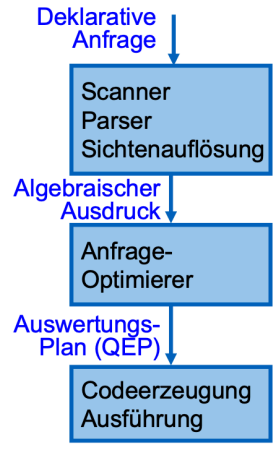




Objektballung: Mittel zur Zugriffsbeschleunigung wobei Daten, die häufig zusammen benötigt werden, auf dem Hintergrundspeicher physisch dicht beieinander gelegt werden, um so die Anzahl der Seitenzugriffe zu minimieren

- immer nur ein Cluster-Index (Primärindex, oft auf dem Primärschlüssel), Daten nach Cluster-Index sortiert

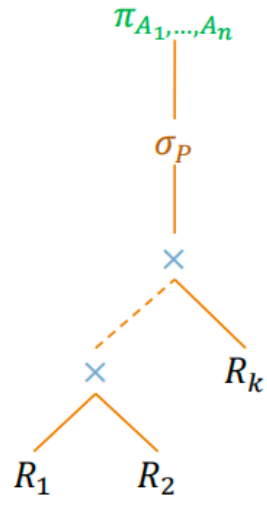
Kapitel 08 - Anfragebearbeitung



Ablauf der Anfrageoptimierung: 1) deklarative Anfrage wird semantisch und syntaktisch analysiert und in einen **algebraischen Ausdruck** aufgelöst; **2) Anfrage-Optimierer** erstellt eine effiziente Implementierung des algebraischen Ausdrucks, den **Auswertungsplan (QEP)**; **3)** Es folgen **Codeerzeugung und Ausführung**

Kanonische Übersetzung

select $A_1,...A_n$
from $R_1, ..., R_k$
where P



Äquivalenzerhaltende Transformationsregeln

1. Aufbrechen von Konjunktionen im Selektionsprädikat: $\sigma_{C1 \wedge C2 \wedge ... \wedge Cn}(R) \equiv \sigma_{C1}(\sigma_{C2}(...(\sigma_{Cn}(R))\ ...))$
2. Selektion σ ist kommutativ: $\sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$
3. π-Kaskaden: Wenn $L_1 \subseteq L_2 \subseteq ... \subseteq L_n$, dann gilt: $\pi_{L1}(\pi_{L2}(...(\pi_{Ln}(R))\ ...)) \equiv \pi_{L1}(R))$
4. Wenn sich σ nur auf Attribute in π bezieht, kann getauscht werden: $\pi_{A1,...,An}(\sigma_C(R)) \equiv \sigma_C(\pi_{A1,...,An}(R))$
5. ×, ∪, ∩ und ⋈ sind kommutativ: $R \bowtie_c S \equiv S \bowtie_c R$
6. Vertauschung von σ mit ⋈, wenn sich σ nur auf Attribute in R bezieht: $\sigma_C(R \bowtie_j S) \equiv \sigma_C(R) \bowtie_j S$
   - 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: $\sigma_C(R \bowtie_j S) \equiv (\sigma_{C1}(R)) \bowtie_j (\sigma_{C2}(S))$
7. Vertauschung π mit ⋈, wenn sich ⋈ nur auf Attribute in π bezieht: $\pi_L(R \bowtie_C S) \equiv (\pi_{A1,...,An}(R)) \bowtie_C (\pi_{B1,...,Bn}(S)))$
8. ×, ∪, ∩ und ⋈ sind jeweils (einzeln betrachtet) assoziativ: $(R \times S) \times T \equiv R \times (S \times T)$
9. σ ist distributiv mit ∪, ∩, − $\sigma_C(R \cup S) \equiv (\sigma_C(R)) \cup (\sigma_C(S))$
10. π ist distributiv mit ∪ $\pi_C(R \cup S) \equiv (\pi_C(R)) \cup (\pi_C(S))$
11. Join-/ Selektionsprädikate können mittels DeMorgan'schen Regeln umgeformt werden
    $\neg (c_1 \wedge c_2) \equiv (\neg c1) \vee (\neg c2)$ $\neg (c_1 \vee c_2) \equiv (\neg c_1) \wedge (\neg c_2)$
12. σ gefolgt von × bezogen auf Attribute aus R und S kann durch ⋈ ersetzt werden: $\sigma_C(R \times S) \equiv R \bowtie_c S$

Logische Optimierung: Anfrage wird zur besseren Übersicht in Baumdarstellung der kanonischen Übersetzung in relationaler Algebra dargestellt und dann optimiert -> 1) Zusammenfassen von Operationen, sodass weniger ausgeführt werden; 2) Vertauschen der Bearbeitungsreihenfolge, sodass Zwischenergebnisse möglichst klein werden

Heuristische Anwendung von Transformationsregeln

1. **Aufspalten der Selektionsprädikate** - Regel 1
2. **Pushing selections:** Selektionen werden möglichst weit "nach unten" propagiert - Regel 2,4,6,9
3. **Kreuzprodukte und Selektion zu Joins zusammenfassen** wenn möglich
4. **Optimierung der Join-Reihenfolge** (Kommutativität und Assoziativität ausnutzen)
5. **Projektionen** werden **eingefügt** und möglichst weit "**nach unten**" propagiert - Regel 3,4,7,10
6. **Operationsfolgen zusammenfassen**, wenn sie in einem "Durchlauf" ausführbar sind

-> Hierbei wird eine möglichst effiziente (nicht unbedingt die optimale) Anfragereihenfolge ermittelt, mit möglichst kleinen Zwischenergebnissen

Alle äquivalente Anfragen haben die selben Ergebnismengen, aber können verschiedene Kosten verursachen

Physische Optimierung

Iteratoren: - ermöglicht, verschiedene Implementierungen gegeneinander auszutauschen
- erlaubt, Zwischenergebnisse zu verwenden ohne sie komplett speichern zu müssen

