

TOC Zusammenfassung

Vorwort

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

Disclaimer

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

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

Inhalt

- [1. Relationale Algebra](#)
- [2. ER-Diagramme](#)
- [3. Normalization](#)
- [4. SQL](#)
- [5. Transaktionen](#)
- [6. Physischer Datenbankentwurf](#)
- [7. Anfrageoptimierung](#)

1. Relationale Algebra

Grundlagen des relationalen Modells

Domänen

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

Relationen

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

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

Relationenschema

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

Beispiele für zulässige Relationen

Gegeben

- Domänen $D_1 = \{1, 2, 3\}$, $D_2 = \{x, y, z\}$, $D_3 = \{\alpha, \beta, \gamma, \omega\}$
- Attribute auf diesen Domänen $A : D_1$, $B : D_2$, $C : D_3$

Welche der folgenden Relationen sind zulässig?

A	B	C
1	y	α
3	x	α
3	y	α

zulässig

A	B	C
α	y	α
3	x	α
2	y	β

nicht zulässig

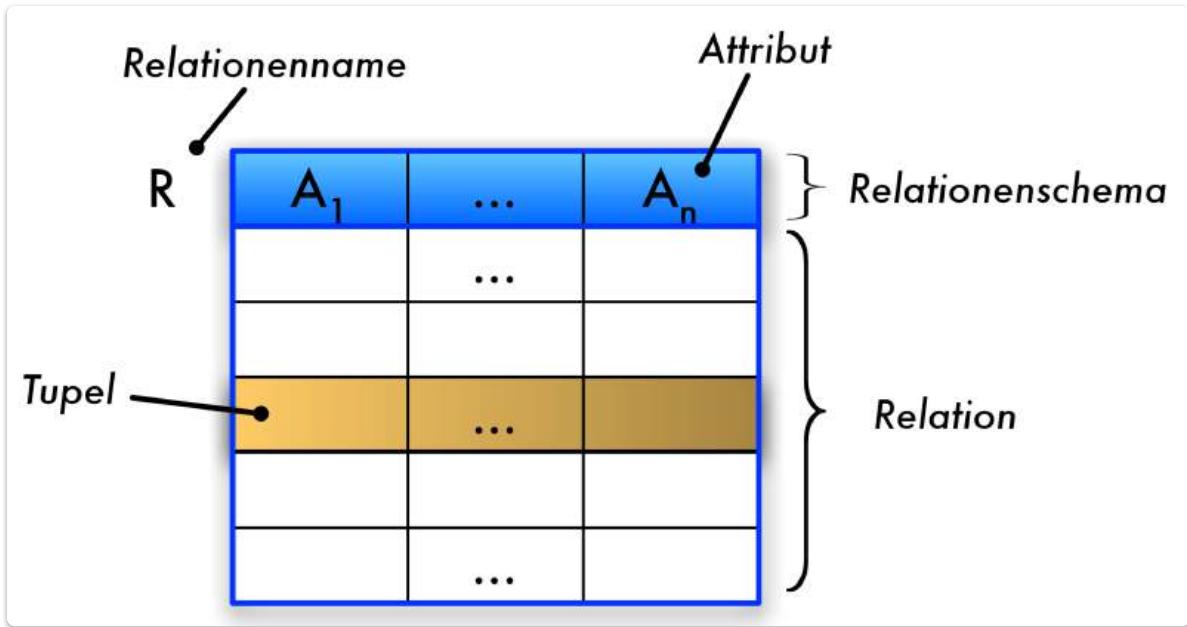
A	B	C
1	y	α
3	x	α
1	y	α

nicht zulässig

Veranschaulichung der Grundbegriffe

- **Fettgeschriebene Kopfzeile:** Relationenschema
- **Spaltenüberschrift:** Attribut
- **Weitere Einträge in der Tabelle:** Relation

- Eine Zeile der Tabelle: Tupel
- Ein Eintrag: Attributwert
- Unterstrichenes Attribut: Primärschlüssel



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

Relationenmodell

wines	wineID	name	color	year	vineyard → producer
1042	La Rose Grand Cru	red	1998	Château La Rose	
2168	Creek Shiraz	red	2003	Creek	
3456	Zinfandel	red	2004	Helena	
2171	Pinot Noir	red	2001	Creek	
3478	Pinot Noir	red	1999	Helena	

producer	vineyard	area	region
Creek	Barossa Valley	South Australia	
Helena	Napa Valley	California	
Château La Rose	Saint-Emilion	Bordeaux	
Château La Pointe	Pomerol	Bordeaux	

Fremdschlüssel

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

Eigenschaften des Relationalen Modells

Tupelreihenfolge

Tupel einer Relation haben keine **Reihenfolge**.

name	age
Pat	1
Fred	2
Sue	3
Pam	4

name	age
Sue	3
Pam	4
Fred	2
Pat	1

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

Attributreihenfolge

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

name	age
Pat	1
Fred	2
Sue	3
Pam	4

age	name
1	Pat
2	Fred
3	Sue
4	Pam

Aber...

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

Atomare Werte

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

name	age
Pat	1
Fred	2
Sue	3
Pam	4

Alle Werte sind atomar

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

name ist nicht atomar

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

Null Werte

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

name	age
Pat	1
Fred	2
Sue	null
Pam	null

name	age
Pat	1
Fred	2
Sue	⊥
Pam	⊥

Alternative Notation

Duplikate

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

name	age
Pat	1
Fred	2
Sue	3
Pam	4

zulässig

name	age
Pat	1
Fred	2
Sue	3
Sue	3

unzulässig

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

Beispiele

Welche der folgenden Relationen sind zulässig?

name	age
Pat	1
Fred	2
Sue	⊥
Sue	3

gültig

name	age
Pat	1
Fred	2
Sue	⊥
Sue	⊥

gültig

Relationale Algebra

Die Operatoren der relationalen Algebra

- Projektion π
- Selektion σ
- Umbenennung ρ
- Kreuzprodukt \times
- Vereinigung \cup
- Differenz $-$
- Schnitt \cap
- Join (Verbund) \bowtie
- Linker äußerer Join \bowtie_l
- Rechter äußerer Join \bowtie_r
- Äußerer Join \fullouterjoin
- Semi-Join (linker) \ltimes
- Semi-Join (rechter) \rtimes
- Gruppierung γ
- Division \div

Basisoperatoren

- Projektion π
- Selektion σ
- Umbenennung ρ
- Kreuzprodukt \times
- Vereinigung \cup
- Differenz $-$

Basisoperatoren

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

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

Unäre vs. binäre Operatoren

- **Unäre Operatoren:** σ, π, ρ (Arbeiten auf einer einzelnen Relation.)
- **Binäre Operatoren:** $\times, \cup, -$ (Arbeiten auf zwei Relationen.)

Auswertung der Operatoren

Operatoren und deren Verwendung

- Input: eine oder mehrere Relationen
- Output: eine Relation
- Operatoren können (nach bestimmten Regeln) kombiniert werden.

Projektion

$\pi_{name, dep_name}(instructor)$

instructor			
ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000

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

$\pi_{name, dep_name}(instructor)$	
name	dep_name
Srinivasan	Comp. Sci.
Wu	Finance
Mozart	Music
Einstein	Physics

Erweiterte Projektion

$\pi_{ID, name, dep_name, salary \div 12}(instructor)$

Vorher:

instructor			
ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

Nacher:

instructor			
ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	5417
12121	Wu	Finance	7500
15151	Mozart	Music	3333
22222	Einstein	Physics	7917
32343	El Said	History	5000
33456	Gold	Physics	7250

Selektion

- Symbol: σ_F
- Selektionsprädikat F besteht aus:

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

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

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

Auswahl von Zeilen einer Tabelle anhand eines Selektionsprädikats.

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

instructor			
ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

$\sigma_{\text{salary} > 80000}(\text{instructor})$			
ID	name	dept_name	salary
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Umbenennung (ρ)

Die Umbenennungsoperation dient dazu, Namen von Relationen oder Attributen zu ändern. Dies ist besonders nützlich, um Eindeutigkeit in komplexen Abfragen zu gewährleisten oder die Lesbarkeit zu verbessern. In der Literatur wird auch oft das Symbol β verwendet.

Umbenennung einer Relation

- Syntax: $\rho_S(R)$

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

Umbenennung von Spalten (Attributen)

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

Kartesisches Produkt (Kreuzprodukt) (\times)

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

- **Syntax:** $(R \times S)$
- **Erklärung:** Das Kreuzprodukt zweier Relationen R und S besteht aus allen möglichen Kombinationen von Tupeln. Die Anzahl der resultierenden Tupel ist das Produkt der Anzahlen der Tupel in R und S , also $|R| \times |S|$ Paare.
- **Schema des Resultats:**
 - $sch(R \times S) = sch(R) \cup sch(S) = R \cup S$
 - Das Schema der Ergebnisrelation ist die Vereinigung der Schemata der beiden Ausgangsrelationen.
- **Wichtiger Hinweis:** Das Kartesische Produkt enthält oft viele (auch viele unsinnige) Kombinationen. Es ist eine grundlegende Operation, die häufig als Basis für Joins verwendet wird, bei denen dann zusätzliche Bedingungen angewendet werden, um nur die relevanten Kombinationen zu erhalten.
- **Referenzierung von Attributen:** Beim Referenzieren der Attribute des resultierenden Schemas wird $R.A$ und $S.A$ verwendet. Dies ist besonders wichtig bei Überlappungen der einzelnen Schemata (d.h. wenn beide Relationen Attribute mit dem gleichen Namen haben), um Unklarheiten zu vermeiden, welches Attribut gemeint ist.
 - **Beispiel:** Wenn sowohl Relation `Kunden` als auch Relation `Bestellungen` ein Attribut `ID` haben, würde man im Kreuzprodukt `Kunden.ID` und `Bestellungen.ID` verwenden, um sie zu unterscheiden.

Beispiel

$$\pi_{V1.course}(\sigma_{V2.successor=5216 \wedge V1.successor=V2.course}$$

$$(\rho_{V1}(prerequisite) \times \rho_{V2}(prerequisite)))$$

prerequisite	
course	successor
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

V1.course	V1.successor	V2.course	V2.successor
5001	5041	5041	5216

V1.course
5001

Vorgänger zweiter Stufe der Vorlesung mit
Nummer 5216

Relationale Algebra: Mengenoperationen

Die grundlegenden Mengenoperationen **Vereinigung**, **Schnitt** und **Differenz** können auch auf Relationen angewendet werden.

Voraussetzung für Mengenoperationen: Vereinigungskompatibilität

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

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

Vereinigung (\cup)

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

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

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

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

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

- **Syntax:** $(R - S)$ bzw. $(R \setminus S)$
- **Beispiel:** Angenommen, wir haben zwei Relationen `instructor_departments` und `student_departments`, die jeweils die Abteilungsnamen enthalten, in denen Instruktoren bzw. Studenten tätig sind.

<code>instructor_departments</code>	<code>student_departments</code>
dept_name	dept_name
Comp. Sci.	Comp. Sci.
Music	History
History	Finance
Biology	Physics
Elec. Eng.	Music

`instructor_departments` – `student_departments`

- Die Differenz würde die Abteilungsnamen zurückgeben, in denen Instruktoren tätig sind, aber keine Studenten.

dept_name
Biology
Elec. Eng.

Überblick über die Basisoperatoren

Ausdruck	Schema	Arität	Min. Kardinalität	Max. Kardinalität
$\sigma_F(R)$	\mathcal{R}	$ \mathcal{R} $	0	$ \mathcal{R} $
$\pi_L(R)$	L	$\leq \mathcal{R} $	0	$ \mathcal{R} $
$R \cup S$	$\mathcal{R} (= \mathcal{S})$	$ \mathcal{R} (= \mathcal{S})$	$\max(\mathcal{R} , \mathcal{S})$	$ \mathcal{R} + \mathcal{S} $
$R - S$	$\mathcal{R} (= \mathcal{S})$	$ \mathcal{R} (= \mathcal{S})$	0	$ \mathcal{R} $
$R \times S$	$\mathcal{R} \circ \mathcal{S}$	$ \mathcal{R} + \mathcal{S} $	$ \mathcal{R} \cdot \mathcal{S} $	$ \mathcal{R} \cdot \mathcal{S} $
$\rho_S(R)$	\mathcal{R}	$ \mathcal{R} $	$ \mathcal{R} $	$ \mathcal{R} $

Schnitt (Durchschnitt) (\cap)

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

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

- **Erklärung:** Der Schnitt liefert alle Tupel, die sowohl in Relation R als auch in Relation S enthalten sind.
- **Ableitung aus Differenz:** Der Schnitt kann wie folgt als Differenz ausgedrückt werden:
 - $(R \cap S = R - (R - S))$
 - **Intuitive Erklärung:** Nimm alle Tupel aus R . Ziehe davon alle Tupel ab, die *nicht* in S sind (also die Tupel, die nur in R vorkommen). Was übrig bleibt, sind genau die Tupel, die sowohl in R als auch in S sind.
- **Beispiel:** Angenommen, wir haben weiterhin die Relationen `instructor_departments` und `student_departments`.

<code>instructor_departments</code>	<code>student_departments</code>
<code>dept_name</code>	<code>dept_name</code>
Comp. Sci.	Comp. Sci.
Music	History
History	Finance
Biology	Physics
Elec. Eng.	Music

`instructor_departments` \cap `student_departments`

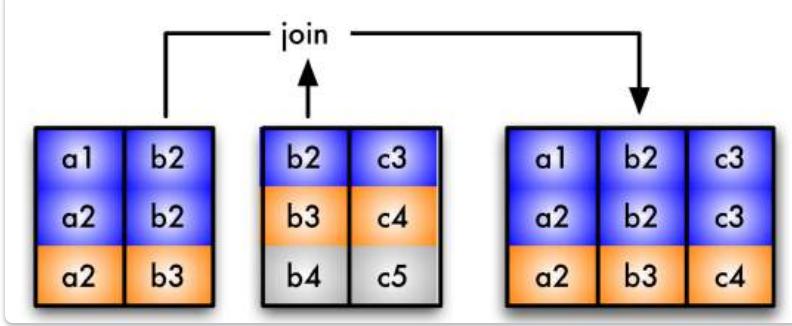
- Der Schnitt würde die Abteilungsnamen zurückgeben, in denen sowohl Instruktoren als auch Studenten tätig sind.

<code>dept_name</code>
Comp. Sci.
Music
History

Natürlicher Verbund (Natural Join) (\bowtie)

Der natürliche Verbund ist eine wichtige Operation, um Informationen aus zwei oder mehr Relationen basierend auf gemeinsamen Attributen zu kombinieren. Er verknüpft Tabellen (Relationen) über **gleichbenannte Spalten** und verschmilzt jeweils zwei Tupel, wenn sie dort **gleiche Werte** aufweisen.

- **Gegeben zwei Relationen (+ Schemata):**
 - $R(A_1, \dots, A_m, B_1, \dots, B_k)$
 - $S(B_1, \dots, B_k, C_1, \dots, C_n)$



- **Erklärung zum Bild:** Die Relationen R und S haben gemeinsame Attribute B_1, \dots, B_k . Der Natural Join sucht Tupelpaare, bei denen die Werte dieser gemeinsamen Attribute übereinstimmen, und kombiniert diese Tupel zu einem neuen, breiteren Tupel. Die gemeinsamen Attribute erscheinen im Ergebnis nur einmal.
- **Der natürliche Join kann durch ein Kreuzprodukt gefolgt von Selektionen und Projektionen ausgedrückt werden.** Dies zeigt, dass der Natural Join kein fundamentaler Basisoperator ist, sondern eine abgeleitete Operation, die bequem ist, aber mit grundlegenderen Operationen simuliert werden kann.
 - **Syntax:** $R \bowtie S = \pi_{A_1, \dots, A_m, R.B_1, \dots, R.B_k, C_1, \dots, C_n} (\sigma_{R.B_1=S.B_1 \wedge \dots \wedge R.B_k=S.B_k} (R \times S))$
 - **Intuitive Erklärung der Ableitung:**
 1. **Kartesisches Produkt ($R \times S$):** Zuerst werden alle möglichen Kombinationen von Tupeln aus R und S gebildet. Das Ergebnis enthält dann doppelte Spalten für die gemeinsamen Attribute (z.B. $R.B_1$ und $S.B_1$).
 2. **Selektion ($\sigma_{...}$):** Anschließend werden nur die Tupel ausgewählt, bei denen die Werte der gleichbenannten Attribute aus R und S übereinstimmen (z.B. $R.B_1 = S.B_1$, $R.B_2 = S.B_2$, usw.). Dies filtert die "sinnvollen" Kombinationen heraus.
 3. **Projektion ($\pi_{...}$):** Zuletzt werden die gewünschten Attribute projiziert. Dabei werden die doppelten Spalten der gemeinsamen Attribute entfernt, sodass jedes gemeinsame Attribut nur einmal im Endergebnis erscheint.
- **Schema des Resultats:**

$$R \bowtie S = \pi_{A_1, \dots, A_m, R.B_1, \dots, R.B_k, C_1, \dots, C_n} (\sigma_{R.B_1=S.B_1 \wedge \dots \wedge R.B_k=S.B_k} (R \times S))$$

$R \bowtie S$											
$R - S$				$R \cap S$				$S - R$			
A_1	A_2	\dots	A_m	B_1	B_2	\dots	B_k	C_1	C_2	\dots	C_n
\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots

- Das resultierende Schema enthält alle Attribute aus R und S , wobei die gemeinsamen Attribute nur einmal auftauchen.
- **Beispiel Schema:** $R \bowtie S$ hätte die Attribute $(A_1, \dots, A_m, B_1, \dots, B_k, C_1, \dots, C_n)$.
- **Reihenfolge der Attribute:** Die Reihenfolge der Attribute in der Ergebnisrelation wird durch die Attribute der gegebenen Relationen bestimmt. Das heißt, das resultierende Schema ist nicht unbedingt wie im obigen Beispiel sortiert (z.B. zuerst alle Attribute von R , dann alle von S), sondern kann auch anders angeordnet sein, je nach Implementierung.

Beispiel

wines \bowtie producer

wines	wineID	name	color	year	vineyard
	1042	La Rose Grand Cru	red	1998	Château La Rose
	2168	Creek Shiraz	red	2003	Creek
	2171	Pinot Noir	red	2001	Creek
	4711	Riesling Reserve	white	1999	Müller

producer	vineyard	area	region		
	Creek	Barossa Valley	South Australia		
	Helena	Napa Valley	California		
	Château La Rose	Saint-Emilion	Bordeaux		

Resultat:

wineID	name	...	vineyard	...	region
1042	La Rose Grand Cru	...	Ch. La Rose	...	Bordeaux
2168	Creek Shiraz	...	Creek	...	South Australia
2171	Pinot Noir	...	Creek	...	South Australia

Tupel, die keinen Partner finden (*dangling tuples*), werden "eliminiert".

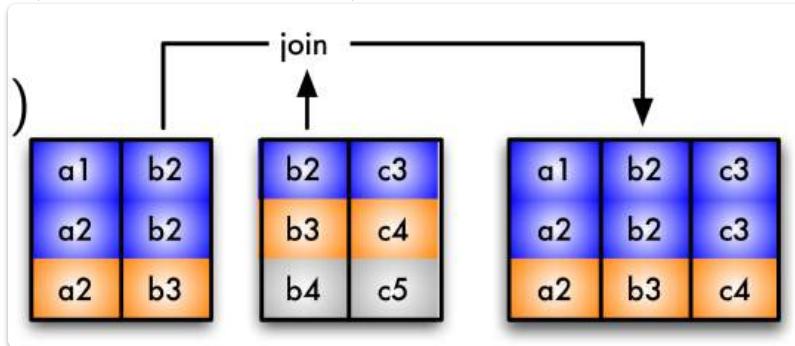
Join Kommutativität

Eine wichtige Eigenschaft von Operationen ist die Kommutativität. Bei Joins stellt sich die Frage, ob die Reihenfolge der Operanden das Ergebnis beeinflusst.