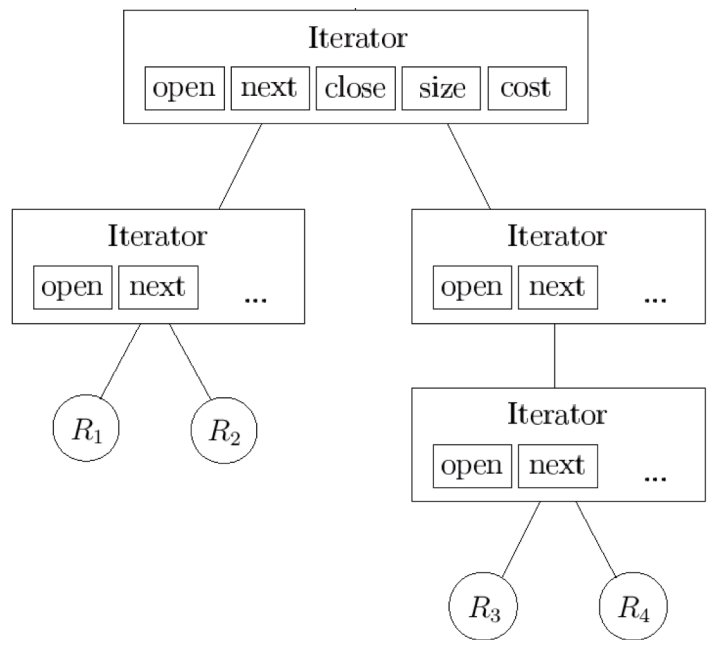
- **`open:`** Initialisierung (Eingabe öffnen, etc.)
- **`next:`** Berechnet das nächste Tupel des Ergebnisses und gibt es zurück
- **`close:`** Schließt Eingabe, gibt Ressourcen frei
- **`cost/ size:`** Gibt (geschätzte) Berechnungskosten bzw. Ergebnisgröße zurück

✅ Eingabe kann Relation, Iterator,… sein; kann mit next immer so viele Tupel angefragt, wie im nächsten Schritt benötigt

Iteratoren - Selektion

• ***Select$_p$/ Scan$_p$*: „Brute Force"**
- **`open:`** Eingabe öffnen
- **`next:`** Rufe so lange Eingabe.next() auf, bis eines die Bedingung p erfüllt -> gib dieses Tupel zurück
- **`close:`** Eingabe schließen

• ***IndexSelect$_p$/ IndexScan$_p$***
- **`open:`** Eingabe öffnen & im Index erstes Tupel suchen, das die Bedingung p erfüllt
- **`next:`** Gib nächstes Tupel zurück, falls es die Bedingung p noch erfüllt
- **`close:`** Schließe Eingabe



Join

• **„Brute Force":** Für jeden Eintrag in R wird S durchsucht

```
foreach r ∈ R
  foreach s ∈ S
    if s.B = r.A then res ≔ res ∪ (r×s)
```

Nested-Loop-Join O(n^2)

• **Iterator – *Nested Loop$_p$***
- **`open:`** Linke Eingabe öffnen
- **`next:`** **1) Rechte Eingabe geschlossen?** Öffnen; **2) Rechts Tuple solange anfordern**, bis Bedingung p erfüllt; **3) Rechte Eingabe erschöpft?** Schließen, nächstes Tuple der linken Eingaben anfordern, start **`next`** neu; **4) Gib Verbund** von aktuellem linken und rechten Tuple **zurück**
- **`close:`** beide Eingabequellen schließen

• **Weitere Transformationsregeln**

$$D \bowtie \sigma_p(T_2) \equiv \sigma_p(D \bowtie T_2)$$
$$D \bowtie (\Gamma_{A;a:f}(T)) \equiv \Gamma_{A \cup \mathcal{A}(D);a:f}(D \bowtie T)$$
$$D \bowtie (\Pi_A(T)) \equiv \Pi_{A \cup \mathcal{A}(D)}(D \bowtie T)$$
$$D \bowtie (T_1 \cup T_2) \equiv (D \bowtie T_1) \cup (D \bowtie T_2)$$
$$D \bowtie (T_1 \cap T_2) \equiv (D \bowtie T_1) \cap (D \bowtie T_2)$$
$$D \bowtie (T_1 \setminus T_2) \equiv (D \bowtie T_1) \setminus (D \bowtie T_2)$$

• **Block nested loop join:** Falls Eingaben zu groß sind, um in den Hauptspeicher zu passen: **1) Unterteilen** in kleinere Blöcke; **2)** Blöcke **einzeln** in den Hauptspeicher **laden** und dann **wie vorher abarbeiten**

✅ Einfach zu implementieren; auch leicht für Theta- und Semi-Joins umsetzen ❌ Quadratische Laufzeit O($|R| \cdot |S|$)

Index-Join: Für jeden Eintrag im anderen Attribut werden passende Werte im Index „nachgeschlagen"
- **Voraussetzung:** Auf einem der Join-Attribute muss ein Index existieren

• **Iterator *IndexJoin$_p$***
- **`open:`** linke Eingabe öffnen; erstes Tuple aus linker Eingabe holen; Joinattributwert im Index nachschlagen
- **`next:`** Falls Index (weiteres) Tupel zum aktuellen Wert liefert: kombinieren & zurückgeben, Sonst: links nächstes Tuple anfordern, Joinattribute wieder im Index nachschlagen
- **`close:`** Eingabe schließen

✅ Schneller als Nested-Loop: O($|R| \cdot \log(|S|)$) ❌ Einen Index zu erstellen ist zu aufwendig -> nicht verwendbar, wenn kein Index existiert; auf Zwischenergebnissen gibt es keine Indexstrukturen

(Sort-) Merge-Join O(n) wenn sortiert, ansonsten O(n log(n))

- **Voraussetzung:** beide Relationen müssen nach den Join-Attributen sortiert sein (sonst vorher sortieren)
- Die Attribute werden dann „Stück für Stück" abgearbeitet

- **Iterator** $MergeJoin_p$
  - `open`: beide Eingaben öffnen; `akt` = linke Eingabe; rechte Eingabe markieren
  - `next`: **1) solange Bedingung p nicht erfüllt:** `akt` = Eingabe mit kleinerem Wert; `akt.next()`; Wert der anderen Eingabe markieren; **2) Linkes und rechtes Tuple verbinden**; **3)** `next` **auf andere Eingabe aufrufen**; **4) Bedingung nicht mehr erfüllt bzw. andere Eingabe erschöpft?** `akt.next()`; falls Wert in `akt` verändert, Wert der anderen Eingabe markieren, sonst andere Eingabe auf Markierung zurücksetzen

    - **intuitive-Version** ($MergeJoin_p$ `next`) - **ABER** funktioniert nicht unbedingt immer !

    - **Beide Pointer zeigen auf den gleichen Wert**
      - Werte verbinden
      - den zuletzt nicht benutzten Pointer bewegen
        - **Wert ändert sich:** Anderen Pointer bewegen
        - **Wert bleibt gleich:** Verbinden & selben Pointer weiterbewegen
    - **Pointer zeigen auf unterschiedliche Werte**
      - Den Pointer, der auf den kleineren Wert zeigt, bewegen
        - **Wert bleibt gleich:** Wenn der Wert in der anderen

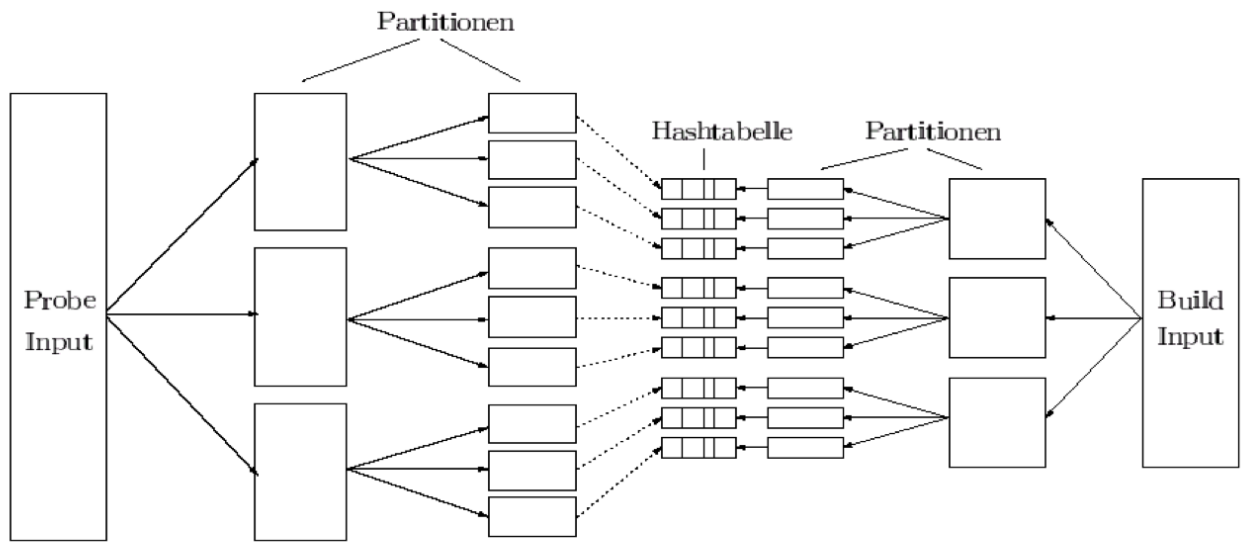
✅ Sehr schnell: O ($|R| + |S|$)

❌ Beide Relationen müssen nach Join-Attribut sortiert sein; Sortieren sehr aufwendig, lohnt sich nicht

Hash-Join: O(n) Relationen S und R werden mit derselben Hashfunktion in Buckets verteilt, für jede Partitionen-Paar soll wiederholen:

1. **Partitionierung:** Die kleinere Relation R (Build Relation) wird mittels einer Hashfunktion so partitioniert, dass jede Partition auf den Hauptspeicher passt (m-1 Seiten werden genutzt). Die Relation S wird mit der gleichen Hashfunktion partitioniert, sodass sich potentielle Join-Partner nun immer in gegenüberliegenden Partitionen befinden.
2. **Build-Phase:** jeweils eine Partitionierung aus R in den Hauptspeicher geladen und die Tupel in einer Hashtabelle angeordnet
3. **Probe-Phase:** Einzelne Blöcke aus der „gegenüberliegenden" Partition werden schrittweise in den Hauptspeicher geladen und Joinpartner in der Hashtabelle gesucht, entsprechende Tupel werden gespeichert.

✅ Sehr schnell: O ($|R| + |S|$); Keine Voraussetzungen



Paralleler Hash Join

- **Schritt 1:** An jeder Station werden mittels Hash-Funktion h1 die jeweiligen Partitionen von R in $R_1,...,R_k$ und S in $S_1,...,S_k$ zerlegt (h1 muss so gewählt werden, dass alle Ri´s aller Stationen in den Hauptspeicher passen)
- **Schritt 2:** Für alle 1 ≤ i <≤ n: Berechne jetzt den Join von R_i mit S_i wie folgt
  1. Wende eine weitere Hash-Funktion h2 an, um R_i auf die Stationen zu verteilen; Sende Tuple t an Station h2(t)
  2. Eintreffende R_i-Tuple werden in die Hash-Tabelle an der jeweiligen Station eingefügt
  3. Sobald alle Tuple aus R_i „verschickt" sind, wird h2 auf S angewendet und Tuple t an Station h2(t) geschickt
  4. Sobald ein S_i-Tupel eintrifft, werden in der R_i-Hashtabelle seine Joinpartner ermittelt

Kostenbasierte Optimierung: 1) alle möglichen Anfrageauswertungspläne erstellen; **2) ihre Kosten bewerten** (Kostenmodelle, Statistiken, Informationen über den verwendeten Rechner und den verfügbaren Speicher); **3) billigste Plan behalten** (nicht unbedingt der Beste aber wahrscheinlich ziemlich gut)

Für einen Join-Baum T sei folgende Kostenfunktion gegeben

$$C_{out}(T) = \begin{cases} 0 & \text{falls } T \text{ eine Basisrelation } R_i \text{ ist} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{falls } T = T_1 \bowtie T_2 \end{cases}$$

Die Kardinalität sei dabei

$$|T| = \begin{cases} |R_i| & \text{falls } T \text{ eine Basisrelation } R_i \text{ ist} \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) |T_1||T_2| & \text{falls } T = T_1 \bowtie T_2 \end{cases}$$