Ist $R \bowtie S = S \bowtie R$ wahr oder falsch?

- Gegeben zwei Relationen (+ Schema):

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



- Die Reihenfolge beeinflusst nicht "wirklich" das Resultat:

- Die Reihenfolge der Attribute im *Schema* der Ergebnisrelation ist unterschiedlich.
- Aber der "*Inhalt*" der Tupel und Relationen ist "*gleich*". Das bedeutet, die Menge der Informationen, die durch die Verknüpfung gewonnen wird, ist identisch, unabhängig davon, ob man R mit S oder S mit R verknüpft. Die Tupel sind dieselben, nur die Reihenfolge der Spalten kann variieren.

- Betrachtung der Kommutativität in der Praxis:

- Momentan (im Kontext der Basiskonzepte) werden Joins und Kreuzprodukte *nicht als kommutativ* betrachtet, wenn man streng die Reihenfolge der Attribute im

Ergebnis beachtet.

- Bei der Anfrageoptimierung (ein späteres Thema in der Lehrveranstaltung) werden natürliche Joins sowie Kreuzprodukte und andere Join-Varianten jedoch *als kommutativ* behandelt. Dies ist entscheidend, um die Reihenfolge der Join-Operationen zu optimieren und so die Effizienz von Datenbankabfragen zu verbessern.
 - Exakte mathematische Definition:
 - Wenn wir die mathematische Definition von Tupeln (die eine feste Reihenfolge von Elementen haben) und Joins (die eine feste Reihenfolge der Attribute im Schema haben) einhalten und trotzdem Kommutativität annehmen wollen, müssen wir eine **Projektion vornehmen, um die Attribute entsprechend zu sortieren.**
 - $\pi_L(R \bowtie S) = \pi_L(S \bowtie R)$
 - Wobei L eine Liste der Attribute ist, die eine einheitliche Sortierung für beide Ergebnisse sicherstellt. Dies bedeutet, dass die Tupelmengen nach der Projektion identisch sind, auch wenn die ursprünglichen Schemata der Joins unterschiedliche Attributreihenfolgen hatten.
-

Relationale Algebra: Zusätzliche Join-Varianten

Neben dem Natural Join gibt es weitere Verbundoperationen, die spezifische Anforderungen an die Verknüpfung von Relationen erfüllen.

Theta-Join (θ -Join oder allgemeiner Join)

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

- **Gegeben zwei Relationen (+ Schemata):**
 - $R(A_1, \dots, A_n)$
 - $S(B_1, \dots, B_m)$
 - θ ist ein beliebiges Prädikat über den beteiligten Attributen (z.B. $R.A_i > S.B_j$, $R.A_i \neq S.B_j$, etc.).
- **Syntax:** $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
 - **Erklärung:** Der Theta-Join wird ausgedrückt als ein **Kartesisches Produkt** ($R \times S$) gefolgt von einer **Selektion** (σ_{θ}), die alle Tupel aus dem Kreuzprodukt herausfiltert, für die das angegebene Prädikat θ zutrifft.

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

$R \bowtie_{\theta} S$							
\mathcal{R}				\mathcal{S}			
A_1	A_2	\dots	A_n	B_1	B_2	\dots	B_m
\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots

- Das Schema des Theta-Joins ist einfach die Konkatenation der Schemata von R und S , da keine Attribute aufgrund von Gleichheit verschmolzen werden wie beim Natural Join.

Equi-Join

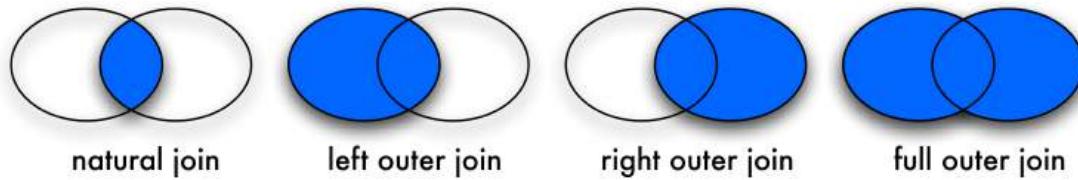
Der Equi-Join ist ein Spezialfall des Theta-Joins, bei dem das Join-Prädikat θ **nur Gleichheit** ($=$) prüft.

- **Erklärung:** Anstatt eines beliebigen Vergleichsoperators ($<$, $>$, \neq , etc.) verwendet der Equi-Join ausschließlich den Gleichheitsoperator ($=$) für die Verknüpfungsbedingung zwischen den Attributen der Relationen. Der Natural Join ist ein Spezialfall des Equi-Joins, der zusätzlich die doppelten Attribute nach dem Join entfernt.

Äußere Verbunde (Outer Joins)

Äußere Verbunde sind wichtig, wenn man Tupel erhalten möchte, die **keinen Joinpartner** in der anderen Relation finden. Diese Tupel werden als "partnerlose" Tupel bezeichnet. Normalerweise würden diese Tupel bei einem (inneren) Join verloren gehen. Outer Joins füllen die fehlenden Attribute dieser partnerlosen Tupel mit Nullwerten auf.

- ☒ Left outer Join (Linker äußerer Verbund):
auch "partnerlose" Tupel der linken Relation bleiben erhalten
- ☒ Right outer Join (Rechter äußerer Verbund):
auch "partnerlose" Tupel der rechten Relation bleiben erhalten
- ☒ (Full) outer Join (Vollständiger äußerer Verbund):
die "partnerlosen" Tupel beider Relationen bleiben erhalten



Natural Join (Natürlicher Verbund)

L			R			Result				
A	B	C	C	D	E	A	B	C	D	E
a_1	b_1	c_1	c_1	d_1	e_1	a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	c_3	d_2	e_2					

Left Outer Join (Linker äußerer Verbund)

L			R			Result				
A	B	C	C	D	E	A	B	C	D	E
a_1	b_1	c_1	c_1	d_1	e_1	a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	c_3	d_2	e_2	a_2	b_2	c_2	\perp	\perp

Right Outer Join (Rechter äußerer Verbund)

L			R			Result				
A	B	C	C	D	E	A	B	C	D	E
a_1	b_1	c_1	c_1	d_1	e_1	a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	c_3	d_2	e_2	\perp	\perp	c_3	d_2	e_2

Outer Join (Äußerer Verbund)

L			R			Result				
A	B	C	C	D	E	A	B	C	D	E
a_1	b_1	c_1	c_1	d_1	e_1	a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	c_3	d_2	e_2	a_2	b_2	c_2	\perp	\perp
			\perp	\perp	c_3	\perp	c_3	d_2	e_2	

Semi-Joins

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

Linker Semi-Join (\ltimes)

- **Definition:** Finde alle Tupel der **linken Relation**, die Joinpartner in der **rechten Relation** haben.
- **Syntax:** $L \ltimes R = \pi_L(L \bowtie R)$
 - **Erklärung:** Ein linker Semi-Join zwischen Relation L und R kann ausgedrückt werden als eine Projektion (π_L) des Ergebnisses eines Natural Joins ($L \bowtie R$) zurück auf die Attribute der linken Relation L .
 - L repräsentiert hier die Menge der Attribute der linken Relation, die im Ergebnis erscheinen sollen. Da es ein Semi-Join ist, sind dies alle Attribute der linken Relation.
- **Beispiel Linker Semi-Join ($L \ltimes R$):**

L			R			Result		
A	B	C	C	D	E	A	B	C
a_1	b_1	c_1	c_1	d_1	e_1	a_1	b_1	c_1
a_2	b_2	c_2	c_3	d_2	e_2			

- **Ergebnis ($L \ltimes R$):** Es werden nur die Tupel aus L ausgegeben, für die es eine Übereinstimmung in R gibt (hier über das Attribut C). Die Spalten von R werden nicht im Ergebnis aufgeführt.

Rechter Semi-Join (\bowtie)

- **Definition:** Finde alle Tupel der **rechten Relation**, die Joinpartner in der **linken Relation** haben.
- **Syntax:** $L \bowtie R = \pi_R(L \bowtie R)$
 - **Erklärung:** Ein rechter Semi-Join zwischen Relation L und R kann ausgedrückt werden als eine Projektion (π_R) des Ergebnisses eines Natural Joins ($L \bowtie R$) zurück auf die Attribute der rechten Relation R .
 - R repräsentiert hier die Menge der Attribute der rechten Relation, die im Ergebnis erscheinen sollen.
- **Beispiel Rechter Semi-Join ($L \bowtie R$):**

L			R			Result		
A	B	C	C	D	E	C	D	E
a_1	b_1	c_1	c_1	d_1	e_1	c_1	d_1	e_1
a_2	b_2	c_2	c_3	d_2	e_2			

\times =

- **Ergebnis ($L \bowtie R$):** Es werden nur die Tupel aus R ausgegeben, für die es eine Übereinstimmung in L gibt (hier über das Attribut C). Die Spalten von L werden nicht im Ergebnis aufgeführt.

Gruppierung und Aggregation

Die Gruppierung und Aggregation sind mächtige Operationen, um Daten in einer Relation zusammenzufassen und Berechnungen auf diesen zusammengefassten Daten durchzuführen.

Gruppierung

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

Aggregation

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

Typische Aggregatfunktionen:

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

Notation:

- $\mathcal{G}_{L,F}(R)$
 - L : Dies ist eine Liste der Attribute, nach denen die Gruppierung erfolgen soll (die "GROUP BY"-Attribute in SQL).
 - F : Dies ist die Aggregatfunktion, die auf jede Gruppe angewendet wird (z.B. `count(*)`, `sum(Gehalt)`, `avg(Alter)`).
 - Alternative Symbole \mathcal{G} oder β können ebenfalls für diese Operation verwendet werden.

Beispiele

Die Anzahl der Studierenden pro Semester

$$\gamma_{\text{semester}; \text{count}(*)}(\text{student})$$

student		
studID	name	semester
24002	Nielsen	18
25403	Hansen	12
26120	Pedersen	10
26830	Andersen	6
27550	Larsen	6

$\gamma_{\text{semester}; \text{count}(*)}(\text{student})$	
semester	count(*)
18	1
12	1
10	1
6	2

Die Anzahl der Studierenden pro Semester sowie
die minimale studID pro Gruppe

$$\gamma_{\text{semester}; \text{count}(*), \text{min(studID)}}(\text{student})$$

student		
studID	name	semester
24002	Nielsen	18
25403	Hansen	12
26120	Pedersen	10
26830	Andersen	6
27550	Larsen	6

$\gamma_{\text{semester}; \text{count}(*), \text{min(studID)}}(\text{student})$		
semester	count(*)	min(studID)
18	1	24002
12	1	25403
10	1	26120
6	2	26830

Aggregatfunktionen werden auch auf Duplikaten evaluiert, um z.B. die
Summe korrekt berechnen zu können!

count vs. count-distinct, sum vs. sum-distinct, etc.

Relationale Division (\div)

Die relationale Division ist eine Operation, die alle Tupel aus einer Relation findet, die mit *allen* Tupeln einer anderen Relation in Beziehung stehen. Sie ist besonders nützlich für "für alle"-Anfragen.

- **Grundidee:** Finde Elemente (aus einer Menge von Werten in Relation R), die mit *allen* Elementen einer bestimmten Menge (aus Relation S) assoziiert sind.
- **Beispiel:** Finde die `studIDs` der Studierenden, die *alle* Vorlesungen mit 4 ECTS besuchen.
 - **Relation takes :** `(studID, courseID)` - welche Studierenden welche Vorlesungen belegen
 - **Relation course :** `(courseID, title, ects, teacher)` - Informationen zu Vorlesungen
 - **Gesuchte Vorlesungen (alle 4 ECTS-Vorlesungen):** $\pi_{courseID}(\sigma_{ects=4}(course))$
- **Formale Definition:**

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

- **Intuitive Erklärung der Definition:**

1. **Schritt 1: Projektion von R auf die Nicht-S-Attribute** ($\pi_{R-S}(R)$): Dies sind die möglichen "Kandidaten" für das Ergebnis der Division (im Beispiel: die `studIDs`).
2. **Schritt 2: Kreuzprodukt der Kandidaten mit S** ($\pi_{R-S}(R) \times S$): Für jeden Kandidaten wird eine "erwartete" Menge an Tupeln generiert, die alle Kombinationen des Kandidaten mit jedem Element aus S enthält. Dies repräsentiert, welche Kombinationen jeder Kandidat *haben sollte*, um alle Elemente von S zu "erfüllen".
3. **Schritt 3: Differenz (($\pi_{R-S}(R) \times S$) – R):** Von diesen erwarteten Tupeln wird die ursprüngliche Relation R abgezogen. Das Ergebnis sind die Tupel, die ein Kandidat *nicht* hat, obwohl er sie haben *sollte*, um alle Elemente von S zu erfüllen.
4. **Schritt 4: Letzte Differenz ($\pi_{R-S}(R) - \dots$):** Von den ursprünglichen Kandidaten (aus Schritt 1) werden diejenigen abgezogen, die in Schritt 3 als "nicht vollständig" identifiziert wurden. Was übrig bleibt, sind genau die Kandidaten, die *alle* erforderlichen Verbindungen zu S hatten.

- **Beispiel zur Veranschaulichung der Division:**

$$\text{takes} \div \pi_{\text{courseID}}(\sigma_{\text{ects}=4}(\text{course}))$$

takes	
studID	courselD
26120	5001
27550	5001
27550	4052
28106	4052
28106	4630
28106	5001
28106	5041
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

course			
courselD	title	ects	teacher
5001	DBS	4	2137
5041	Robotics	4	2125
5043	Software Engineering	3	2126
5049	Ethics	2	2125
4052	Logic	4	2125
5052	Theory of Science	3	2126
5216	Bioethics	2	2126
5259	Chemistry	2	2133
5022	Believe and Knowledge	2	2134
4630	Physics	4	2137

takes	
studID	courselD
26120	5001
27550	5001
27550	4052
28106	4052
28106	4630
28106	5001
28106	5041
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

$\pi_{\text{courseID}}(\sigma_{\text{ects}=4}(\text{course}))$	
courselD	
5001	
5041	
4052	
4630	

result	
studID	
28106	

R	
M	V
m_1	v_1
m_1	v_2
m_1	v_3
m_1	v_4
m_2	v_1
m_2	v_2
m_3	v_1
m_3	v_3
m_4	v_3

S_1
V
v_1

$R \div S_1$
M
m_1
m_2
m_3

S_2
V
v_1
v_2

$R \div S_2$
M
m_1
m_2

S_3
V
v_1
v_2
v_3

$R \div S_3$
M
m_1

- Ein weiteres Beispiel (Wine Recommendation):

wineRecommendation

wine	reviewer
La Rose Grand Cru	Parker
Pinot Noir	Parker
Riesling Reserve	Parker
La Rose Grand Cru	Clarke
Pinot Noir	Clarke
Riesling Reserve	Gault-Millau

wineGuide1

reviewer
Parker
Clarke

wineGuide2

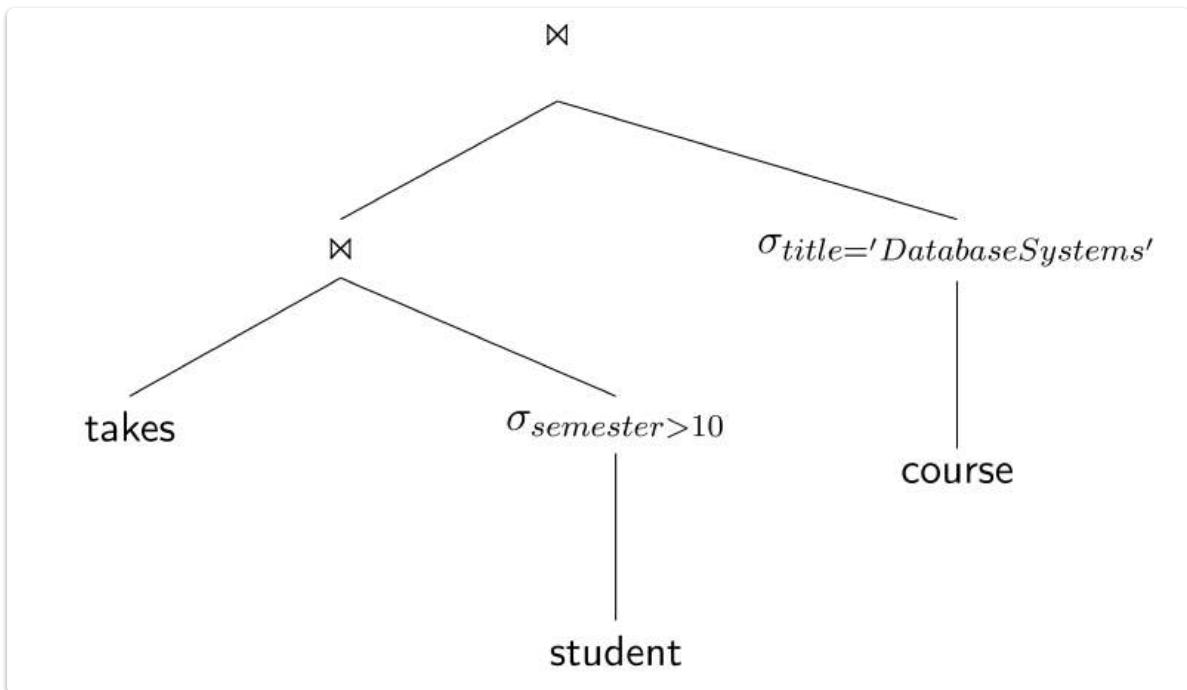
reviewer
Parker
Gault-Millau

$$wineRecommendation \div wineGuide2$$

wine
Riesling Reserve

Operatorenbaumdarstellung

Die Operatorenbaumdarstellung ist eine visuelle Methode, um relationale Algebra-Ausdrücke darzustellen. Sie zeigt die Reihenfolge der Operationen und deren Abhängigkeiten. Die Blätter des Baumes sind die Relationen, und die inneren Knoten sind die relationalen Operatoren.



- **Erklärung:** In einem Operatorenbaum fließen die Daten von den Blättern (den ursprünglichen Relationen) nach oben zu den Wurzelknoten, wo die letzte Operation ausgeführt wird. Jeder innere Knoten repräsentiert eine Operation der relationalen Algebra (z.B. Projektion, Selektion, Join, Vereinigung etc.). Die Reihenfolge, in der die Operationen ausgeführt werden, ist hierarchisch von unten nach oben im Baum. Diese Darstellung ist besonders wichtig für die Anfrageoptimierung in Datenbanksystemen, da verschiedene Baumstrukturen (die das gleiche Ergebnis liefern) unterschiedliche Ausführungszeiten haben können.

Einschränkungen der relationalen Algebra

- **Eingeschränkte Arithmetik**
 - Kann beispielsweise nicht direkt die Mietpreiserhöhung um 10% berechnen (z.B. "Finde den Mietpreis bei einer angenommenen Erhöhung von 10%").
- **Aggregatfunktionen fehlen in den Basisoperatoren der relationalen Algebra**
 - Fragen wie "Wie viele Filme hat jeder Kunde reserviert?" können nicht direkt beantwortet werden, da Funktionen wie SUM, COUNT, AVG fehlen.
- **Transitive Hülle kann nicht gebildet werden**
 - Beispiel: Für eine $Part(Part, ConstituentPart)$ Relation ist es schwierig, alle Teile zu finden, aus denen ein Auto besteht, wenn es Unterteilungen gibt (z.B. Motor besteht

aus X, X aus Y, Y aus Z).