Kardinalität eines Join-Resultats

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

Selektivität $f_{i,j}$ = Anteil an qualifizierten Tupeln einer Operation relativ zur Gesamtanzahl der Tupel

- **Selektion mit Bedingung p :** $\mathrm{sel}_p := \frac{|\sigma_p(R)|}{|R|}$
- **Join von R mit S :** $\mathrm{sel}_{RS} := \frac{|R \rhd\lhd S|}{|R \times S|} = \frac{|R \rhd\lhd S|}{|R| \cdot |S|}$
- **Equi-Join über Fremdschlüssel auf R (bzw. Primärschlüssel A von R):** $\mathrm{sel}_{\mathrm{R.A=S.B}} := \frac{1}{|\mathrm{R}|}$
- **Selektion mit Schlüssel A von R :** $\mathrm{sel}_{\mathrm{R.A=C}} := \frac{1}{|\mathrm{R}|}$
- **falls A ungefähr gleichverteilt (i = Anzahl der Attributwerten):** $\mathrm{sel}_{\mathrm{R \cdot A}} = \mathrm{C} := \frac{1}{\mathrm{i}}$

-> Sonst komplexere Verfahren nötig - z.B. parametrisierten Verteilungen, Histogrammen und Stichproben

Dynamische Programmierung

- **Idee:** basiert auf **Bellmann-Optimalitätskriterium** (Das Gesamtproblem kann **in Subprobleme partitioniert** werden, wobei **die optimale Lösung des Gesamtproblems aus optimalen Lösungen dieser Subprobleme besteht**)
- **Phase 1: Access Root (Aufzählen der Zugriffspläne)**
- **Phase 2: Join Root (Aufzählen der Join-Kombinationen):** Zugriffspläne, für die es äquivalente bessere Pläne gibt werden eliminiert, teurere Pläne mit „interessanten Eigenschaften", z.B. Sortierungen (also nicht mehr äquivalent) werden behalten. Kostenabschätzung wie Kostenfunktion
- **Phase 2: Finish Root (sort, group-by):** Hier wird dann nur noch der billigste Plan behalten

✅ Findet die optimale Lösung; sehr effiziente DP-Algorithmen

❌ exponentielle Laufzeit im worst case; Kostenmodell muss stimmen

BestPlansTable	
Index (S)	Alternative Plans
n, c, m	$(\mathrm{iseek}_{\mathrm{City='Munich'}}(\mathrm{m}) \bowtie \mathrm{iseek}_{\mathrm{From}}(\mathrm{c})) \bowtie \mathrm{iseek}_{\mathrm{City='NY'}}(\mathrm{n})$
c, m	$\mathrm{iseek}_{\mathrm{City='Munich'}}(\mathrm{m}) \bowtie \mathrm{iseek}_{\mathrm{From}}(\mathrm{c})$
n, m	$\mathrm{iseek}_{\mathrm{City='NY'}}(\mathrm{n}) \times \mathrm{iseek}_{\mathrm{City='Munich'}}(\mathrm{m})$
n, c	$\mathrm{iseek}_{\mathrm{City='NY'}}(\mathrm{n}) \bowtie \mathrm{iseek}_{\mathrm{To}}(\mathrm{c})$
Airport m	$\mathrm{scan}(\mathrm{m})$, $\mathrm{iseek}_{\mathrm{City='Munich'}}(\mathrm{m})$, $\mathrm{iscan}_{\mathrm{Code}}(\mathrm{m})$
Connection c	$\mathrm{scan}(\mathrm{c})$, $\mathrm{iseek}_{\mathrm{From}}(\mathrm{c})$, $\mathrm{iseek}_{\mathrm{To}}(\mathrm{c})$
Airport n	$\mathrm{scan}(\mathrm{n})$, $\mathrm{iseek}_{\mathrm{City='NY'}}(\mathrm{n})$, $\mathrm{iscan}_{\mathrm{Code}}(\mathrm{n})$

Ein möglicher Ansatz zur Bestimmung der optimalen Joinreihenfolge ist die Dynamische Programmierung

Kapitel 09 - Transaktionsverwaltung

Transaktion = Bündelung von Operationen, die zusammengehören

- **Recovery -** d.h. Behebung von eingetretenen Fehlersituationen
- **Synchronisation** von mehreren gleichzeitig ablaufenden Transaktionen

Eigenschaften von Transaktionen (ACID)

- **Atomicity (Atomarität):** **Transaktionen** werden entweder komplett oder gar nicht ausgeführt; **Transaktionen** sind die kleinste **unteilbare** Einheit
- **Consistency (Konsistenz):** Daten in der Datenbank sind konsistent bzgl **referentieller Integrität oder anderen Integritätsbedingungen**; Eine **Transaktion** überführt die Datenbank von einem konsistenten in einen anderen konsistenten Zustand
- **Isolation (Isolieren):** Jede **Transaktion** sieht die Datenbank so als **wäre** sie alleine; **Parallel laufende Transaktionen** beeinflussen sich nicht gegenseitig; **Parallel laufende Transaktionen** kommen zum gleichen Ergebnis, als wenn sie sequentiell abgearbeitet worden wären
- **Durability (Dauerhaftigkeit):** Alle erfolgreich abgeschlossene Transaktionen bleiben dauerhaft in der Datenbank

In klassischen Transaktionssystemen

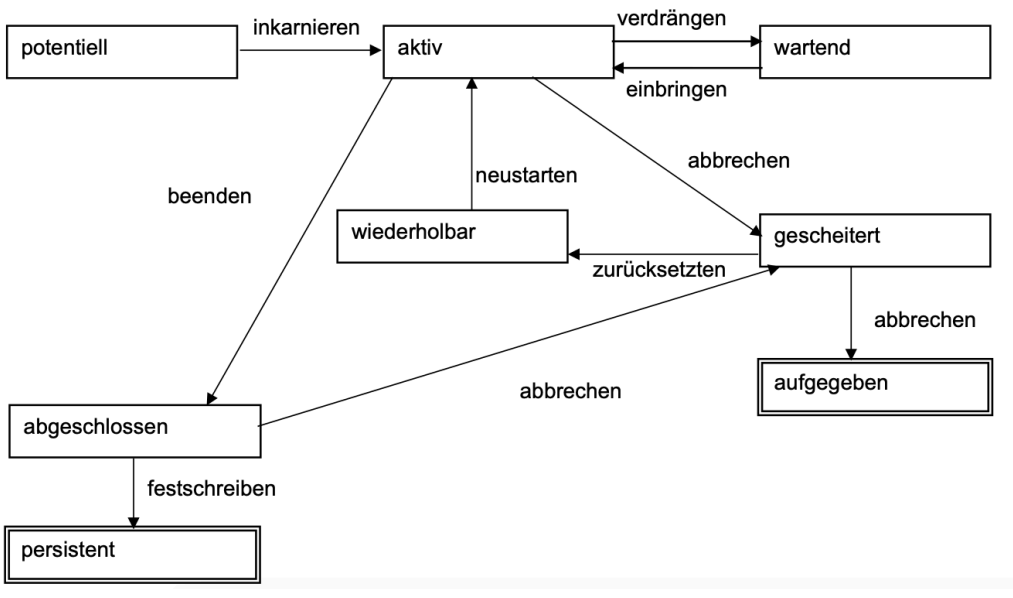
- **begin of transaction (BOT)** = Beginn einer Transaktion wird angezeigt
- **commit** = Transaktion beendet erfolgreich, alle Datenbasis-Änderungen werden festgeschrieben (dauerhaft)
- **abort** = Transaktion beendet erfolglos, alle Datenbasis-Änderungen wird rückgängig in vor-Transaktion-Zustand gemacht

Erweiterte Operationen (zusätzlich in neuen Datenbankanwendungen)

- **define savepoint** = Definieren eines Sicherungspunkts, auf den sich eine noch aktive Transition zurücksetzen lässt
- **backup transition** = Zurücksetzen eine aktive Transition auf den letzten (= zuletzt angelegten) Sicherungspunk

-> Damit Transaktionen im Fehlerfall nicht komplett abgebrochen werden müssen

Zustandsübergangs-Diagramm für Transaktionen

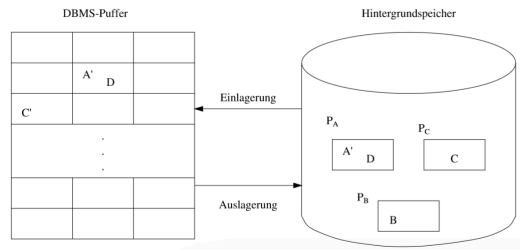


Kapitel 10 - Transaktionsverwaltung

Fehlerklassifikation

1. **Lokale Fehler** in einer noch nicht festgeschriebenen Transaktion (= vor commit)
   - **R1-Recovery:** Änderungen zurücksetzen
2. **Fehler mit Hauptspeicherverlust**
   - **R2-Recovery:** Abgeschlossene Transaktionen müssen erhalten bleiben = **redo**
   - **R3-Recovery:** Nicht abgeschlossene Transaktionen müssen zurückgesetzt werden = **undo**
3. **Fehler mit Hintergrundspeicherverlust**
   - **R4-Recovery:** Durch Backups etc. Datenverlust verhindern (mithilfe einer Archivkopie und einem Log-Archiv mit allen vollzogenen Änderungen)

Speicherhierarchie



- **Erinnerung:** Daten werden im Hintergrundspeicher gespeichert, im Puffer bearbeitet
- **Gefahren**
  - DMBS-Puffer geht verloren -> Änderungen ebenfalls verloren
  - Seiten zu früh zurückgeschrieben -> möglicherweise bleiben manche Änderungen von abgebrochenen Transaktionen auf dem Hintergrundspeicher

- **Ersetzen von Puffer-Seiten**
  - **¬steal:** Die (von noch aktiven Transaktion modifizierte) Seiten können nicht ersetzt werden
  - **steal:** Jede nicht fixierte (= momentan nicht benötigte) Seite kann ersetzt werden
- **Einbringen von Änderungen abgeschlossener Transaktionen**
  - **force:** Änderungen werden zum Transaktionsende auf Hintergrundspeicher geschrieben
  - **¬force:** geänderte Seiten können im Puffer bleiben und erst später zurückgeschrieben

	force	**¬force**
¬steal	Kein Undo Kein Redo	Redo kein Undo
steal	Kein Redo Undo	Redo Undo

Einbringungsstrategien

- **Update in Place (direkte Einbringungsstrategie):** Jeder Seite hat genau einen Speicherplatz im Hintergrundspeicher; Beim Auslagern wird der alte Zustand der Seite überschrieben
- **Twin-Block-Verfahren (indirekte Einbringungsstrategie):** Jeder Seite werden zwei Speicherblöcke im Hintergrundspeicher zugeordnet, wobei nur eine der beiden Blöcke aktuell ist (wird in einem globalen Bit gespeichert); Änderungen werden jeweils nur in die aktuellen Blöcke gespeichert, bei einem Fehler wird auf den zweiten Block zurückgesprungen
- **Schattenspeicherkonzept (indirekte Einbringungsstrategie):** Nur geänderte Seiten werden dupliziert

kleine Sperrgranulate: Transaktionen können auch kleinere Objekte als eine komplette Seite sperren, eine Seite kann also dreckige Daten und festgeschriebene Updates enthalten

Protokollierung von Änderungsoperationen

Struktur der Log-Einträge [LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]

- **LSN #i:** eindeutige Kennung Log-Eintrags (monoton aufsteigend, chronologische Reihenfolge der Protokolleinträge)
- **TransaktionsID T_i:** eindeutige Kennung der (durchgeführt Änderung) Transaktion
- **PageID P_A:** eindeutige Kennung der Seite, die verändert wurde (eine pro Log-Eintrag)
- **Redo:** wie die Änderung nachvollzogen werden kann

  A += 50 (logisch), A = 50 (physisch)
- **Undo:** wie die Änderung rückgängig gemacht werden kann

  A -= 50 (logisch), A = 0 (physisch)
- **PrevLSN #j:** Zeiger auf den vorherigen Log-Eintrag mit LSN #j der Transaktion