- **Kein Sortieren oder Ausgeben in verschiedenen Formaten**
 - Es ist nicht möglich, eine Zusammenfassung von Rechnungen nach Kundennamen sortiert anzuzeigen. Die relationale Algebra liefert lediglich unstrukturierte Mengen von Tupeln.
- **Keine Modifikationen an einer Relation möglich**
 - Operationen wie "Erhöhe alle 3.25 Preise auf 3.50" sind nicht vorgesehen. Die relationale Algebra ist primär für Abfragen gedacht, nicht für Datenmanipulation.

Erweiterte Relationale Algebra

- Diese Einschränkungen werden in der **erweiterten relationalen Algebra** behoben.

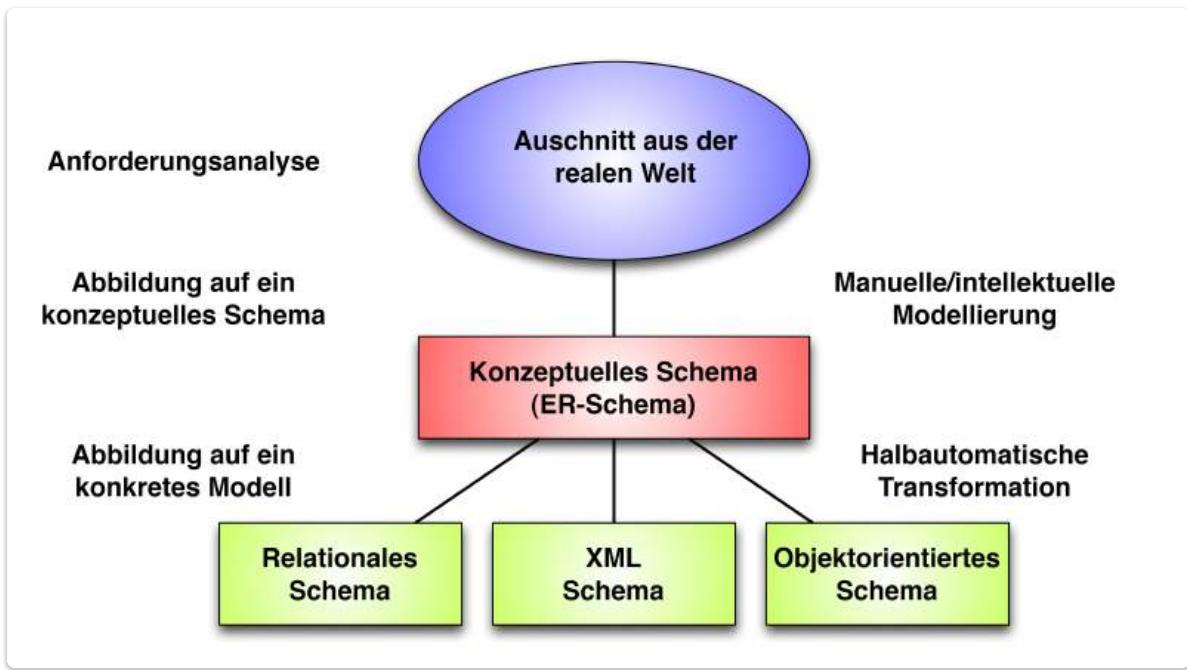
Zusammenfassung

- **Relationenmodell**
 - Keine Duplikate (mengenbasiert) - (Jedes Tupel in einer Relation ist einzigartig, wie in einer mathematischen Menge).
- **Relationale Algebra**
 - Jeder Operator erfordert Relationen als Input und erzeugt eine Relation als Output.
 - **Operatoren**
 - Sechs fundamentale Operatoren.
 - Viele abgeleitete Operatoren.
 - Operatoren erzeugen eine Abfragesprache.
 - Vereinigungskompatibilität (Relationen müssen die gleiche Anzahl und die gleichen Typen von Attributen haben, um vereint werden zu können).
 - **Verschiedene Join-Arten**
 - Es ist wichtig, die verschiedenen Join-Typen in der **Praxis** zu schätzen und zu verstehen, wenn man Abfragen an eine Datenbank stellt.
- Die relationale Algebra ist die Basis für **Anfragebearbeitung und -optimierung**.

2. ER-Diagramme

Datenbankentwurf

Schritte des Datenbankentwurfs



1. Analyse des realen Welt-Ausschnitts:

- Startpunkt ist ein relevanter Ausschnitt aus der realen Welt, der in die Datenbank integriert werden soll.
- Dies beinhaltet eine **manuelle/intellektuelle Modellierung** durch den Datenbankdesigner.

2. Anforderungsanalyse:

- Erfassen aller relevanten Anforderungen an die Datenbank (Was soll die Datenbank können? Welche Daten müssen gespeichert werden?).

3. Abbildung auf ein Konzeptuelles Schema (ER-Schema):

- Das konzeptuelle Schema ist eine abstrakte Beschreibung der Daten und ihrer Beziehungen, unabhängig von einer spezifischen Datenbanktechnologie.
- Dies ist ein wichtiger Zwischenschritt, da es die Komplexität der realen Welt auf ein verständlicheres Modell reduziert.

4. Abbildung auf ein konkretes Modell:

- Das konzeptuelle Schema wird in ein spezifisches Datenmodell überführt. Dies kann sein:
 - **Relationales Schema** (für relationale Datenbanken)
 - **XML-Schema** (für XML-Datenbanken)
 - **Objektorientiertes Schema** (für objektorientierte Datenbanken)

- Dieser Schritt kann **halbautomatisch** erfolgen.
-

Schritt 1: Anforderungsanalyse

- **Definition:** Die Anforderungsanalyse ist der Prozess des Sammelns und Analysierens von Informationen über die Bedürfnisse und Erwartungen der Benutzer an das zukünftige Datenbanksystem.
- **Beispiele für Anforderungen:**
 - Studierende nehmen an Vorlesungen teil.
 - Professor:innen bieten Vorlesungen an.
 - Studierende werden durch die Matrikelnummer eindeutig identifiziert.

Objektbeschreibung

- **Ziel:** Identifizierung und Beschreibung der Objekte (Entitäten) und ihrer Attribute, die in der Datenbank gespeichert werden sollen.
- **Beispiel: Angestellte der Universität**
 - **PersonalNummer**
 - Typ: char (Zeichenkette)
 - Länge: 9
 - Wertebereich: 0.. .999.999.999
 - Verfügbarkeit: 100% (muss immer vorhanden sein)
 - Identifizierend: ja (kann einen Angestellten eindeutig identifizieren)
 - **Gehalt**
 - Typ: dezimal (Dezimalzahl)
 - Länge: (5, 2) (5 Ziffern insgesamt, davon 2 nach dem Komma, z.B. 123.45)
 - Verfügbarkeit: 10% (kann optional sein, nicht jeder Angestellte hat ein Gehalt im System hinterlegt)
 - Identifizierend: nein (ein Gehalt identifiziert keinen Angestellten eindeutig)
 - **Rang**
 - Typ: String (Text)
 - Länge: 50
 - Verfügbarkeit: 100%
 - Identifizierend: nein

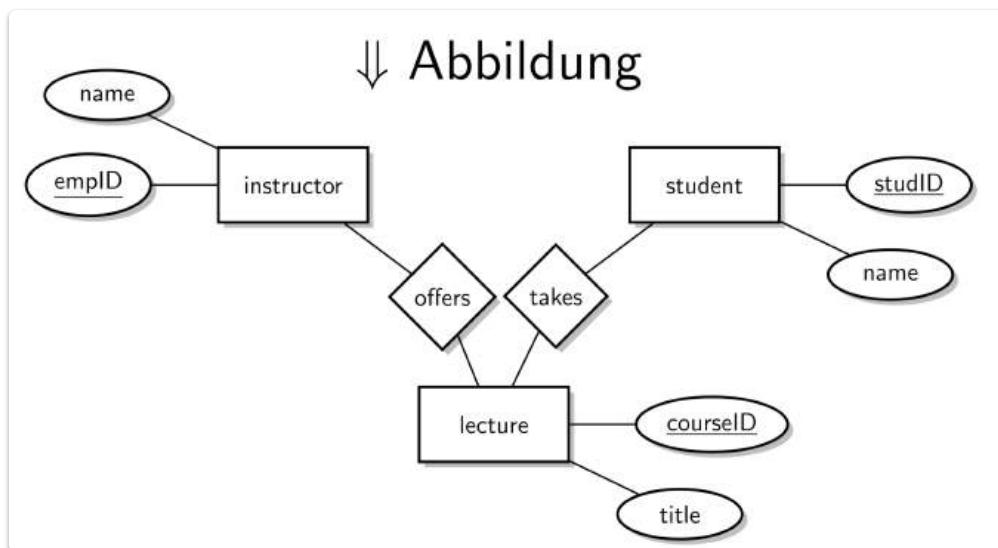
Beziehungsbeschreibung

- **Ziel:** Identifizierung und Beschreibung der Beziehungen zwischen den identifizierten Objekten (Entitäten).
- **Beispiel: Beziehung „prüfen“**

- **Beteiligte Objekte:**
 - Vortragende:r als Prüfer:in
 - Studierende:r als Prüfling
 - Vorlesung als Prüfungsstoff
- **Attribute der „prüfen“-Beziehung:**
 - Datum (Wann fand die Prüfung statt?)
 - Uhrzeit (Um welche Uhrzeit fand die Prüfung statt?)
 - Note (Welche Note wurde in der Prüfung erzielt?)

Schritt 2: Abbildung auf ein konzeptuelles Modell

- **Ziel:** Erstellung eines ER-Modells (Entity-Relationship-Modell) basierend auf den identifizierten Anforderungen.
- **Anforderungen (aus Schritt 1):**
 - Studierende nehmen an Vorlesungen teil.
 - Professor:innen bieten Vorlesungen an.
 - Studierende werden durch die Matrikelnummer eindeutig identifiziert.
- **Funktionale Anforderungen (Zusätzliche Anforderungen):**
 - Sekretär:in muss Noten eintragen können.



Schritt 3: Abbildung auf das relationale Modell

- **Ziel:** Überführung des ER-Modells in ein relationales Schema, das aus Tabellen (Relationen), Attributen und Schlüsseln besteht.
- **Resultierendes Relationales Modell (Beispiele):**
 - `student (studID: integer, name: string)`

- `studID` ist der Primärschlüssel (eindeutige Kennung für jeden Studenten).
 - `takes (studID: integer, courseID: integer)`
 - `studID` und `courseID` bilden zusammen den Primärschlüssel (ein Student kann ein spezifisches Fach nur einmal belegen).
 - `studID` ist ein Fremdschlüssel, der auf `student.studID` verweist.
 - `courseID` ist ein Fremdschlüssel, der auf `lecture.courseID` verweist.
 - `lecture (courseID: integer, title: string)`
 - `courseID` ist der Primärschlüssel (eindeutige Kennung für jede Vorlesung).
-

Schritt 4: Praktische Umsetzung und Implementierung

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

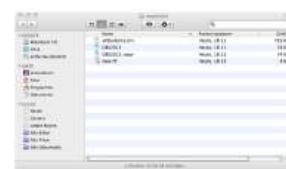
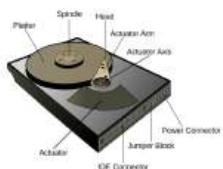
--hier Bild: Tabelle "student" mit Spalten "studID", "name" und Beispieldaten. Tabelle "takes" mit "studID", "courseID" und Beispieldaten. Tabelle "lecture" mit "courseID", "title" und Beispieldaten.--

- Tabellen einer DB (Beispieldaten):

student		takes		lecture	
studID	name	studID	courseID	courseID	title
26120	Pedersen	25403	5022	5001	DBS
25403	Hansen	26120	5001	5022	Belief and Knowledge
...

↓ Abbildung

Speicherseiten, Datenstrukturen, Indizes, Dateien, Geräte



- **Physische Speicherung:**

- Dies beinhaltet die Speicherung der Daten auf physischen Geräten.
- **Speicherseiten, Datenstrukturen, Indizes, Dateien, Geräte** spielen hier eine Rolle.

Zusammengefasst

1. Anforderungsanalyse

- *Mit was haben wir es zu tun?* (Was sind die Anforderungen und der Umfang des Datenbanksystems?)

2. Abbildung auf ein konzeptuelles Modell (konzeptueller Entwurf)

- *Welche Daten und Zusammenhänge müssen erfasst werden?* (Wie sollen die Daten und ihre Beziehungen abstrakt, d.h. unabhängig von einer spezifischen Datenbanktechnologie, dargestellt werden? Dies führt oft zu einem ER-Modell.)

3. Abbildung auf ein konkretes Modell (logischer Entwurf)

- *Wie müssen die Daten in einem bestimmten Modell strukturiert werden (hier: das relationale Modell)?* (Wie werden die Daten für ein spezifisches Datenbankmodell, z.B. das relationale Modell, organisiert und dargestellt? Dies beinhaltet die Definition von Tabellen, Attributen und Schlüsseln.)

4. Umsetzung und Implementierung (Physischer Entwurf)

- *Welche Anpassungen und Optimierungen sieht ein konkretes DBMS vor?* (Wie werden die logischen Schemata physisch auf Speichermedien abgebildet? Dies beinhaltet Aspekte wie Indizierung, Speicherorganisation und Dateiformate, um Performance und Speichereffizienz zu optimieren.)

Ein guter Entwurf vermeidet Redundanz und Unvollständigkeit. (Das bedeutet, dass Daten nicht unnötig mehrfach gespeichert werden sollten, um Inkonsistenzen zu vermeiden, und dass alle notwendigen Informationen vorhanden sind.)

Grundkonzepte des ER-Modells

Entität und Entitätstypen

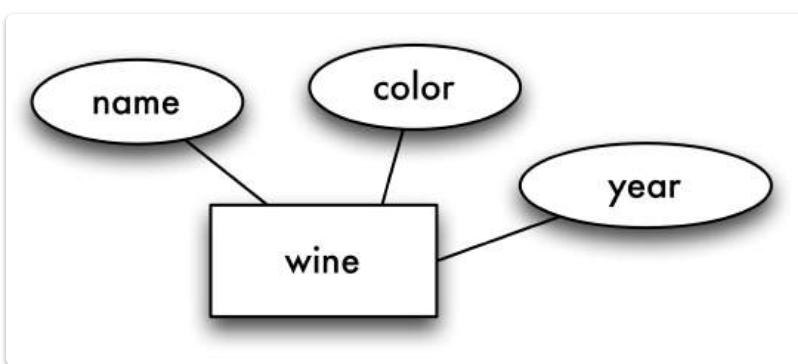
- **Entitäten** sind Objekte der realen Welt, über die wir Informationen abspeichern wollen.
 - Nur **Eigenschaften** der Entitäten können in einer Datenbank gespeichert werden (Beschreibung), nicht die Entitäten selbst.
- Entitäten werden in **Entitätstypen** eingeteilt.



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

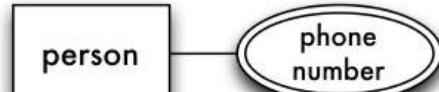
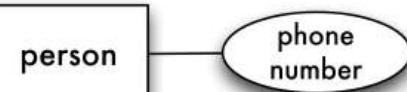
Attribute

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



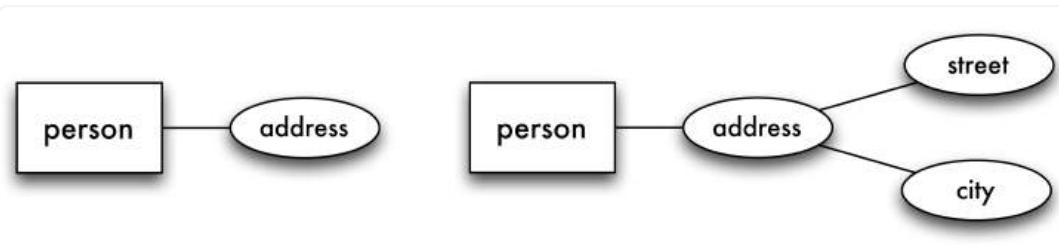
Single-valued (einwertige) vs. Multi-valued (mehrwertige) Attribute

- **Single-valued (einwertig):** Ein Attribut kann pro Entität nur einen einzigen Wert annehmen.
- **Multi-valued (mehrwertig):** Ein Attribut kann pro Entität mehrere Werte annehmen.
 - Beispiel: Eine Person kann mehrere Telefonnummern haben (oder eine einzige).



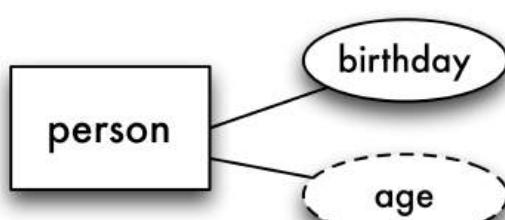
Simple (einfache) Attribute vs. Composite (zusammengesetzte) Attribute

- **Simple (einfache):** Ein Attribut, das nicht weiter unterteilt werden kann.
- **Composite (zusammengesetzte):** Ein Attribut, das aus mehreren einfacheren Attributen besteht.
 - Beispiel: Eine Adresse kann als String modelliert werden oder sich aus Straße und Stadt zusammensetzen.



Gespeicherte Attribute vs. derived (abgeleitete) Attribute

- **Gespeicherte Attribute:** Attribute, deren Werte direkt in der Datenbank abgelegt werden.
- **Derived (abgeleitete) Attribute:** Attribute, deren Werte aus anderen gespeicherten Attributen berechnet oder abgeleitet werden können. Sie werden nicht direkt gespeichert, sondern bei Bedarf ermittelt.
 - Zum Beispiel: Alter (kann aus dem Geburtsdatum abgeleitet werden).



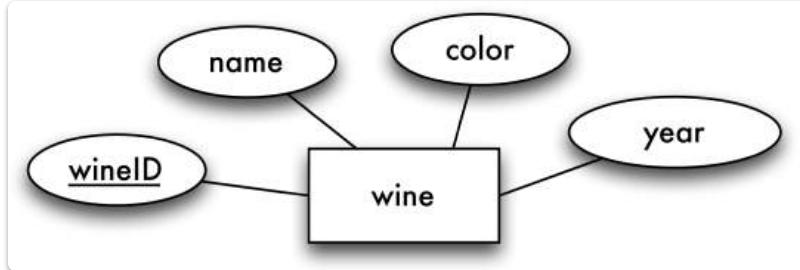
Schlüssel

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

- geht.)
- Gibt es mehrere Schlüsselkandidaten, so wird einer als **Primärschlüssel** ausgewählt.

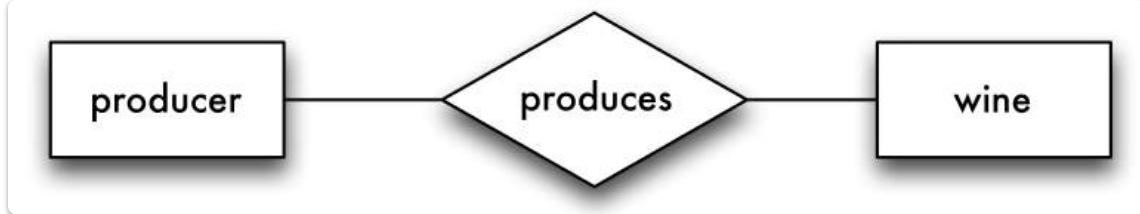
Primärschlüssel

- Attribute des Primärschlüssels werden durch Unterstrichen markiert.