Beispiel einer Log-Datei

Schritt	T_1	T_2	Log [LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1,T_1,**BOT**,0]
2.	$r(A,a_1)$		
3.		**BOT**	[#2,T_2,**BOT**,0]
4.		$r(C,c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A,a_1)$		[#3,T_1,P_A,A-=50,A+=50,#1]
7.		$c_2 := c_2 + 100$	
8.		$w(C,c_2)$	[#4,T_2,P_C,C+=100,C-=100,#2]
9.	$r(B,b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B,b_1)$		[#5,T_1,P_B,B+=50,B-=50,#3]
12.	**commit**		[#6,T_1,**commit**,#5]
13.		$r(A,a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A,a_2)$	[#7,T_2,P_A,A-=100,A+=100,#4]
16.		**commit**	[#8,T_2,**commit**,#7]

Logische vs. physische Protokollierung

- **Physische Protokollierung protokolliert Inhalte/ Zustände**
  - **before-image** enthält den Zustand vor Ausführung der Operation
  - **after-image** enthält den Zustand nach Ausführung der Operation
- **Logische Protokollierung protokolliert Aktionen/ Zustandsübergänge**
  - **before-image** wird durch das Ausführen des Undo aus dem afterimage generiert
  - **after-image** wird durch Ausführen des Redo aus dem before-image generiert

-> Dafür muss **beim Wiederanlauf** entschieden werden, ob sich ein before-/ after-image auf dem Hintergrundspeicher befindet. Dazu wird auf jeder Seite die LSN des jüngsten diese Seite betreffenden Log-Eintrags gespeichert

- **LSN der Seite < LSN des Log-Eintrags:** before-image (Änderung wurde noch nicht auf Hintergrundspeicher propagiert)
- **LSN der Seite ≥ LSN des Log-Eintrags:** after-image (Änderung wurde bereits auf den Hintergrundspeicher propagiert)

WAL-Prinzip (Write Ahead Log-Prinzip)

1. Bevor eine Transaktion festgeschrieben (committed) wird, müssen alle zugehörigen Log-Einträge ausgeschrieben werden (notwendig für Redo)
2. Bevor eine modifizierte Seite ausgelagert wird, müssen alle zugehörigen Log-Einträge in die Log-Datei und das Log-Archiv ausgeschrieben werden (nicht nur im Puffer) (notwendig für Undo)

Log-Einträge werden typischerweise doppelt gespeichert

- In eine **Log-Datei** für „schnellen" Zugriff (z.B. auf Plattenspeicher) **für R1, R2 und R3-Recovery**
- In ein **Log-Archiv** für sicheren Zugriff (getrennt, z.B. auf Magnetband) **für R4-Recovery**

Winner vs. Looser

- **Winner (erfolgreicher commit):** bereits abgeschlossene Transaktionen müssen erhalten bleiben (Redo)
- **Loser (noch kein commit):** noch nicht abgeschlossene (= zum Absturz-Zeitpunkt noch aktiv) Transaktionen müssen zurückgesetzt werden (Undo)

Drei Phasen des Wiederanlaufs

1. **Analyse:** Die Winner- und Loser-Transaktionen werden bestimmt
2. **Alle Änderungen (winner und loser) werden in Ausführungsreihenfolge wiederhergestellt (Redo)**
   - LSN der Seite mit LSN des Log-Eintrags vergleichen, falls before-image dann Redo; und die LSN dementsprechend aktualisiert (falls das System beim Wiederanlauf abstürzt)
3. **Undo aller Loser in umgekehrten Ausführungsreihenfolge:** Zu jedem Undo wird ein CLR ins Log geschrieben
   Compensation Log Record (CLR): <LSN', TA, PageID, Redo, PrevLSN, UndoNextLSN>
   Beispiel: <#3', T_1, P_A, A -= 50, #3, #1 > zu Log-Eintrag [#3, T_1, P_A, A += 50, A -= 50, #1]

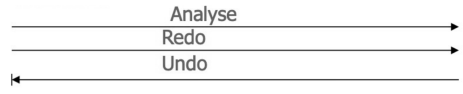
"redo" in <> entspricht eigentlich genau das "undo" in []

Fehlertoleranz des Wiederanlaufs (da Systemabsturz während des Wiederanlaufs möglich)

- Undo- und Redo-Phase müssen **idempotent (= mehrfache Ausführung führt zum selben Ergebnis)** sein
  - **undo(undo(...(undo(a))...)) = undo(a)** -> Undo-Phase-Idempotenz durch CLR im Log
  - **redo(redo(...(redo(a))...)) = redo(a)** -> Redo-Phase-Idempotenz durch Aktualisieren der LSN-Einträge auf Seiten

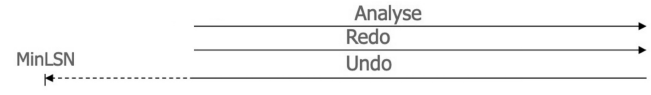
Sicherungspunkte: eine markierte Position im Log, über die man beim Wiederanlauf nicht hinausgehen muss

- **Transaktionskonsistente Sicherungspunkte**



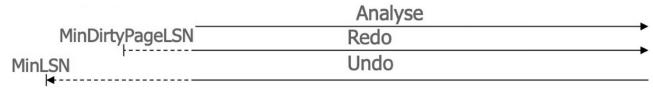
  - Vor dem Anlegen eines Sicherungspunktes müssen alle Transaktionen vollständig ausgeführt und neue Transaktionen werden verzögert; Alle modifizierten Seiten werden ausgelagert; Log-Datei kann wieder von neuem begonnen werden, beim Absturz werden **nur** Informationen **nach dem Sicherungspunkt** benötigt

- **Aktionskonsistente Sicherungspunkte**



  - Vor dem Anlegen eines Sicherungspunktes müssen alle Änderungsoperationen vollständig abgeschlossen werden; Alle modifizierten Seiten werden ausgelagert (keine DirtyPages im Puffer); Beim Absturz werden **Redo** nur bis zum **Sicherungspunkt** benötigt, aber **Undo** bis zur **MinLSN** benötigt

- **Unscharfe (fuzzy) Sicherungspunkte**



  - Modifizierte Seiten werden nicht ausgeschrieben, nur ihre Kennung PageID werden gesammelt ausgeschrieben (dirty Pages werden notiert); Beim Absturz werden **Redo** bis zur **MinDirtyPageLSN** und **Undo** bis zur **MinLSN** benötigt

*__DirtyPages:__ Menge der modifizierten Seiten
*__MinDirtyPageLSN:__ die minimale LSN, deren Änderungen noch nicht ausgeschrieben wurde
*__MinLSN:__ die kleinste LSN aller zum Sicherungspunkt aktiven Transaktionen

Kapitel 11 - Mehrbenutzersynchronisation

Fehler bei unkontrolliertem Mehrbenutzerbetrieb

- **Verlorengegangene Änderungen (lost Update):** eine von T_1 geschriebene Änderung geht verloren, da T_2 diese auf Basis eines alten Werts überschreibt
- **Abhängigkeit von nicht freigegebenen Änderungen (dirty read):** ein Wert wird gelesen, das so niemals in einem gültigen Zustand (da T_2 wird abgebrochen) der Datenbasis vorkommt
- **Non-repeatable read:** T_1 liest hier mehrfach den Wert von x ein, da T_2 dazwischen x überschrieben hat, bekommt T_1 zwei verschiedene Werte für x
- **Phantomproblem:** ein neuer Wert wird während der T_2-Abarbeitung eingefügt, der eigentlich hätte berücksichtigt werden müssen -> wiederholte Anfrage von T_1 vor und nach T_2-Neu-Einfügung führt zur verschiedenen Ergebnissen

T_1	T_2
bot	
$r_1(x)$	
↪	bot
	$r_2(x)$
$w_1(x)$	↩
↪	$w_2(x)$
commit	↩
↪	commit

T_1	T_2
bot	
↪	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	↩
$w_1(y)$	
commit	
↪	abort

T_1	T_2
bot	
$r_1(x)$	
↪	bot
	$w_2(x)$
	commit
$r_1(x)$	↩
...	

T_1	T_2
bot	
select count(*) from R;	
↪	bot
	insert into R ...;
	commit
select count(*) from R;	↩
...	

-> Idealerweise sollten diese Fehler alle verhindert werden -> Am sichersten: **Serialisierbare Transaktionen**

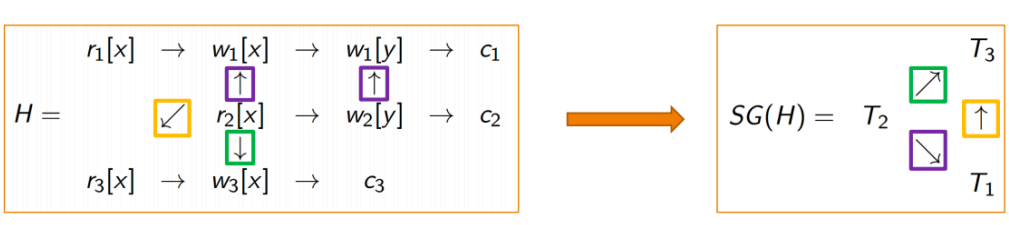
- **Serialisierbarkeit:** Bei serialisierbaren Historie ist das Ergebnis der Verzahnung äquivalent zur seriellen Ausführung (sonst nicht serialisierbar und wird vom DBS abgelehnt)
  - **Ti-Operationen: $r_i(A)$** Lesen von A; **$w_i(A)$** Schreiben von A; **a_i** abort/ Abbruch; **c_i** commit/ Erfolg
  - **Konsistenzanforderung einer Transaktion:** 1) entweder abort oder commit; 2) Führt T_i ein abort/ commit durch müssen alle anderen Operationen $p_i(A)$ vor a_i/ c_i ausgeführt werden; 3) T_i liest/ schreibt ein Datum, muss die Reihenfolge festgelegt werden
  - Historien der **Serialisierbarkeit:** zeitliche Anordnung der einzelnen verzahnt ausgeführten Elementaroperationen einer Menge von nebenläufig arbeitenden Transaktionen -> hier muss mindestens für alle Konfliktoperationen mit einer **Ordnungsrelation $<_i$** in eine Reihenfolge gebracht werden
    - **$r_i(A)$ und $r_j(A)$:** Reihenfolge irrelevant
    - **$r_i(A)$ und $w_j(A)$:** Konflikt, Reihenfolge muss spezifiziert werden
    - **$w_i(A)$ und $w_j(A)$:** Konflikt, Reihenfolge muss spezifiziert werden

Äquivalenz zweier Historien: Zwei Historien über der gleichen Menge von Transaktionen sind äquivalent, wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen.

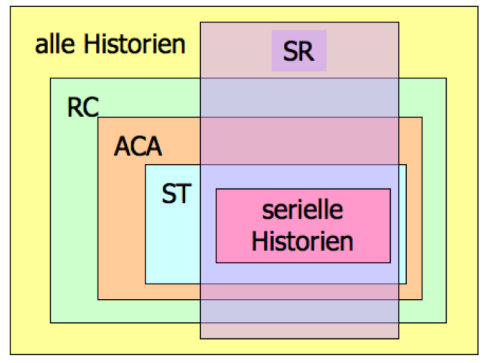
Serialisierbare Historie (SR): Eine Historie ist serialisierbar, wenn sie äquivalent zu einer seriellen Historie ist

- **Serialisierbarkeitsgraph (SG):** Knoten = Historie-Transaktionen, Kante $T_i \rightarrow T_j$, wenn Konfliktoperationen $p_i < q_j$
- **Serialisierbarkeitstheorem:** Historie H serialisierbar gdw. zugehörige Serialisierbarkeitsgraph SG(H) azyklisch 非循环

Transaktion mit `abort` sollte nicht in den Serialisierbarkeitsgraph gezeichnet werden



Hier: Azyklisch → serialisierbar (Reihenfolge: T_2 | T_1 | T_3)