Beziehung und Beziehungstyp

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



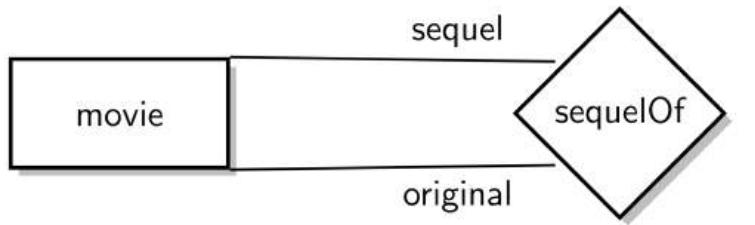
- Eine konkrete Verbindung zwischen zwei oder mehreren Entitäten wird **Beziehung (-instanz)** genannt.
- Eine **Beziehungsmenge** ist eine Menge von Beziehungsinstanzen.
- Die zwei Begriffe Beziehungsmenge (relationship set) und Beziehungstyp (relationship type) werden oft als Synonyme verwendet.

Mathematische Auffassung einer Beziehung

- Ein Beziehungstyp R zwischen den Entitätstypen E_1, E_2, \dots, E_n wird als **Relation** im mathematischen Sinne aufgefasst.
- Ausprägung** des Beziehungstyps R :
 - $R \subseteq E_1 \times E_2 \times \dots \times E_n$
- Ein Element $(e_1, e_2, \dots, e_n) \in R$ bezeichnet man als **Instanz** des Beziehungstyps.
 - Dabei ist $e_i \in E_i$ für alle $1 \leq i \leq n$.
- Anmerkung:** Diese Notation umfasst keine Attribute von Beziehungstypen.

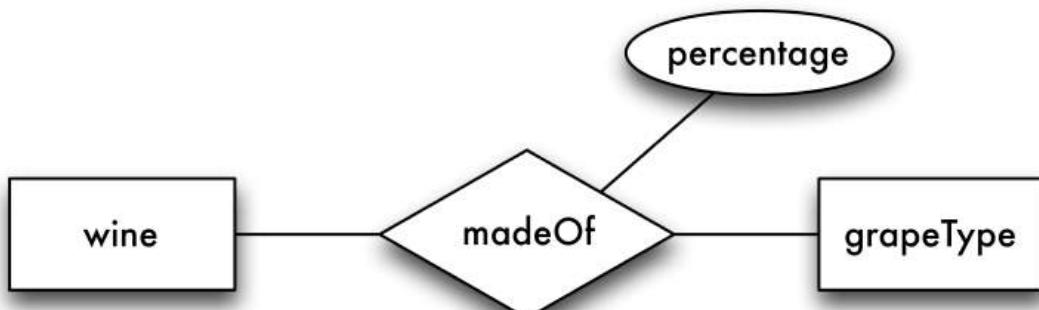
Rollennamen und Rekursive Beziehungstypen

- Rollennamen** sind optional und werden zur weiteren Charakterisierung einer Beziehung verwendet.
- Besonders sinnvoll bei einem **rekursiven Beziehungstyp**, bei dem ein Entitätstyp mehrfach an einem Beziehungstyp beteiligt ist.



Beziehungsattribute

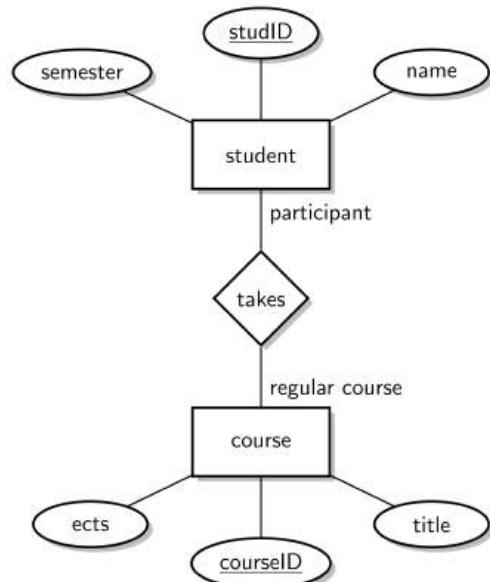
- Beziehungstypen können ebenfalls Attribute besitzen.



Beispiel mit Ablauf zum erstellen von einem ERD

Studierende nehmen an Vorlesungen teil

1. Entity → Entitytyp



2. Beziehung → Beziehungstyp

3. Attribute (Eigenschaften)

4. Primärschlüssel

5. Rollen

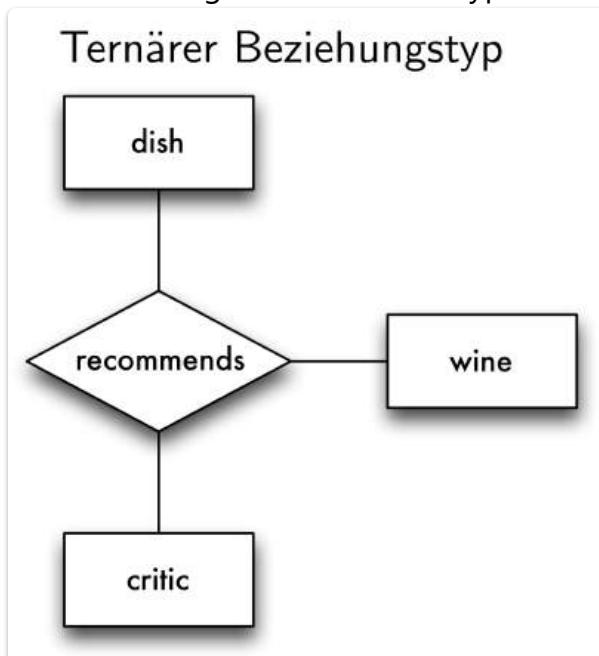
Eigenschaften von Beziehungstypen

Merkmale von Beziehungstypen

- **Stelligkeit bzw. Grad**
 - Beschreibt die Anzahl der beteiligten Entitätstypen an einer Beziehung.
 - Häufig: **binär** (zwei Entitätstypen beteiligt)
 - Weniger häufig: **ternär** (drei Entitätstypen beteiligt)
 - Allgemein: **n-stellig** (n Entitätstypen beteiligt)
- **Funktionalität / Kardinalität / Participation Constraints**
 - Beschreibt die Anzahl von Entitäten, die an einer Beziehung teilnehmen.
 - **Funktionalität (Chen-Notation):**
 - $1 : 1$ (Eins-zu-Eins)
 - $1 : N$ (Eins-zu-Viele)
 - $N : M$ (Viele-zu-Viele)
 - **Participation Constraints:**
 - **partiell:** Eine Entität muss nicht an der Beziehung teilnehmen.
 - **total:** Eine Entität muss an der Beziehung teilnehmen.
 - **Kardinalität ([min,max]-Notation):** Gibt die minimale und maximale Anzahl der Teilnahmen an einer Beziehung an.
 - $[min, max]$

Zwei- vs. mehrstellige Beziehungen

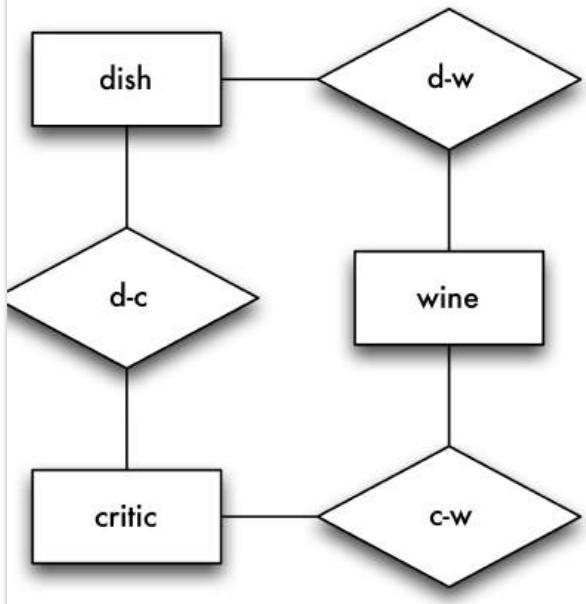
- **Ternärer Beziehungstyp:**
 - Eine Beziehung, die drei Entitätstypen miteinander verbindet.



- **Drei binäre Beziehungstypen:**

- Eine ternäre Beziehung kann auch durch drei binäre Beziehungen dargestellt werden, aber dies kann zu Informationsverlust führen oder komplexer sein.

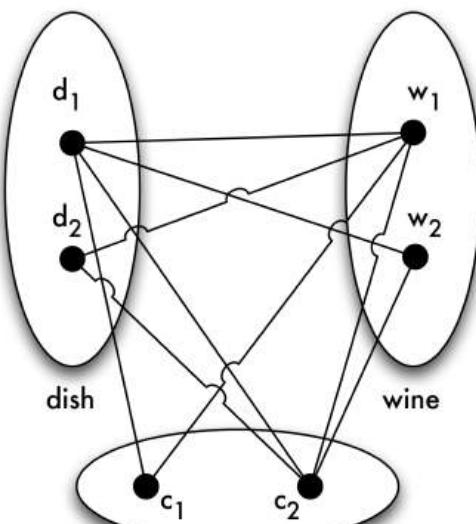
Drei binäre Beziehungstypen



Mehrstellige Beziehungstypen (Ausprägung)

- Zeigt die möglichen Instanzen bzw. Ausprägungen von mehrstelligen Beziehungen.

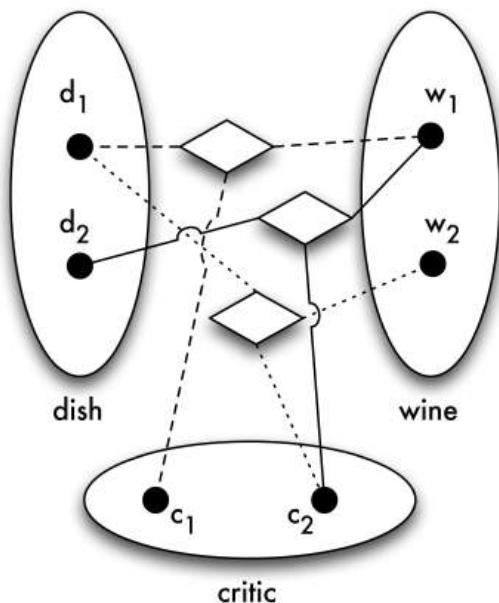
Binäre Beziehungstypen



Rekonstruierbare Beziehungen

- $d_1 - c_1 - w_1$
- $d_1 - c_2 - w_2$
- $d_2 - c_2 - w_1$
- aber auch: $d_1 - c_2 - w_1$

Ternärer Beziehungstyp



Mit binären Beziehungstypen können wir die folgende Beziehung rekonstruieren
 $d_1 - c_2 - w_1$
nicht jedoch mit einem ternären Beziehungstypen!

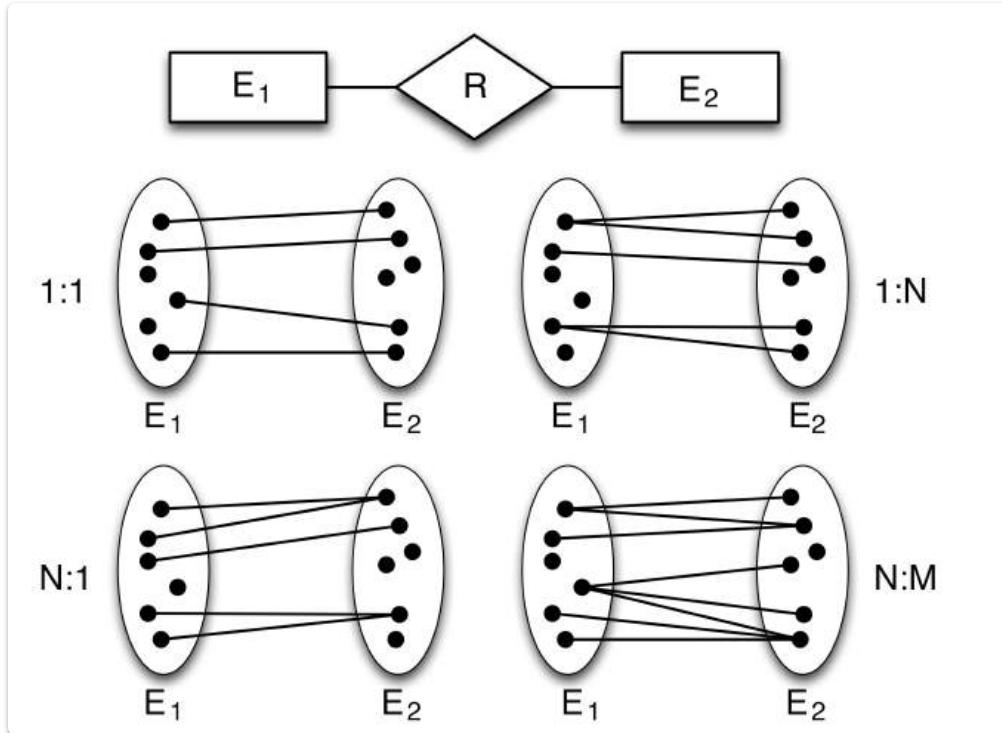
Merkmale von Beziehungstypen (Wiederholung und Ergänzung)

- **Stelligkeit bzw. Grad:**
 - Anzahl der beteiligten Entitätstypen.
 - Häufig: binär
 - Weniger häufig: ternär
 - Allgemein: n-stellig
- **Funktionalität / Kardinalität / Participation Constraints:**
 - Anzahl von Entitäten, die an einer Beziehung teilnehmen.
 - **Funktionalität (Chen-Notation):** Definiert, wie viele Instanzen eines Entitätstyps mit Instanzen eines anderen Entitätstyps in Beziehung stehen können.
 - 1 : 1 (Eins-zu-Eins)
 - 1 : N (Eins-zu-Viele)
 - N : M (Viele-zu-Viele)
 - **Participation Constraints:**
 - **partiell:** Eine Entität muss nicht an der Beziehung teilnehmen.
 - **total:** Eine Entität muss an der Beziehung teilnehmen.
 - **Kardinalität ([min,max]-Notation):** Gibt die minimale und maximale Anzahl der Teilnahmen an einer Beziehung an.
 - $[min, max]$

Chen-Notation (Funktionalität)

- Eine Beziehung R zwischen zwei Entitätstypen E_1 und E_2 wird in der Chen-Notation dargestellt.

- $R \subseteq E_1 \times E_2$



Funktionale Beziehungstypen

- $1 : 1$, $1 : N$ und $N : 1$ Beziehungen können als **partielle Funktionen** angesehen werden (oft auch als **totale Funktionen**).
 - **1:1 Beziehungstypen:** $R : E_1 \rightarrow E_2$ und $R^{-1} : E_2 \rightarrow E_1$ (jeder e_1 ist zu höchstens einem e_2 zugeordnet und umgekehrt)
 - **1:N Beziehungstypen:** $R^{-1} : E_2 \rightarrow E_1$ (für jedes e_2 gibt es höchstens ein e_1 , mit dem es in Beziehung steht)
 - **N:1 Beziehungstypen:** $R : E_1 \rightarrow E_2$ (für jedes e_1 gibt es höchstens ein e_2 , mit dem es in Beziehung steht)
 - Dies wird auch als **funktionale Beziehung** genannt.
- Die "Richtung" ist entscheidend!
 - Die Funktion geht immer vom "N"-Entitätstyp zum "1"-Entitätstyp, d.h., die Seite mit der "vielen" Kardinalität bildet auf die Seite mit der "einen" Kardinalität ab.
- In dieser Vorlesung wird zwischen partiellen Funktionen (\rightarrow) und totalen Funktionen (\rightarrow) unterschieden. Es wird jedoch vereinfacht geschrieben.

Notation funktionaler Beziehungstypen

- **1:N Beziehungstyp:**



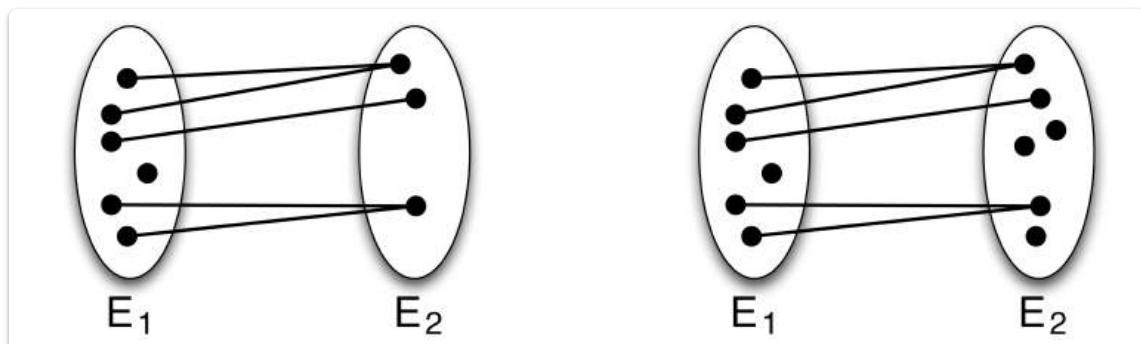
- Hier kann ein Produzent viele Weine produzieren, aber ein Wein wird nur von einem Produzenten produziert.
- **1:1 Beziehungstyp:**



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

Participation Constraints

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



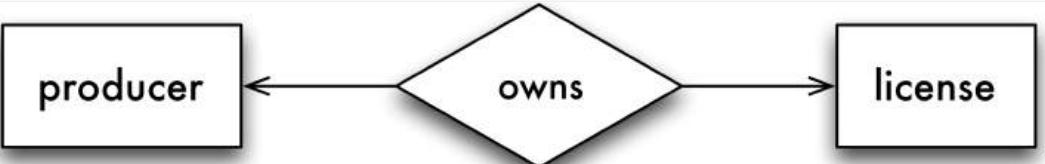
Graphische Darstellung von Participation Constraints

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





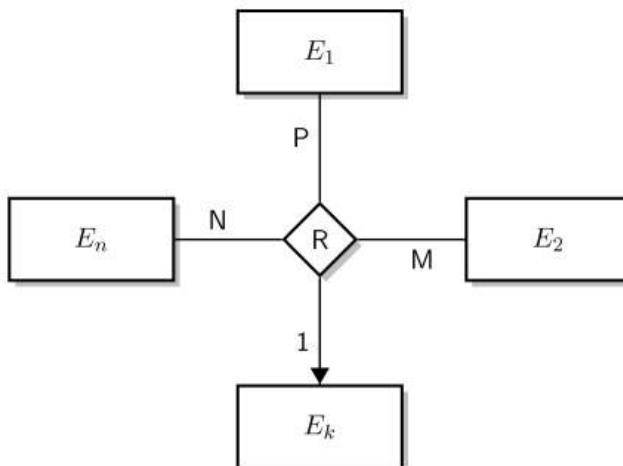
- Das bedeutet, jeder Produzent muss Weine produzieren und jeder Wein muss von einem Produzenten produziert werden.
- 1:1 Beziehungstyp mit partieller Partizipation:**



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

Funktionalitäten bei n-stelligen Beziehungstypen

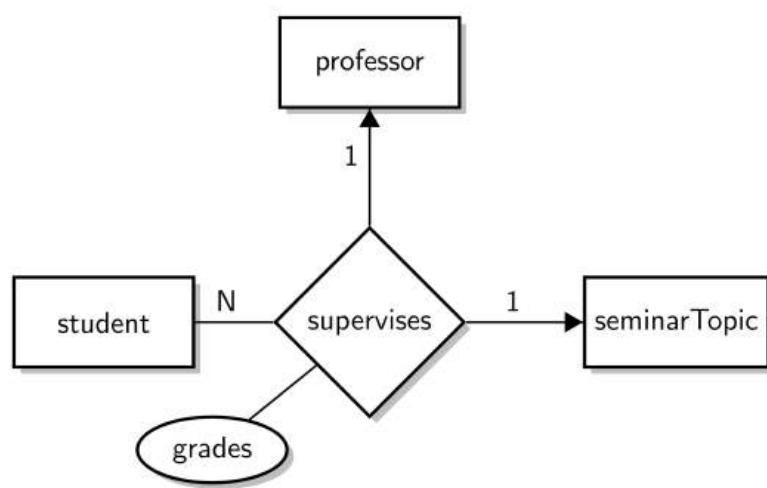
- Bei n-stelligen Beziehungstypen gibt es nicht nur die einfachen 1:1, 1:N oder N:M Funktionalitäten, sondern komplexere Zuordnungen.
- Eine n-stellige Beziehung R ist ein Teilprodukt der Entitätstypen:
 - $R \subseteq E_1 \times E_2 \times \dots \times E_k \times E_{k+1} \times \dots \times E_n$



$$R : E_1 \times E_2 \times \dots \times E_{k-1} \times E_{k+1} \times \dots \times E_n \rightarrow E_k$$

- Anmerkung zur Notation im Allgemeinen:**
 - Die Verwendung von Pfeilen und 1, N, M etc. sind äquivalent.
 - Beide Darstellungen sind richtig, wobei eine eben manchmal hilfreicher sein kann, um die Funktionalität intuitiver zu erfassen.

Beispielbeziehungstyp: "supervises" (Betreut)



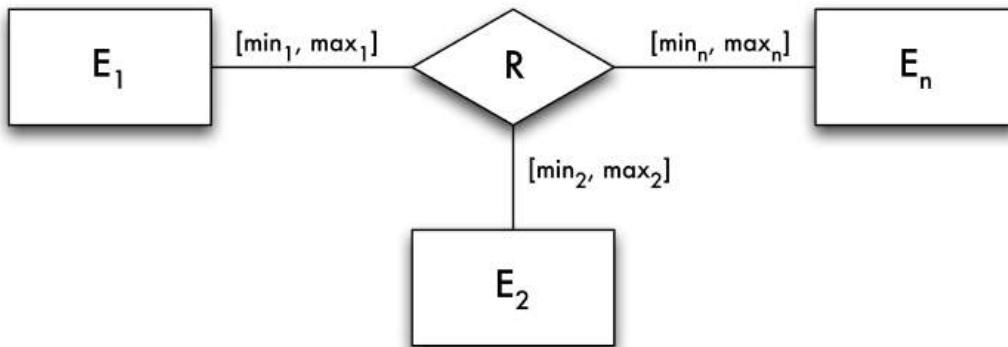
- Dieser ternäre Beziehungstyp "supervises" verbindet:
 - professor
 - student
 - seminarTopic
- Zusätzlich gibt es ein Beziehungsattribut `grades` (Noten).
- Die Funktionalitäten sind hier komplexer als bei binären Beziehungen.
- Der Beziehungstyp `supervises` kann als Abbildung dargestellt werden:
 - `supervises: professor × student → seminarTopic`
 - Dies bedeutet, dass eine bestimmte Kombination aus Professor und Student **maximal einem** Seminar-Thema zugeordnet ist.
 - `supervises: professor × seminarTopic → student`
 - Dies bedeutet, dass eine bestimmte Kombination aus Professor und Seminar-Thema **maximal einem** Studenten zugeordnet ist.

Merkmale von Beziehungstypen (Wiederholung)

- **Stelligkeit bzw. Grad:**
 - Anzahl der beteiligten Entitätstypen.
 - Häufig: binär
 - Weniger häufig: ternär
 - Allgemein: n-stellig
- **Funktionalität / Kardinalität / Participation Constraints:**
 - Anzahl von Entitäten, die an einer Beziehung teilnehmen.
 - Funktionalität (Chen-Notation): $1 : 1$, $1 : N$, $N : M$.
 - Participation Constraints: partiell oder total.
 - **Kardinalität ([min,max]-Notation):** $[min, max]$

$[min, max]$ -Notation (Kardinalität)

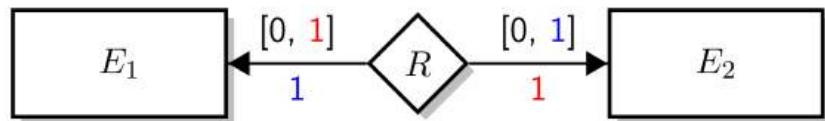
- Die $[min, max]$ -Notation schränkt ein, wie oft ein Entity eines Entitätstyps an einer Beziehung teilnehmen kann.



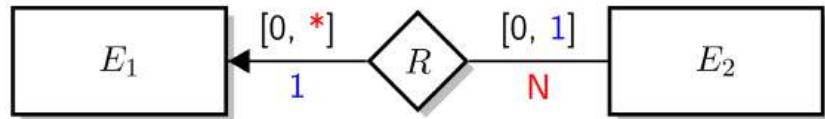
- Für jedes $e_i \in E_i$ gibt es:
 - mindestens** min_i Instanzen des Beziehungstyps der Art (\dots, e_i, \dots) und
 - höchstens** max_i viele Instanzen des Beziehungstyps der Art (\dots, e_i, \dots).
- Die Kardinalitätsbedingung ist also: $min_i \leq |\{r \in R \mid r \text{ enthält } e_i\}| \leq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

Chen vs min, max

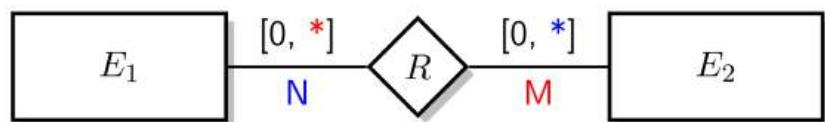
1:1 Beziehungstyp



1:N Beziehungstyp



N:M Beziehungstyp

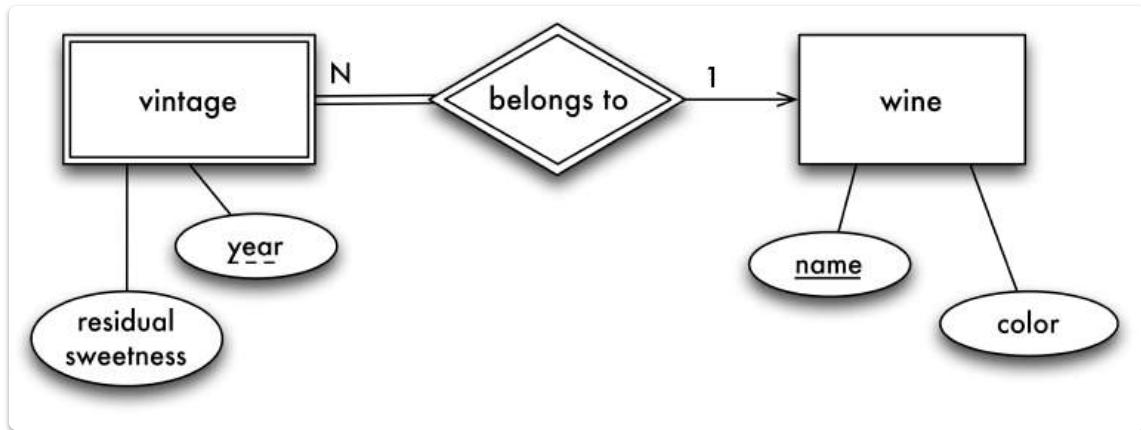


Abgrenzung von Funktionalität und Kardinalität

- In der Literatur werden die Begriffe **Funktionalität** und **Kardinalität** oft synonym verwendet.
- In dieser Vorlesung wird unterschieden:
 - **Funktionalität:**
 - Wird im Zusammenhang mit der **Chen-Notation** verwendet (z.B. $1 : 1$, $1 : N$, $N : M$).
 - **Kardinalität:**
 - Wird im Zusammenhang mit der **[min,max]-Notation** verwendet (z.B. $[0, 1]$, $[1, *]$).

Zusätzliche Konzepte

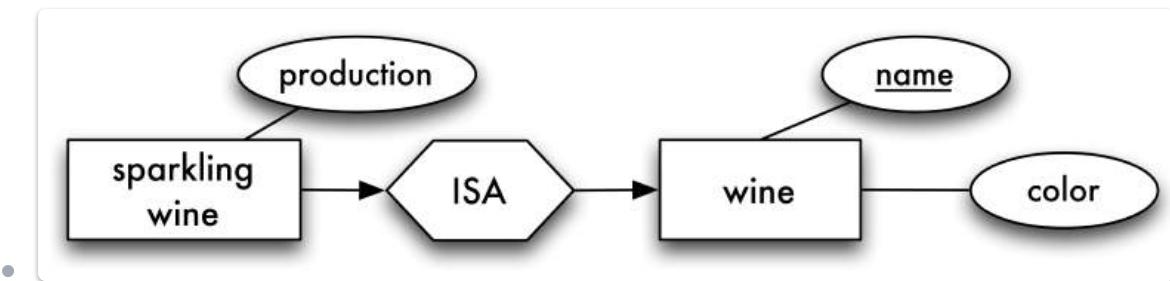
Schwache Entitätstypen



- Hier hat "vintage" die Attribute "year" und "residual sweetness".
- "wine" hat die Attribute "name" und "color".
- Die Funktionalität ist $N : 1$ von "vintage" zu "wine".
- Die Existenz eines **schwachen Entitys** (z.B. `vintage`) hängt von der Existenz eines **starken Entitys** (identifying/owning entity, z.B. `wine`) ab.
 - Sie sind durch eine **identifizierende Beziehung** verbunden.
- **Eigenschaften von schwachen Entitätstypen:**
 - **Totale Partizipation** des schwachen Entitätstyps: Ein schwaches Entity kann ohne die Beziehung zu seinem starken Entity nicht existieren. (Im Beispiel: Eine "vintage" (Jahrgang) existiert nur im Kontext eines "wine" (Weins)).
 - Nur in Kombination mit $1 : N$ ($N : 1$) oder selten auch $1 : 1$ Beziehungstypen.
 - Der starke Entitätstyp ist immer auf der "1"-Seite der Beziehung (also der Entitätstyp, von dem die schwache Entität abhängt).
- **Schlüsselattribute bei schwachen Entitätstypen:**
 - Schwache Entitäten sind oft nur mit dem **Schlüssel des entsprechenden starken Entitys eindeutig identifizierbar**.
 - Die Schlüsselattribute des schwachen Entitytyps werden gestrichen unterstrichen (sogenannte **Teilschlüssel**).
 - Im Beispiel wäre der "year" des "vintage" ein Teilschlüssel, da ein Jahrgang nur in Verbindung mit einem bestimmten Wein eindeutig ist (z.B. "Chardonnay 2020" vs. "Merlot 2020").

Der ISA-Beziehungstyp (Spezialisierung/Generalisierung)

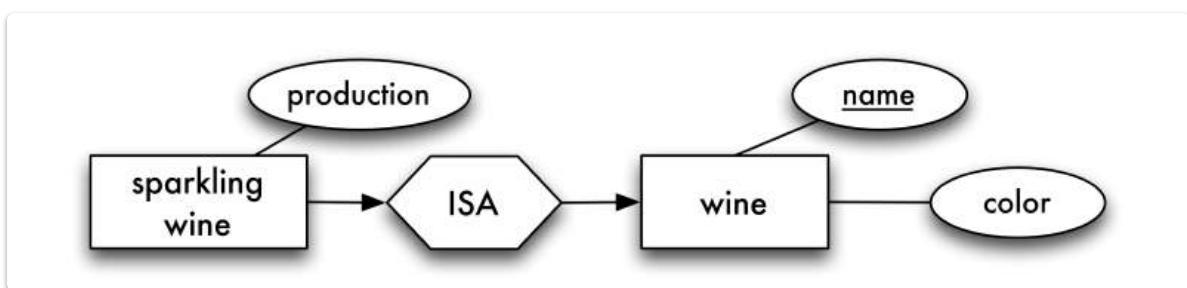
- **Spezialisierung und Generalisierung** (Vererbung) wird durch den **ISA-Beziehungstyp** ausgedrückt.



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

Eigenschaften von ISA-Beziehung

Grundlagen

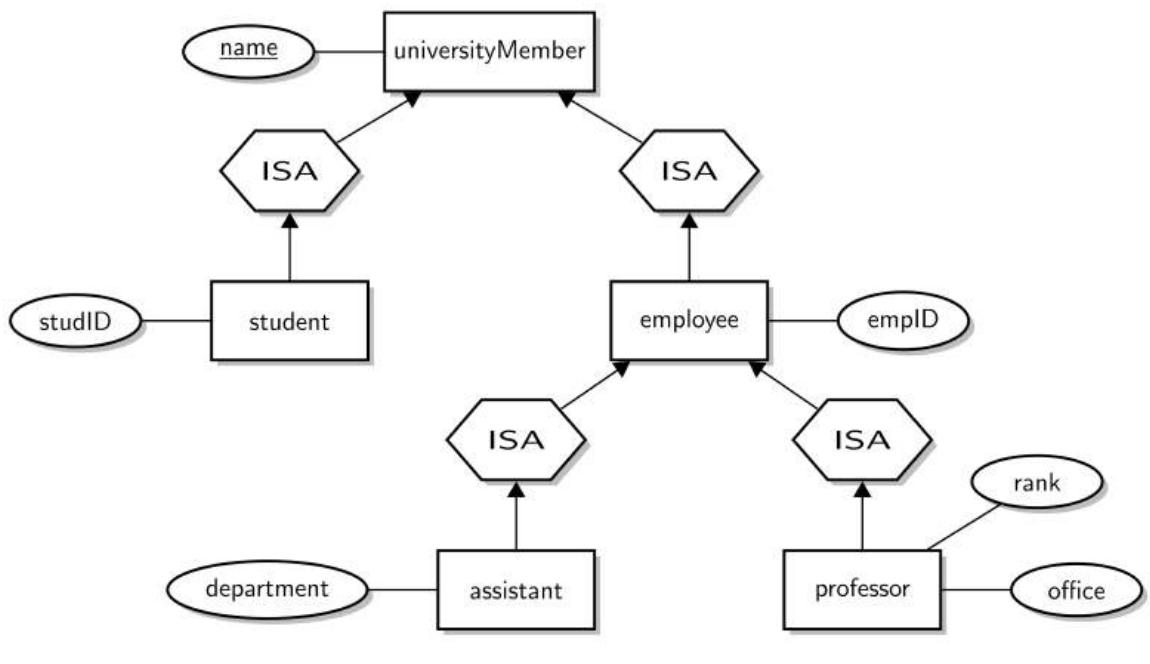


- Jeder Schaumwein ist genau einem Wein zugeordnet.
- Schaumweine werden durch die funktionale ISA-Beziehung identifiziert. (Eine Spezialisierung, bei der die Unterklasse eine Teilmenge der Oberklasse ist und alle Instanzen der Unterklasse auch Instanzen der Oberklasse sind.)
- Nicht jeder Wein ist zugleich ein Schaumwein. (Die Beziehung ist nicht umkehrbar; es gibt Weine, die keine Schaumweine sind.)
- Attribute des Entitytyps *wine* werden an den Entitytyp *sparkling wine* vererbt. (Das bedeutet, *sparkling wine* erbt alle Attribute von *wine*, wie z.B. *name* und *color*).

Kardinalität bei ISA-Beziehungen

- Die Kardinalitäten sind $ISA(E_1)[1, 1], E_2[0, 1]$
 - Jede Instanz von E_1 (*sparkling wine*) nimmt genau einmal an der Beziehung teil. (Jeder Schaumwein ist genau ein Wein.)
 - Instanzen von E_2 (*wine*) nehmen maximal einmal an der Beziehung teil. (Ein Wein kann ein Schaumwein sein, muss aber nicht.)

Beispiel: Universitätsbeispiel (überlappende Spezialisierung)



Dieses Beispiel zeigt eine hierarchische Struktur, bei der:

- Ein *university member* kann entweder ein *student* oder ein *employee* sein.
- Ein *employee* kann weiter als *assistant* oder *professor* spezialisiert werden.
- Es handelt sich um eine **überlappende Spezialisierung**, d.h., eine Instanz der Oberklasse kann zu mehreren Unterklassen gehören. (Beispiel: Ein Mitarbeiter könnte gleichzeitig ein Student sein, z.B. ein wissenschaftlicher Mitarbeiter, der noch eingeschrieben ist.)

Spezielle Eigenschaften des ISA-Beziehungstyps (Zusätzliche Konzepte im ER-Modell)

Überlappende Spezialisierung

- **Definition:** Ein Entity (eine Instanz der Oberklasse) kann zu mehreren spezialisierten Entitymengen (Unterklassen) gehören.
 - **Beispiel:** Eine Person kann gleichzeitig *Student* und *Mitarbeiter* sein.
- **Darstellung im ER-Modell:** Separate ISA-Symbole werden verwendet, um die möglichen Mehrfachzugehörigkeiten zu kennzeichnen.

Disjunkte Spezialisierung

- **Definition:** Ein Entity (eine Instanz der Oberklasse) kann zu höchstens einer spezialisierten Entitymenge (Unterklasse) gehören.
 - **Beispiel:** Ein Fahrzeug kann entweder ein *Auto* oder ein *Motorrad* sein, aber nicht beides gleichzeitig.
- **Darstellung im ER-Modell:** Ein gemeinsames ISA-Symbol wird verwendet, um anzudeuten, dass sich die Unterklassen gegenseitig ausschließen.

Attribute und Beziehungstypen bei Spezialisierung/Generalisierung

Vererbung bei spezialisierten Entitytypen

Spezialisierte Entitytypen (Unterklassen) **erben**:

- **Attribute** von weniger spezialisierten Entitytypen (Oberklassen).
 - Beispiel: Ein *Student* erbt Attribute wie *Name* und *Adresse* von der Oberklasse *Person*.
- **Teilnahme an Beziehungstypen** von weniger spezialisierten Entitytypen.
 - Beispiel: Wenn *Person* an der Beziehung *wohnt in* (*Ort*) teilnimmt, dann nimmt auch *Student* an dieser Beziehung teil.

Zusätzliche Eigenschaften spezialisierter Entitytypen

Spezialisierte Entitytypen können:

- **Zusätzliche Attribute** haben, die nur für sie relevant sind.
 - Beispiel: *Student* hat das zusätzliche Attribut *Matrikelnummer*, das *Person* nicht hat.
- **An Beziehungstypen teilnehmen**, an denen die weniger spezialisierten Entitytypen nicht teilnehmen.
 - Beispiel: *Student* nimmt an der Beziehung *studiert* (*Studiengang*) teil, während *Person* nicht unbedingt an dieser Beziehung teilnehmen muss.