Eigenschaften von Historien bezüglich der Recovery

- **Rücksetzbare Historien (RC):** rücksetzbar, falls immer die schreibende Transaktion T_j vor der lesenden Transation T_i, ihr `commit` durchführt, also: $c_j <_H c_i$ d.h. jede lesende Transaktion darf ihr `commit` erst dann durchführen, wenn alle schreibenden Transaktionen, von denen c_i sie gelesen hat, ihre `commit` bereits abgeschlossen haben -> kein Lesen dreckiger Daten
- **Historien ohne kaskadierendes Rücksetzen (ACA):** wenn immer eine TA T_i von einer anderen TA T_j liest, gilt $c_j <_H r_i(A)$ gilt (Es darf immer nur von bereits erfolgreich abgeschlossenen TAs gelesen werden) **falls keine Transaktion von anderen ließt, dann RC und ACA**
  - **kaskadierendes Rücksetzen:** durch `abort` einer Transaktion, müssen mehrere von ihr lesende Transaktionen zurückgesetzt
- **Strikte Historien (ST):** falls $w_i < o_j$ (mit $o_j = r_j(A)$ oder $w_j(A)$) gilt, dann auch $a_i < o_j$ oder $c_i < o_j$
  - Nur von abgeschlossenen TAs darf gelesen oder dürfen Datenobjekte überschrieben werden

Sperrbasierte Synchronisation

Mehrere Transaktionen können gleichzeitig eine S-Sperre auf dem selben Datenobjekt besitzen

Aber eine Transaktion kann maximal eine X-Sperre für ein Datum besitzen

- **Zwei Sperrmodi:** - **S** (shared, read lock, Lesesperre)
  - **X** (exclusive, write lock, Schreibsperre)

- **Verträglichkeitsmatrix (= Kompatibilitätsmatrix)**

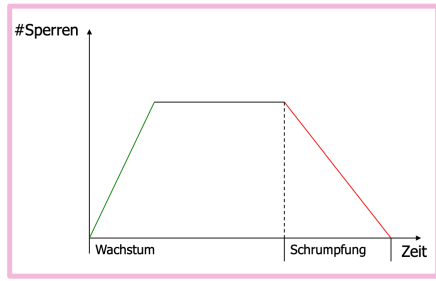
	NL	S	X
S	✓	✓	-
X	✓	-	-

- **Zwei-Phasen-Sperrprotokoll (2PL)** -> Deadlockbehandlung nötig

1. **Jedes Objekt das benutzt werden soll, muss vorher gesperrt werden** (die bereits von einer Transaktion besitzende Sperren werden nicht nochmal angefordert; Verträglichkeit der Sperren muss beachtet werden, ggf. muss die Transaktion warten)
2. **Jede Transaktion durchläuft zwei Phasen**
   - **Wachstumsphase:** Sperren werden angefordert, aber nicht freigegeben
   - **Schrumpfphase:** bereits erworbene Sperren werden freigegeben, aber darf keine weiteren angefordert werden
3. **Bei Transaktionsende werden alle Sperren zurückgegeben**

-> garantiert serialisierbare Historien, kaskadierendes Rücksetzen nicht ausgeschlossen

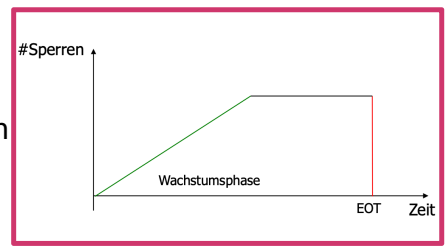
2PL garantiert Historien aus SR, strenge 2PL garantiert Historien aus SR und ST



- **Strenges Zwei-Phasen Sperrprotokoll (Strenges 2PL)** -> Deadlockbehandlung nötig

- Alle Sperren werden bis zum Transaktionsende (EOT) gehalten und auf einmal freigegeben

-> garantiert Serialisierbarkeit und Striktheit, kaskadierendes Rücksetzen ausgeschlossen



Verklemmungen (Deadlocks): Entstehen, wenn zwei Transaktionen gleichzeitig auf die Freigabe der jeweils andere besitzenden Sperren warten und sich damit gegenseitig blockiert

- **Erkennung durch Wartegraph: 1)** Wartegraph besitzt mindestens einen Zyklus/ Kreis; **2)** Kante $T_i \rightarrow T_j$, wenn T_i ein Lock anfordert, das T_j gerade hält; **3)** Wartegraph besitzt die umgedrehten Kanten des SG (Kante = „wartet auf")

-> Auflösen: indem eine der im Zyklus vorkommenden Transaktionen aufgelöst/ abgebrochen wird

Verklemmungsvermeidung durch Zeitstempel: Jeder Transaktion wird ein eindeutiger Zeitstempel (TS) zugeordnet (ältere TAs haben einen kleineren Zeitstempel als jüngere TAs); TAs dürfen nicht mehr bedingungslos auf Sperre warten

- **wound-wait Strategie:** 1) T_1 will Sperre erwerben, die von T_2 gehalten wird; 2) Wenn T_1 älter als T_2, wird T_2 abgebrochen und zurückgesetzt, so dass T_1 weiterlaufen kann; sonst wartet T_1 auf die Freigabe der Sperre durch T_2
- **wait-die Strategie:** 1) T_1 will Sperre erwerben, die von T_2 gehalten wird; 2) Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre; sonst wird T_1 abgebrochen und zurückgesetzt

Multi-Granularity Locking (MGL) -> Deadlockbehandlung nötig

- **Erweiterte Sperrmodi**
  - **NL:** keine Sperrung (no lock)
  - **S:** Sperrung durch Leser
  - **X:** Sperrung durch Schreiber
  - **IS (intention share):** Weiter unten in der Hierarchie ist eine Lesesperre (S) beabsichtigt
  - **IX (intention exclusive):** Weiter unten in der Hierarchie ist eine Schreibsperre (X) beabsichtigt

- **Verträglichkeitsmatrix (= Kompatibilitätsmatrix)**

angeforderte Sperre \ gehaltene Sperre	NL	S	X	IS	IX
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

- **Sperrprotokoll des MGL**

- Bevor ein Knoten mit S oder IS gesperrt wird, müssen alle Vorgänger in der Hierarchie vom Sperrer (also der Transaktion, die die Sperre anfordert) im IX- oder IS-Modus gehalten werden
- Bevor ein Knoten mit X oder IX gesperrt wird, müssen alle Vorgänger vom Sperrer im IX-Modus gehalten werden
- Die Sperren werden von unten nach oben (bottom up) freigegeben, so dass bei keinem Knoten die Sperre freigegeben wird, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat

Beispielausprägung

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

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

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

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

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

Assistenten			
PersNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2134

prüfen			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Abbildung 1: Beispielausprägung für eine Universitäts-Datenbank

Heruntergeladen von Studydrive