Participations-Constraints (Teilnahmebedingungen)

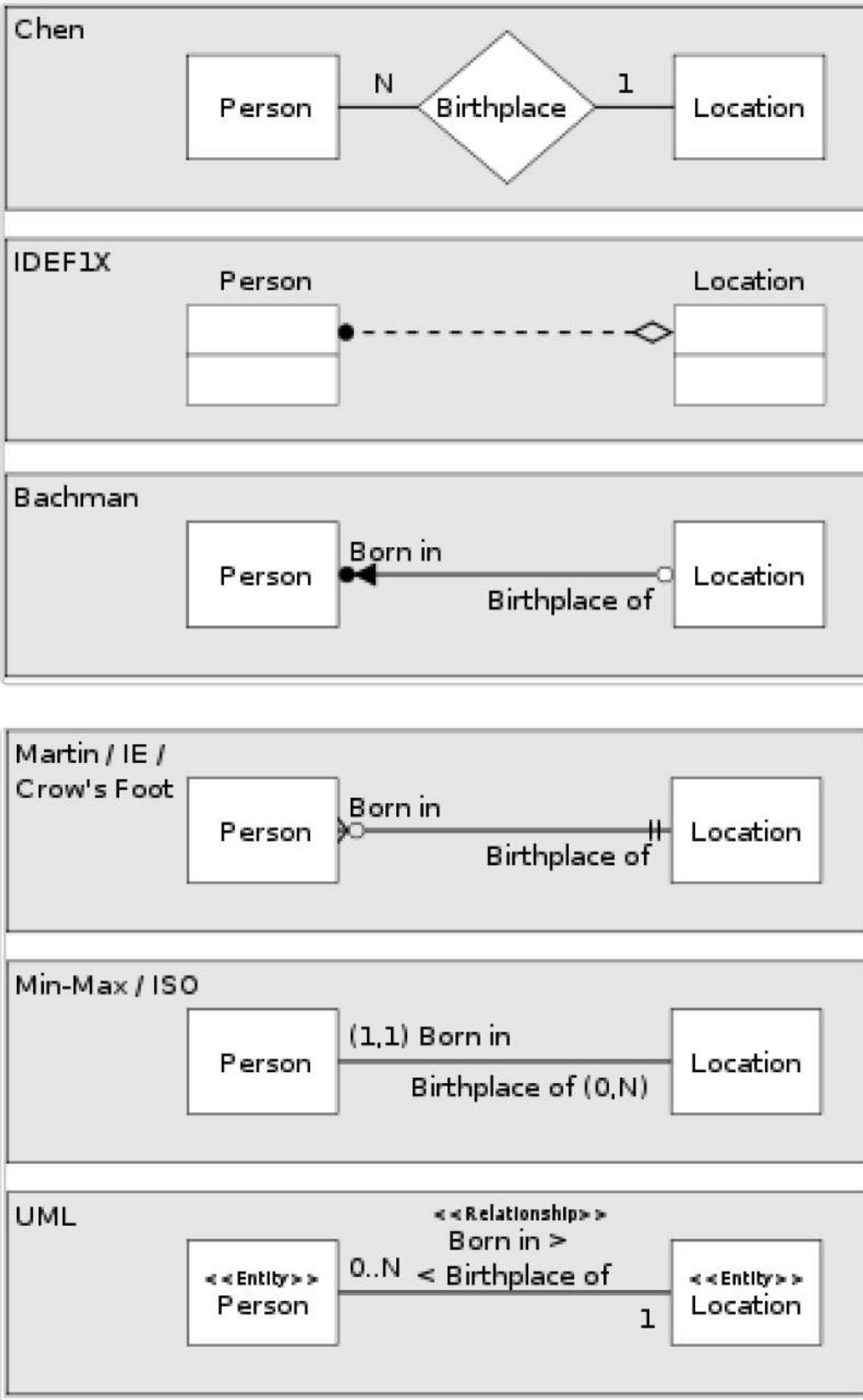
Totale Generalisierung/Spezialisierung

- **Definition:** Jeder weniger spezialisierte Entitytyp (Instanz der Oberklasse) **muss** zu einem spezialisierteren Entitytyp (mindestens einer Unterklasse) gehören.
 - Das bedeutet, jede Instanz der Oberklasse muss einer der definierten Spezialisierungen zugeordnet werden.
- **Notation im ER-Modell:** Eine **doppelte Linie** unter dem ISA-Symbol.

Partielle Generalisierung/Spezialisierung (Default)

- **Definition:** Jeder weniger spezialisierte Entitytyp (Instanz der Oberklasse) **kann** (aber muss nicht) zu einem spezialisierten Entitytyp (einer der Unterklassen) gehören.
 - Das bedeutet, es kann Instanzen der Oberklasse geben, die keiner der definierten Spezialisierungen zugeordnet werden.
- **Notation im ER-Modell:** Eine **einfache Linie** unter dem ISA-Symbol (dies ist der Standardfall, wenn nicht anders angegeben).

Alternative Notationen



Relationen aus Grundkonzepten ableiten

Entwurfsanmerkungen für ER-Modelle

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

Relationale Modellierung aus ER-Modell

Entitätstypen

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

Grundsätzliches Vorgehen für Entitätstypen

Für jeden Entitätstyp wird eine Relation erstellt:

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

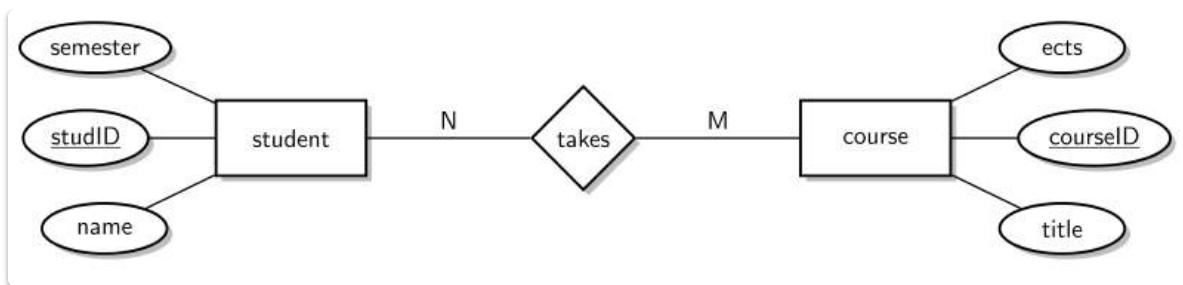
Notation von Relationenschemata

Beispiel:

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

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

Abbildung von N:M-Beziehungstypen



Grundsätzliches Vorgehen für N:M-Beziehungstypen

- Ein neues Relationenschema wird erstellt, das alle Attribute des Beziehungstyps enthält.
- Alle Primärschlüssele der beteiligten Entitätstypen werden übernommen.
- Die "importierten" Schlüsselattribute der beteiligten Entitätstypen werden **Fremdschlüssel** genannt.

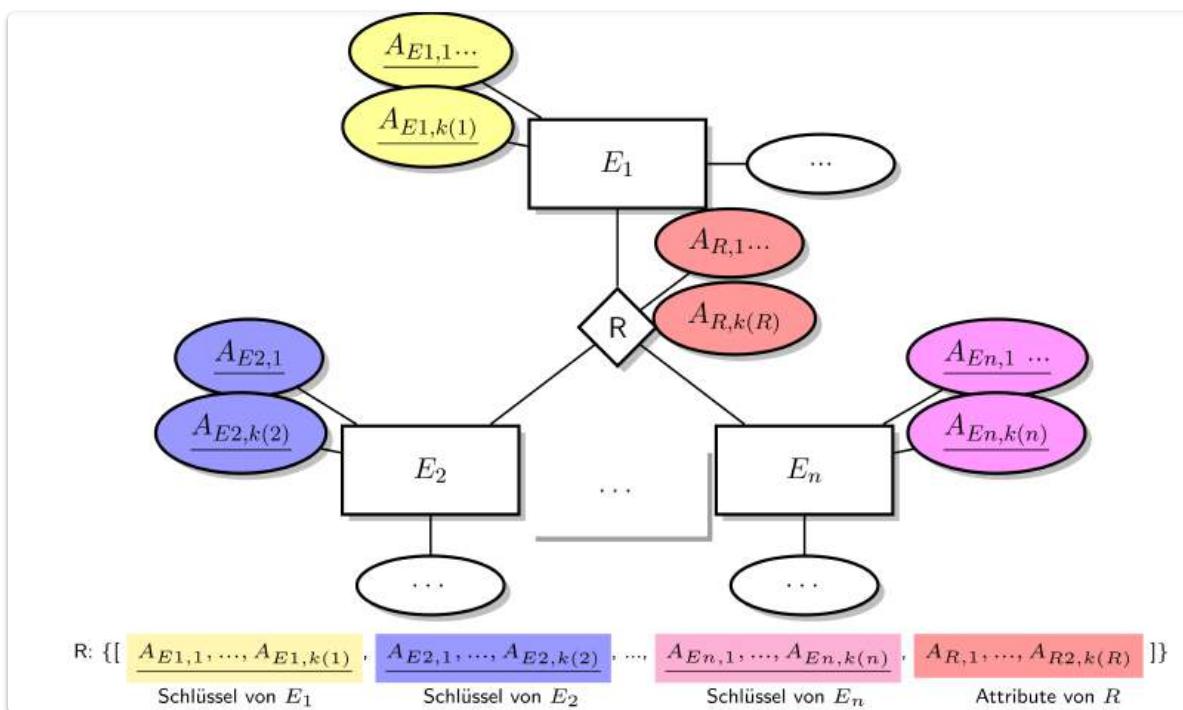
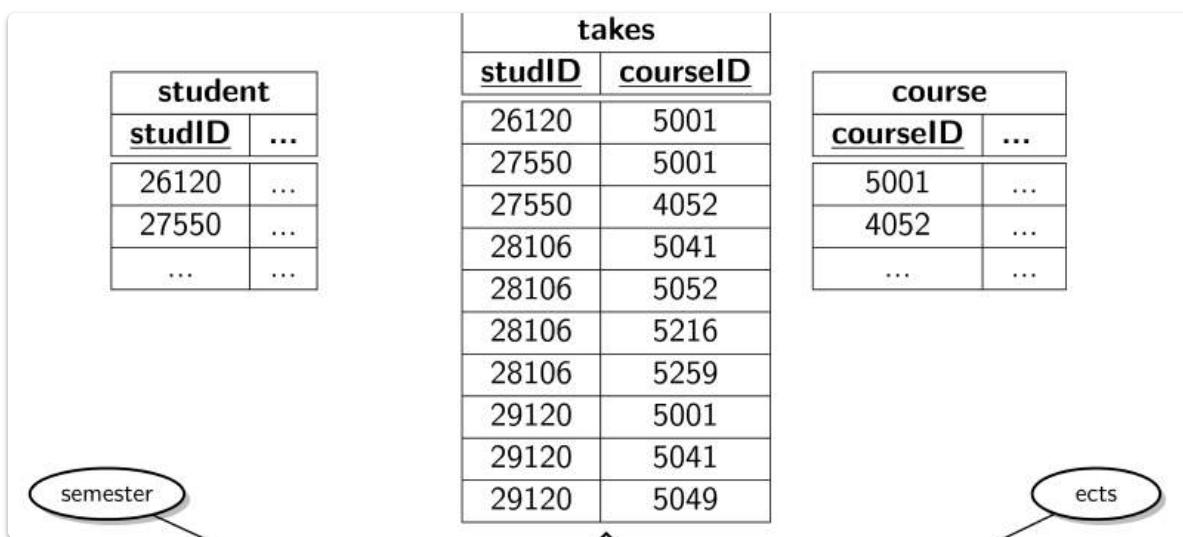
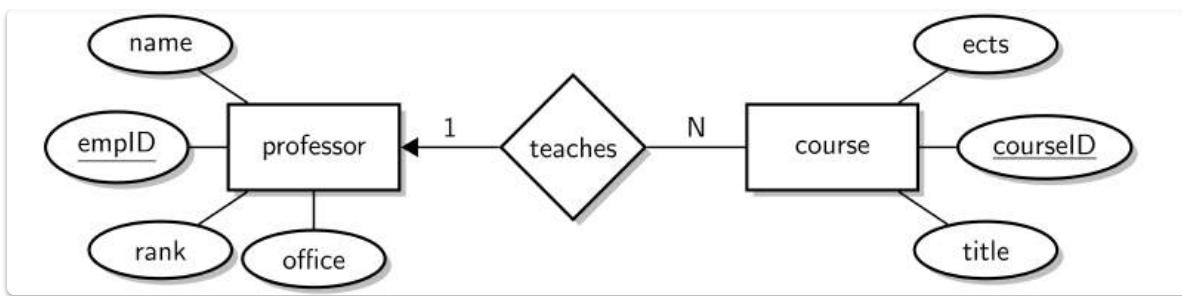


Abbildung von 1:N-Beziehungstypen



Grundsätzliches Vorgehen für 1:N-Beziehungstypen

- Ein neues Relationenschema wird mit allen Attributen des Beziehungstyps erstellt.
- Alle Primärschlüsselelemente der beteiligten Entitätstypen werden übernommen.
- Der Primärschlüssel der N-Seite (die Seite mit der Mehrfachbeziehung) wird zum Schlüssel im neuen Relationenschema.

Initialentwurf

- `course : {{ courseID, title, ects }}`
- `professor : {{ empID, name, rank, office }}`
- `teaches : {{ courseID \rightarrow course, empID \rightarrow professor }}`

Verbesserung durch Zusammenlegung (Finale Abbildung)

Bei 1:N-Beziehungen kann der Primärschlüssel der 1-Seite (in diesem Fall `professor`) direkt in die Relation der N-Seite (hier `course`) als Fremdschlüssel integriert werden. Dies vermeidet die Notwendigkeit einer separaten Relation für die Beziehung selbst.

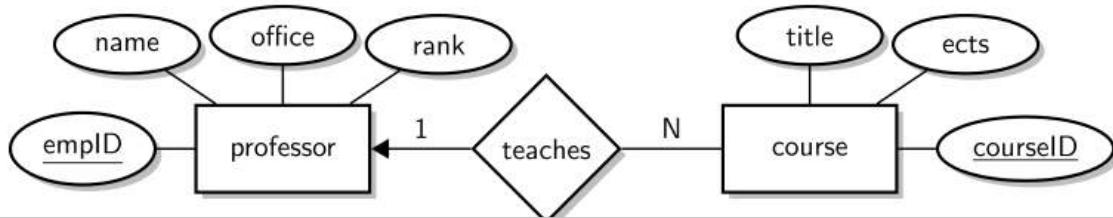
- `course : {{ courseID, title, ects, taughtBy \rightarrow professor }}`
- `professor : {{ empID, name, rank, office }}`

`taughtBy` ist ein **Fremdschlüssel** und referenziert den Primärschlüssel der Relation `professor`. Die Werte von `taughtBy` entsprechen den Werten von `empID` in der Relation `professor`.

Relationen mit denselben Schlüsseln können und sollten immer kombiniert werden, aber nur diese und keine anderen!

professor			
emplID	name	rank	office
2125	Socrates	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

course			
courselID	title	ects	taughtBy
5001	DBS	4	2137
5041	Robotics	4	2125
5043	Software Engineering	3	2126
5049	Ethics	2	2125
4052	Logic	4	2125
5052	Theory of Science	3	2126
5216	Bioethics	2	2126
5259	Chemistry	2	2133
5022	Belief and Knowledge	2	2134
4630	Physics	4	2137

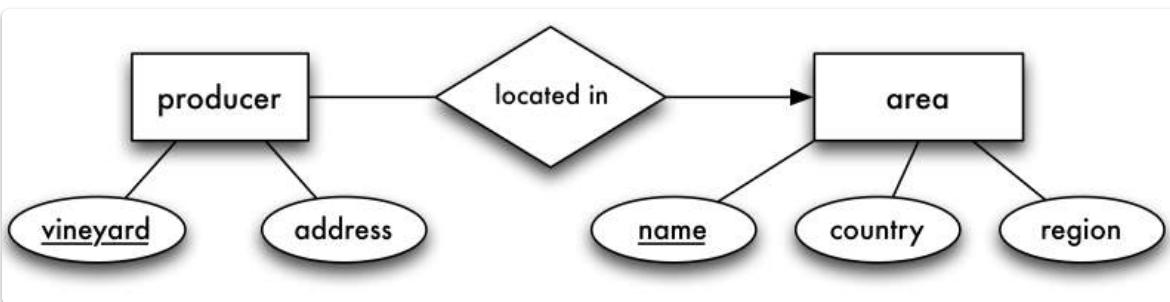


Achtung: Das funktioniert nicht

professor				
emplID	name	rank	office	teaches
2125	Socrates	C4	226	5041
2125	Socrates	C4	226	5049
2125	Socrates	C4	226	4052
...
2134	Augustinus	C3	309	5022
2136	Curie	C4	36	??
...

course		
courselID	title	ects
5001	DBS	4
5041	Robotics	4
5043	Software Engineering	3
5049	Ethics	2
4052	Logic	4
5052	Theory of Science	3
5216	Bioethics	2
5259	Chemistry	2
5022	Belief and Knowledge	2
4630	Physics	4

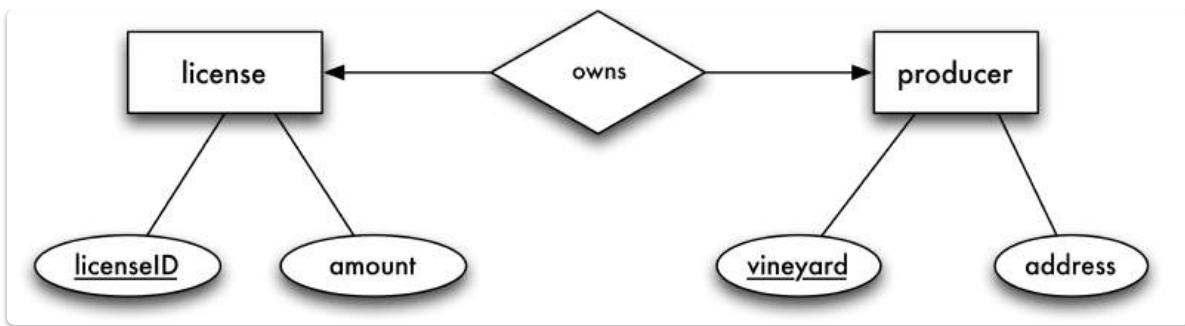
Zusammenfassung: N:1-Beziehungstypen



Wie bildet man dieses ER-Diagramm auf Relationen ab?

- producer : {{ vineyard, address, locatedIn \$\rightarrow\$ area }} (Der Fremdschlüssel zur 1-Seite (area) wird in die N-Seite (producer) integriert.)
- area : {{ name, country, region }}

1:1-Beziehungstypen



Unterscheidet sich die Herangehensweise hier zu einem 1:N-Beziehungstypen?

Ja, bei 1:1-Beziehungen gibt es mehr Flexibilität bei der Wahl, wo der Fremdschlüssel platziert wird.

Grundsätzliches Vorgehen für 1:1-Beziehungstypen

- Ein neues Relationenschema mit allen Attributen des Beziehungstyps wird erstellt (dies ist eine Option, aber nicht zwingend notwendig, wie weiter unten gezeigt wird).
- Alle Primärschlüsselelemente der beteiligten Entitätstypen werden übernommen.
- Irgendein Primärschlüssel des involvierten Entitätstyps wird der Schlüssel im neuen Relationenschema (wenn eine separate Relation für die Beziehung erstellt wird).

Initialentwurf

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

Verbesserung durch Zusammenfassung

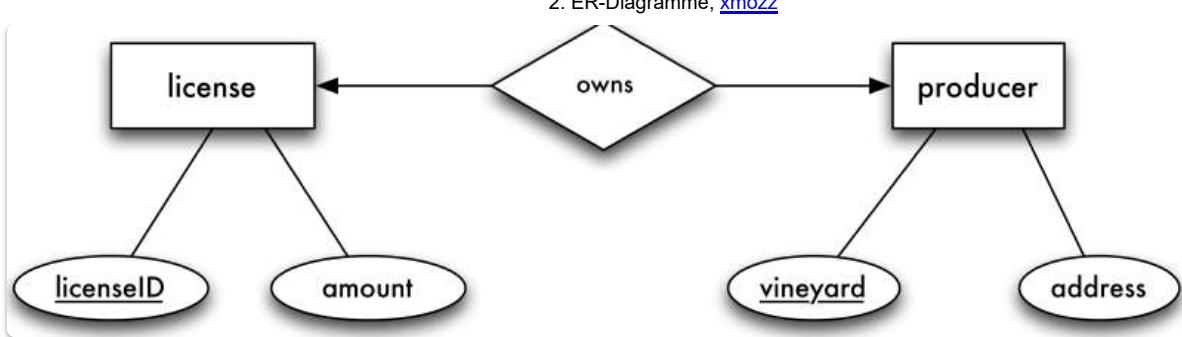
- **license:** {[licenseID, amount, ownedBy → producer]}
- **producer:** {[vineyard, address]}

oder

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

Am besten erweitert man eine Relation mit **totaler Partizipation**.

Warum nicht eine einzige Relation für 1:1-Beziehungen?



Problem bei der Zusammenführung in eine einzige Relation

Eine Zusammenführung von `license` und `producer` in eine einzige Relation (z.B. `producer` mit `licenseID` und `amount` als Attribute) wäre nur korrekt bei **totaler Partizipation** ((1, 1)) beider beteiligter Entitätstypen.

- **Totale Partizipation** bedeutet: Jede Instanz des einen Entitätstyps muss mit genau einer Instanz des anderen Entitätstyps in Beziehung stehen.
 - Beispiel: Jeder `producer` besitzt genau eine `license` UND jede `license` gehört genau einem `producer`.

producer	vineyard	address	licenseID	amount
	Rotkäppchen	Freiberg	42-007	10.000
	Weingut Müller	Dagstuhl	⊥	⊥
	⊥	⊥	42-003	100.000

Was passiert bei partieller Partizipation?

- **Partielle Partizipation** beider Entitätstypen (z.B. ein `producer` hat keine `license` oder eine `license` gehört keinem `producer`):
 - Führt zu **Null-Werten** in allen Attributen, die den nicht beteiligten Entitätstyp repräsentieren würden (z.B. `licenseID` und `amount` wären `NULL` für einen `producer` ohne Lizenz).
 - Die Bestimmung eines Primärschlüssels ist nicht mehr eindeutig oder schwierig, da Nullwerte auftreten können und die Identität nicht immer gegeben ist.
 - Führt zu **Speicherplatzverschwendungen** (unnötige `NULL`-Werte belegen Speicherplatz).

Zusammenfassend: Eine einzige Relation für 1:1-Beziehungen ist nur effizient und korrekt, wenn eine totale Partizipation von beiden Seiten der Beziehung vorliegt. Andernfalls sollte man die Entitäten getrennt halten und den Fremdschlüssel in eine der beiden Relationen platzieren, um Nullwerte zu minimieren.

Zusammenfassung: Beziehungstypen auf Relationen abbilden

M:N-Beziehungstyp

- **Neue Relation** mit Attributen des Beziehungstyps erstellen.

- Attribute hinzufügen, die die Primärschlüssel der involvierten Entitätstypen referenzieren (*Fremdschlüssel*).
- **Primärschlüssel:** Menge der Fremdschlüsselelemente (Der Primärschlüssel dieser neuen Relation setzt sich aus den Primärschlüsseln der Entitäten zusammen, die an der M:N-Beziehung beteiligt sind. Dies stellt die Eindeutigkeit jeder Beziehung sicher).

1:N-Beziehungstyp

- Attribut zur Relation des Entitätstyps auf der „N“-Seite hinzufügen:
 - *Fremdschlüssel*, der den Primärschlüssel des Entitätstyps auf der „1“-Seite referenziert (Der Fremdschlüssel auf der "N"-Seite verweist auf den Primärschlüssel der "1"-Seite, um die Beziehung zwischen den Entitäten herzustellen).
 - Attribute des Beziehungstyps hinzufügen (falls vorhanden).

1:1-Beziehungstyp

- Attribut zur Relation eines der involvierten Entitätstypen hinzufügen:
 - *Fremdschlüssel*, der den Primärschlüssel des Entitätstyps der anderen Seite referenziert (Der Fremdschlüssel kann zu einer der beiden beteiligten Entitäten hinzugefügt werden, da die Beziehung in beide Richtungen eindeutig ist. Es ist oft sinnvoll, den Fremdschlüssel zu der Entität hinzuzufügen, die seltener an der Beziehung beteiligt ist oder aus logischen Gründen besser passt).
 - Attribute des Beziehungstyps hinzufügen (falls vorhanden).

Fremdschlüssel

Ein **Fremdschlüssel** ist ein Attribut (oder eine Kombination von Attributen) einer Relation, das auf den **Primärschlüssel** (oder Schlüsselkandidaten) einer anderen Relation verweist.

Notation

- course: {{ courseID, title, ects, taughtBy -> professor }}
- professor: {{ empID, name, rank, office }}

Alternative Notation

- course: {{ courseID, title, ects, taughtBy }}
- professor: {{ empID, name, rank, office }}

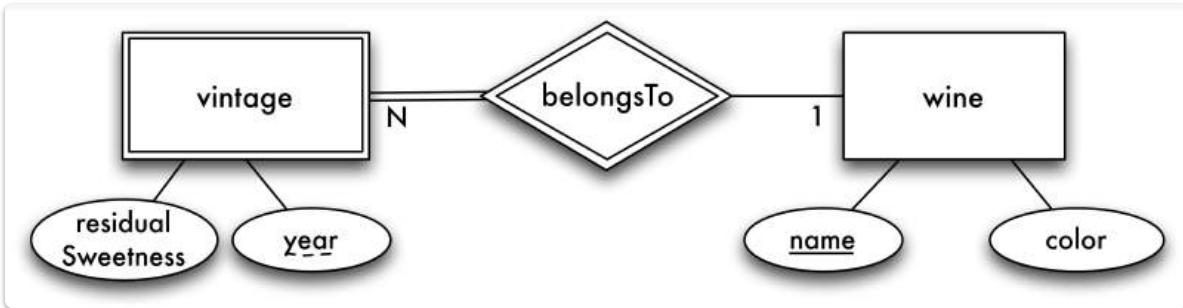
Foreign key: course.taughtBy → professor.empID (Dies bedeutet, dass das Attribut `taughtBy` in der Relation `course` ein Fremdschlüssel ist, der auf das Attribut `empID` in der Relation `professor` verweist.)

Notation für zusammengesetzte Schlüssel

$\{R.A_1, R.A_2\} \rightarrow \{S.B_1, S.B_2\}$ (Hierbei verweisen die Attribute A_1, A_2 der Relation R auf die Attribute B_1, B_2 der Relation S . Dies wird verwendet, wenn ein Fremdschlüssel aus mehreren Attributen besteht, die zusammen einen Primärschlüssel in einer anderen Relation bilden.)

Relation aus zusätzlichen Konzepten ableiten

Schwache bzw. (existenz-)abhängige Entitätstypen



Entität eines schwachen Entitätstyps sind:

- In ihrer Existenz von einem anderen, *übergeordneten (starken) Entitätstypen* abhängig. (Ein schwacher Entitätstyp kann nicht ohne die Existenz des starken Entitätstyps existieren.)
- Normalerweise nur in Kombination mit dem Schlüssel des übergeordneten Entitätstyps eindeutig identifizierbar. (Der eigene "Teilschlüssel" des schwachen Entitätstyps ist nur in Verbindung mit dem Primärschlüssel des starken Entitätstyps eindeutig.)

Wie kann man schwache Entitätstypen als Relationen abbilden?

Schwache Entitätstypen und deren identifizierende Beziehungstypen können **immer zusammengeführt** werden.

- `wine: {{ color, name }}`
- `vintage: {{ [[name]] -> wine, year, residualSweetness }}` (Der doppelt umklammerte Name `[[name]]` in `vintage` deutet auf den **partiellen Schlüssel** des schwachen Entitätstyps hin, der zusammen mit dem Primärschlüssel der starken Entität (`wine`) den vollständigen Primärschlüssel bildet. Der Pfeil `->` `wine` zeigt an, dass `name` aus `wine` als Fremdschlüssel integriert wird.)

Rekursive Beziehungstypen

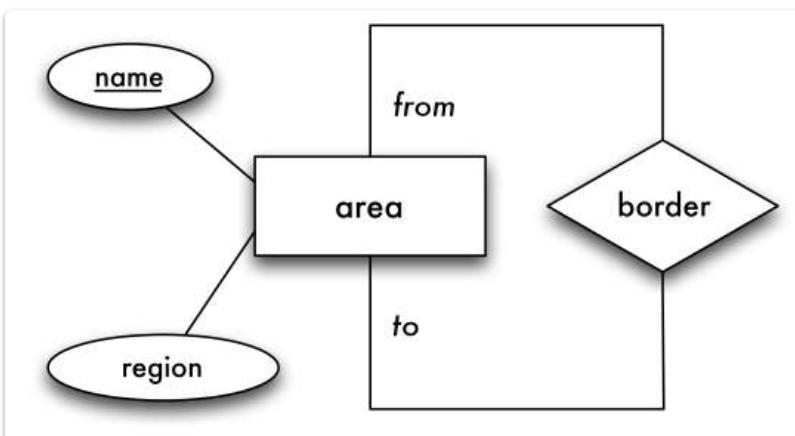


Abbildung wie normale M:N-Beziehungstypen mit Umbenennung der Fremdschlüsselelemente.

- area: {{ name, region }} (Hier wird angenommen, dass `region` ein weiteres Attribut ist, das `area` eindeutig identifiziert oder beschreibt.)
- border: {{ [[from]] -> area, [[to]] -> area }} (Der Primärschlüssel der Relation `border` besteht aus zwei Fremdschlüsseln, die jeweils auf den Primärschlüssel von `area` verweisen. `from` und `to` sind dabei die Rollen, die die Beziehung innerhalb der `area`-Entität beschreiben.)

Rekursive funktionale Beziehungstypen

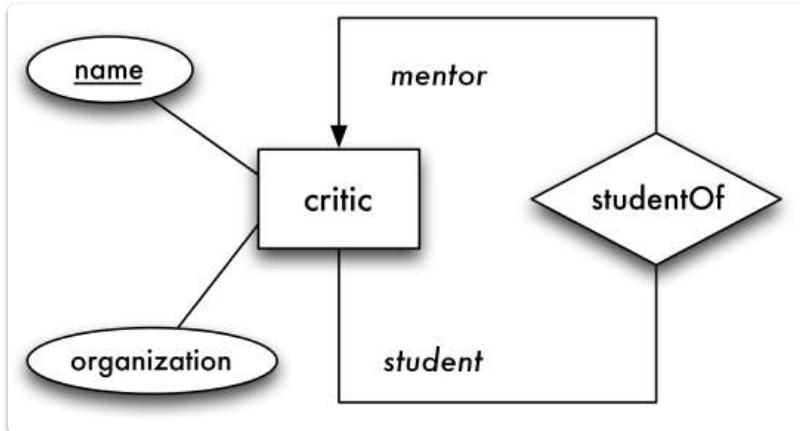
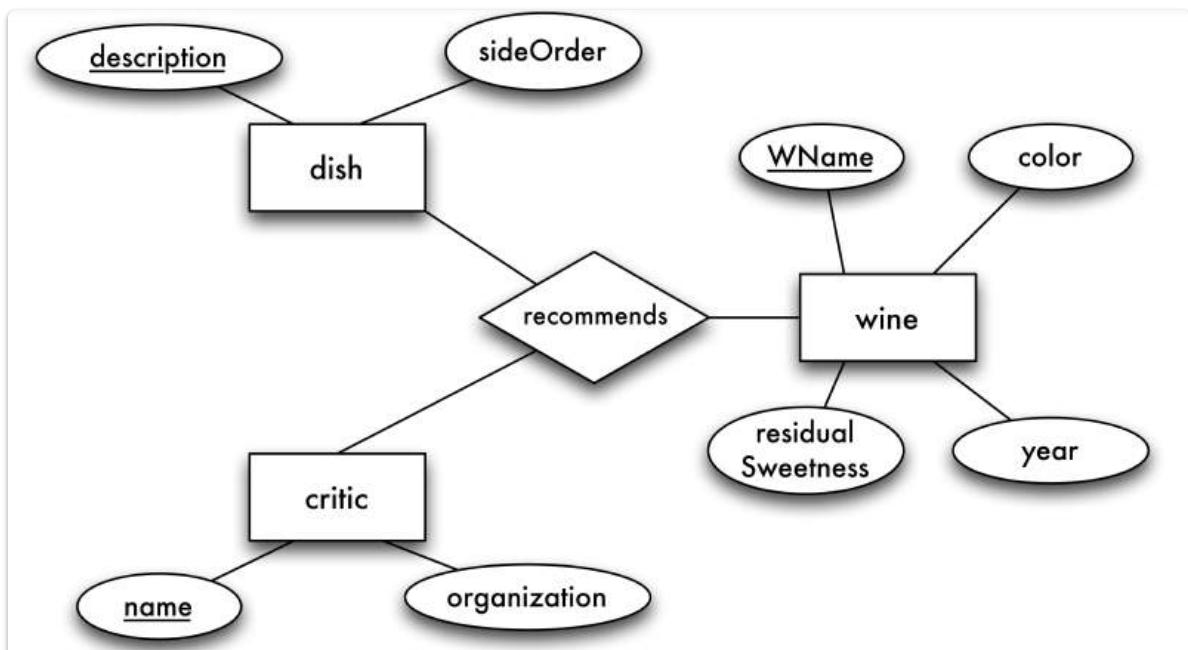


Abbildung wie normale 1:N-Beziehungstypen inklusive Zusammenführung.

- `critic: {{ [[name]], organization, mentor -> critic }}` (Hier ist `mentor` ein Fremdschlüssel, der auf den Primärschlüssel von `critic` selbst verweist, um die Mentor-Beziehung abzubilden. Da die Beziehung funktional ist (1:N), wird der Fremdschlüssel (`mentor`) direkt in die Relation des Entitätstyps auf der "N"-Seite (`critic`) integriert.)

N-äre Beziehungstypen



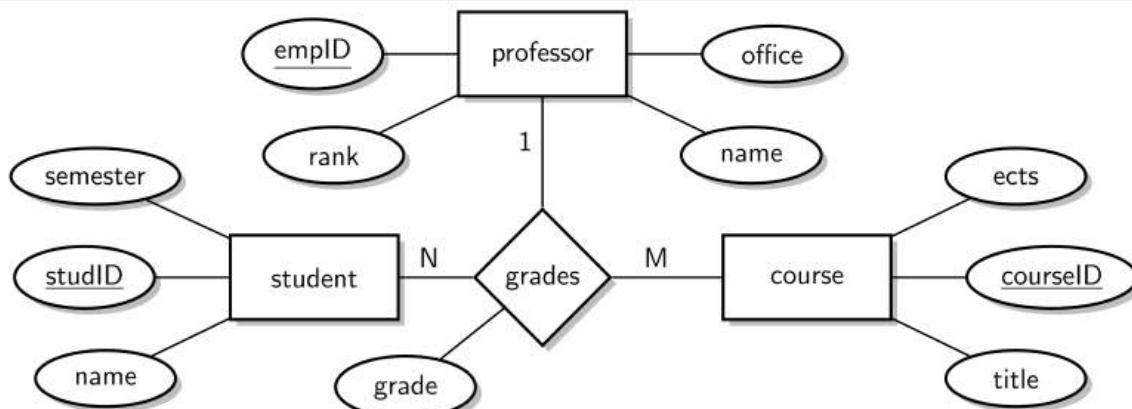
Entitätstypen wurden zunächst nach den Standardregeln abgebildet.

- critic: {{ [[name]], organization }}
- dish: {{ [[description]], sideOrder }}
- wine: {{ [[WName]], year, residualSweetness }}

N-äre Beziehungstypen (N:M:P)

- recommends: {{ WName -> wine, description -> dish, name -> critic }} (Für einen n-ären Beziehungstyp wird eine eigene Relation erstellt. Der Primärschlüssel dieser Relation besteht aus den Fremdschlüsseln der beteiligten Entitätstypen. Hier sind `WName`, `description` und `name` Fremdschlüssel, die jeweils auf die Primärschlüssel von `wine`, `dish` und `critic` verweisen.)

N:M:1-Beziehungstyp

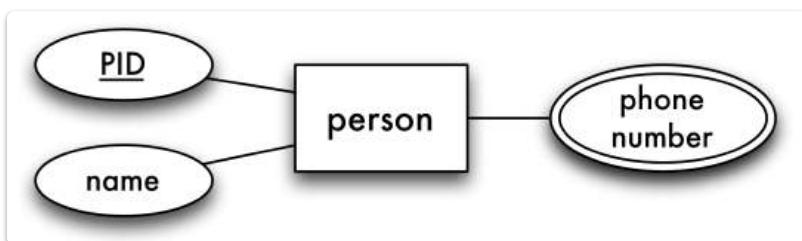


Relationen

- student: {[studID, name, semester]}
- course: {[courselD, title, ects]}
- professor: {[emplID, name, rank, office]}
- grades: {[studID → student, courselD → course, emplID → professor, grade]}

Spezielle Attribute

Mehrwertige Attribute

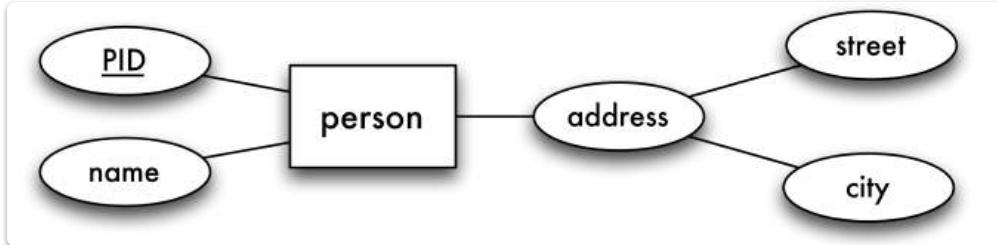


Es wird eine separate Relation für jedes mehrwertige Attribut erstellt.

Relationen:

- person: {{ PID, name }} (Der Entitätstyp person wird als normale Relation abgebildet.)
- phoneNumber: {{ [[PID]] -> person, number }} (Eine neue Relation phoneNumber wird für das mehrwertige Attribut erstellt. Ihr Primärschlüssel besteht aus dem Primärschlüssel des ursprünglichen Entitätstyps (PID , hier als Fremdschlüssel auf person verweisend) und dem Attribut des mehrwertigen Attributs (number). Dies ermöglicht mehrere Telefonnummern pro Person.)

Zusammengesetzte Attribute

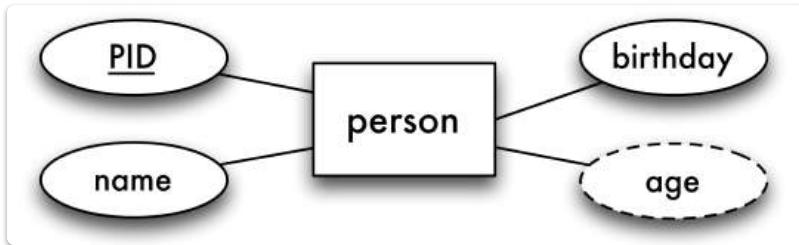


Zusammengesetzte Attribute werden der Relation des zugehörigen Entitätstyps hinzugefügt.

Relation:

- person: {{ PID, name, street, city }} (Die Komponenten des zusammengesetzten Attributs (street , city) werden direkt als einzelne Attribute in die Relation des Entitätstyps (person) aufgenommen. Das zusammengesetzte Attribut address selbst wird nicht als separate Relation abgebildet.)

Abgeleitete Attribute



Diese werden bei der Abbildung in Relationen **ignoriert** und später mittels **Views** hinzugefügt. (Abgeleitete Attribute wie age können zur Laufzeit aus anderen Attributen (birthday) berechnet werden und müssen daher nicht physisch in der Datenbank gespeichert werden. Views sind virtuelle Tabellen, die das Ergebnis einer Abfrage sind und zur Darstellung abgeleiteter Daten verwendet werden können.)

Überblick der Schritte

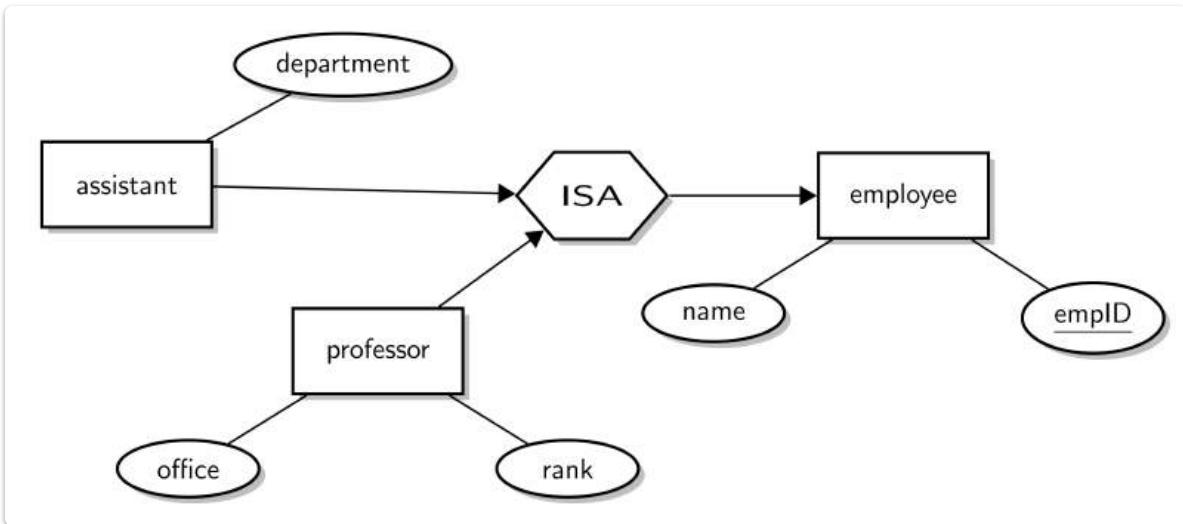
- **Regulärer Entitätstyp:** Relation erstellen, spezielle Attributtypen beachten.
- **Schwacher Entitätstyp:** Relation erstellen, Primärschlüssel aus partiellem Schlüssel und Fremdschlüssel des starken Entitätstyps bilden.

- **1:1 binärer Beziehungstyp:** Erweitern einer Relation mit Fremdschlüssel.
- **1:N binärer Beziehungstyp:** Erweitern einer Relation mit Fremdschlüssel.
- **M:N Beziehungstyp:** Erstellen einer neuen Relation.
- **N-ärer Beziehungstyp:** Erstellen einer neuen Relation.

Relationale Modellierung der Generalisierung

Das relationale Modell unterstützt keine Generalisierung und kann daher keine Vererbung ausdrücken.

→ Generalisierung wird **simuliert**.



Arten um dieses ER-Diagramm auf Relationen abzubilden

Alternative 1: Hauptklassen

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

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

employee	
empID	name
2123	P. Müller
2124	A. Schmidt

employee:
{{ empID, name }}

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

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

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

assistant:
{{ empID, name, department }}

Alternative 2: Partitionierung

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

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

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

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

professor		
empID	rank	office
2125	C4	226
...

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

assistant	
empID	department
2150	DBS
2151	Physics

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

Alternative 3: Vollständige Redundanz

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

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

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

employee:
{{ empID, name }}

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

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

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

assistant:
{{ empID, name, department }}

Alternative 4: Eine einzige Relation

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

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

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

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

Allgemeine Lösung

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

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

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

Zusammenfassung

ER-Diagramme auf Relationen abbilden:

- **Entitätstypen:** Abbildung als Relation.
- **Binäre Beziehungstypen:**
 - 1:1-Beziehungstypen: Integration des Fremdschlüssels in eine der beteiligten Relationen.
 - 1:N-Beziehungstypen: Integration des Fremdschlüssels in die Relation auf der "N"-Seite.
 - M:N-Beziehungstypen: Erstellung einer *neuen Relation* für den Beziehungstyp, deren Primärschlüssel aus den Fremdschlüsseln der beteiligten Entitätstypen besteht.
- **N-stellige Beziehungstypen:** Erstellung einer *neuen Relation*, deren Primärschlüssel aus den Fremdschlüsseln der beteiligten Entitätstypen besteht.
- **Schwache Entitätstypen:** Integration in die Relation des starken Entitätstyps oder Erstellung einer eigenen Relation mit einem Primärschlüssel, der den partiellen Schlüssel und den Fremdschlüssel zum starken Entitätstyp kombiniert.
- **Rekursive Beziehungstypen:** Abbildung analog zu den entsprechenden binären oder n-stelligen Beziehungstypen, wobei die Fremdschlüssel auf dieselbe Relation verweisen.
- **Generalisation:**
 - Die „**Partitionierungsvariante**“ wird in den meisten Anwendungen bevorzugt. (Dies entspricht Alternative 2, bei der Super- und Subklassen eigene Relationen erhalten und über Fremdschlüssel verknüpft sind.)
 - Die behandelten „Abbildungsregeln“ haben das Ziel, die **Anzahl von Relationen zu minimieren, nicht notwendigerweise die NULL-Werte**. (Dies ist ein wichtiger Design-Grundsatz, der die Effizienz der Datenbank beeinflusst. Die Minimierung von Relationen kann die Komplexität von Abfragen reduzieren, auch wenn dies in manchen Fällen zu NULL-Werten führen kann.)

3. Normalization

Einführung

Anomalien in Datenbanksystemen

- Anomalien sind unerwünschte Nebeneffekte, die bei der Datenmanipulation (Einfügen, Ändern, Löschen) in schlecht entworfenen Datenbanken auftreten können. Sie führen zu Dateninkonsistenzen und Datenverlust.

Hauptquellen für Anomalien

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

courseOffers						
emplID	teacher	rank	office	courseID	title	ects
2125	Socrates	C4	226	5041	Robotics	4
2125	Socrates	C4	226	5049	Ethics	2
2125	Socrates	C4	226	4052	Logic	4
2133	Curie	C3	52	5259	Chemistry	2
2137	Curie	C4	7	4630	Physics	4
⊥	⊥	⊥	⊥	5432	Economy	4
...

Arten von Anomalien

1. Änderungsanomalien (*Update Anomaly*)

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

2. Einfügeanomalien (*Insertion Anomaly*)

- Problem:** Eine neue LVA (Lehrveranstaltung) kann nicht eingefügt werden, ohne gleichzeitig Null-Werte (*NULL values*) für Attribute zu setzen, die eigentlich nicht unbekannt sein sollten (z.B. ein zugehöriger Professor:in).

- **Beispiel:** Ein neuer Kurs kann nicht erfasst werden, wenn der zugehörige Professor noch nicht existiert oder zugewiesen ist, obwohl der Kurs selbst schon bekannt ist.

3. Löschanomalien (*Deletion Anomaly*)

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

Ziel des Datenbankentwurfs

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

Strategie zur Problemlösung

- **Transformation funktionaler Abhängigkeiten:** Alle gegebenen funktionalen Abhängigkeiten sollen in Schlüsselabhängigkeiten transformiert werden.
 - **Ziel:** Dabei dürfen keine semantischen Informationen (Bedeutung der Daten und ihrer Beziehungen) verloren gehen.
 - (Funktionale Abhängigkeiten beschreiben, wie Werte eines Attributs die Werte anderer Attribute bestimmen. Schlüsselabhängigkeiten bedeuten, dass die Abhängigkeit über einen Schlüssel läuft, was Redundanz reduziert)

Gute und schlechte Zerlegungen

Gute Zerlegung

Grundidee zur Vermeidung von Redundanz, Anomalien etc.

- Zerlege *courseOffers* in:
 - *course* (*courseID, title, ects, empID*)
 - *instructor* (*empID, teacher, rank, office*)

course			
courseID	title	ects	empID
5041	Robotics	4	2125
5049	Ethics	2	2125
4052	Logic	4	2125
5259	Chemistry	2	2133
4630	Physics	4	2137

instructor			
empID	teacher	rank	office
2125	Socrates	C4	226
2133	Curie	C4	36
2137	Curie	C4	7

$\pi_{empID, teacher, rank, office}(courseOffers)$

$\pi_{courseID, title, ects, empID}(courseOffers)$

courseOffers						
empID	teacher	rank	office	courseID	title	ects
2125	Socrates	C4	226	5041	Robotics	4
2125	Socrates	C4	226	5049	Ethics	2
2125	Socrates	C4	226	4052	Logic	4
2133	Curie	C3	52	5259	Chemistry	2
2137	Curie	C4	7	4630	Physics	4

$courseOffers =$
 $instructor \bowtie course$

Diese Zerlegung ist im Sinne der Normalisierung sinnvoll, da so Redundanzen vermieden werden.

Anforderungen an eine gute Zerlegung

- Alle Attribute des originalen Schemas müssen in der Zerlegung vorkommen:

$$R = R_1 \cup R_2 \cup \dots \cup R_n$$

- Das bedeutet, dass die Vereinigung aller Attribute der zerlegten Relationen R_i wieder alle Attribute der ursprünglichen Relation R ergeben muss. Keine Information darf verloren gehen.

- Zerlegung mit verlustfreiem Join:

Für Relationen R mit Schema R gilt:

$$R = \pi_{R_1}(R) \bowtie \pi_{R_2}(R)$$

- Wenn die ursprüngliche Relation R durch den *natürlichen Join* (\bowtie) ihrer Projektionen (π) auf die zerlegten Schemata R_1 und R_2 wiederhergestellt werden kann, spricht man von einem verlustfreien Join. Dies ist entscheidend, um die ursprünglichen Daten exakt wiederherstellen zu können und somit keine Informationen durch die Zerlegung zu verlieren.

Schlechte Zerlegung

course			
courseID	title	ects	teacher
5041	Robotics	4	Socrates
5049	Ethics	2	Socrates
4052	Logic	4	Socrates
5259	Chemistry	2	Curie
4630	Physics	4	Curie

instructor			
empID	teacher	rank	office
2125	Socrates	C4	226
2133	Curie	C4	36
2137	Curie	C4	7

$\pi_{courseID, title, ects, teacher}(courseOffers) \ \pi_{empID, teacher, rank, office}(courseOffers)$

courseOffers						
empID	teacher	rank	office	courseID	title	ects
2125	Socrates	C4	226	5041	Robotics	4
2125	Socrates	C4	226	5049	Ethics	2
2125	Socrates	C4	226	4052	Logic	4
2133	Curie	C3	36	5259	Chemistry	2
2133	Curie	C3	36	4630	Physics	4
2137	Curie	C4	7	5259	Chemistry	2
2137	Curie	C4	7	4630	Physics	4
...

$courseOffers =$
 $instructor \bowtie course$

Funktionale Abhängigkeiten

Definition

Symbole

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

Definition einer funktionalen Abhängigkeit

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

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

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

Triviale funktionale Abhängigkeit

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

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

Beispiele

Postleitzahl und Stadt

Wenn ich die Postleitzahl kenne, dann kenne ich die Stadt.

- Die Postleitzahl 1040 gibt mir genau eine Stadt: Wien
- Die Postleitzahl 3400 gibt mir nicht {Wien, Klosterneuburg} (Dies würde bedeuten, dass 3400 mal Wien und mal Klosterneuburg sein könnte, was der Definition der funktionalen Abhängigkeit widerspricht.)
- **Notation:** {Postleitzahl} \rightarrow {Stadt} (Die Postleitzahl bestimmt die Stadt eindeutig.)

Sozialversicherungsnummer und Name

Die Sozialversicherungsnummer (VSNR) gibt mir einen Namen.

- VSNR 8476120478 gibt mir Sean Connery
- **Notation:** $VSNR \rightarrow Vorname, Nachname$ (Die Sozialversicherungsnummer bestimmt Vorname und Nachname eindeutig.)

Welche funktionalen Abhängigkeiten (FDs, Functional Dependencies) sind erfüllt?

R			
A	B	C	D
a4	b2	c4	d3
a1	b1	c1	d1
a1	b1	c1	d2
a2	b2	c3	d2
a3	b2	c4	d3

Analyse der funktionalen Abhängigkeiten basierend auf der gegebenen Tabelle (Instanz) R:

- $\{A\} \rightarrow \{A\}$: **ja** (Trivial, da A in A enthalten ist.)
- $\{A\} \rightarrow \{B\}$: **ja** (Wenn A gleich, dann B gleich. Prüfen: (a1,b1) und (a1,b1) sind identisch in A und B.)
- $\{A\} \rightarrow \{C\}$: **ja** (Wenn A gleich, dann C gleich. Prüfen: (a1,c1) und (a1,c1) sind identisch in A und C.)
- $\{A\} \rightarrow \{D\}$: **nein** (Prüfen: Für $A = a1$ gibt es $D = d1$ (Zeile 2) und $D = d2$ (Zeile 3). Der A-Wert $a1$ bestimmt den D-Wert nicht eindeutig.)
- $\{A\} \rightarrow \{C, D\}$: **nein** (Wenn A gleich, dann C und D gleich. Da $A \rightarrow D$ nicht gilt, kann auch $A \rightarrow \{C, D\}$ nicht gelten.)
- $\{B\} \rightarrow \{D\}$: **nein** (Prüfen: Für $B = b2$ gibt es $D = d3$ (Zeile 1) und $D = d2$ (Zeile 4). Der B-Wert $b2$ bestimmt den D-Wert nicht eindeutig.)
- $\{B\} \rightarrow \{A\}$: **nein** (Prüfen: Für $B = b2$ gibt es $A = a4$ (Zeile 1) und $A = a2$ (Zeile 4). Der B-Wert $b2$ bestimmt den A-Wert nicht eindeutig.)
- $\{C, D\} \rightarrow \{A\}$: **nein** (Prüfen: Für $\{C, D\} = \{c4, d3\}$ gibt es $A = a4$ (Zeile 1) und $A = a3$ (Zeile 5). Die Kombination der C- und D-Werte $\{c4, d3\}$ bestimmt den A-Wert nicht eindeutig.)
- $\{C, D\} \rightarrow \{B\}$: **ja** (Prüfen:
 - $(c4, d3) \rightarrow b2$ (Zeile 1)
 - $(c1, d1) \rightarrow b1$ (Zeile 2)
 - $(c1, d2) \rightarrow b1$ (Zeile 3)
 - $(c3, d2) \rightarrow b2$ (Zeile 4)

- $(c4, d3) \rightarrow b2$ (Zeile 5)
In allen Fällen, wenn $\{C, D\}$ gleich ist, ist auch B gleich. *Beispiel: Für $\{c4, d3\}$ kommt nur $b2$ als B-Wert vor.)*
- **Vereinfachte Notation:** $CD \rightarrow B$

Semantische Konsistenzbedingungen

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

- **Wichtig:** Eine funktionale Abhängigkeit wird *nicht* dadurch definiert, dass sie in einer *einzelnen* momentanen Dateninstanz gilt. Sie muss vielmehr eine **grundlegende Regel** sein, die für alle *denkbaren und gültigen* Datenzustände der Datenbank zutrifft, basierend auf der Bedeutung der Daten in der realen Welt. Die Überprüfung an einer Instanz dient lediglich zur Illustration oder zum Finden von Verletzungen.

Schlüssel

Superschlüssel

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

Volle funktionale Abhängigkeit

β ist **voll funktional abhängig** von α wenn:

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

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

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

Schlüsselkandidat

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

- Ein Schlüsselkandidat ist also ein *minimaler* Superschlüssel. Das bedeutet, er ist ein Superschlüssel, und wenn man irgendein Attribut aus ihm entfernt, ist er kein Superschlüssel mehr.
- Ein Schlüsselkandidat wird als **Primärschlüssel** ausgewählt!
 - Aus der Menge aller Schlüsselkandidaten wird in der Regel einer als Primärschlüssel für eine Relation bestimmt. Dieser Primärschlüssel dient dann zur eindeutigen Identifikation der Tupel in der Relation.

Ableitung funktionaler Abhängigkeiten

Bestimmung funktionaler Abhängigkeiten

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

- $\{empID\} \rightarrow \{empID, name, rank, office, city, street, zip, dialCode, region, inhabitants, government\}$
 - Dies ist ein **Schlüsselkandidat**. (Da die `empID` alle anderen Attribute eindeutig bestimmt, ist es ein Superschlüssel. Wenn es sich um einen minimalen Superschlüssel handelt, ist es ein Schlüsselkandidat.)
 - Dies impliziert, dass eine Mitarbeiter-ID ausreicht, um alle Informationen über einen Mitarbeiter und eventuell zugehörige Standortdaten (Stadt, Region etc.) eindeutig zu identifizieren.
- $\{city, region\} \rightarrow \{inhabitants, dialCode\}$
 - Die Kombination aus Stadt und Region bestimmt eindeutig die Einwohnerzahl und die Vorwahl.
- $\{zip\} \rightarrow \{region, city, inhabitants\}$
 - Eine Postleitzahl bestimmt eindeutig die Region, die Stadt und die Einwohnerzahl.
- $\{region, city, street\} \rightarrow \{zip\}$
 - Die Kombination aus Region, Stadt und Straße bestimmt eindeutig die Postleitzahl. (Dies ist die Umkehrung der obigen Abhängigkeit und zeigt, dass die PLZ durch detailliertere Adressinformationen bestimmt werden kann.)
- $\{region\} \rightarrow \{government\}$
 - Eine Region bestimmt eindeutig die jeweilige Regierung (z.B. Bundeslandregierung).
- $\{office\} \rightarrow \{empID\}$
 - Ein Büro (als Entität oder spezifischer Standort) bestimmt eindeutig eine Mitarbeiter-ID. (Dies könnte bedeuten, dass ein Büro nur von einem spezifischen Mitarbeiter geleitet wird, oder dass die Büro-ID eine eindeutige Beziehung zu einer Mitarbeiter-ID hat.)

Davon können weitere abgeleitet werden

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

- $\{office\} \rightarrow \{empID, name, rank, office, city, street, zip, dialCode, inhabitants, government\}$
 - **Ableitung:** Wir wissen, dass $\{office\} \rightarrow \{empID\}$ gilt.
 - Wir wissen auch, dass $\{empID\} \rightarrow \{empID, name, rank, office, city, street, zip, dialCode, region, inhabitants, g\}$ gilt.
 - Durch die Transitivitätsregel ($A \rightarrow B$ und $B \rightarrow C \implies A \rightarrow C$) können wir ableiten, dass $\{office\}$ alle Attribute bestimmt, die von $\{empID\}$ bestimmt werden.
 - (Anmerkung: Das Attribut 'city' ist hier zweimal aufgeführt, dies ist wahrscheinlich ein Tippfehler in den Originalfolien.)
- $\{zip\} \rightarrow \{government\}$
 - **Ableitung:** Wir wissen, dass $\{zip\} \rightarrow \{region, city, inhabitants\}$ gilt.
 - Wir wissen auch, dass $\{region\} \rightarrow \{government\}$ gilt.
 - Da $\{zip\}$ die `region` bestimmt, und die `region` die `government` bestimmt, folgt durch Transitivität, dass $\{zip\} \rightarrow \{government\}$.

Herleitung weiterer FDs

- Aus einer Menge von FDs F sind **weitere FDs herleitbar**.
 - F^+ beinhaltet **alle FDs**, die aus F abgeleitet werden können, d.h., alle FDs, die von FDs in F **logisch impliziert** werden.
 - F^+ wird **Hülle (closure)** von F genannt.
 - **Inferenzregeln** (Armstrong-Axiome) beschreiben die Herleitung.
-

Die Armstrong-Axiome

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

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

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

Weitere Ableitungsregeln

Nicht zwingend notwendig, aber oftmals komfortabel für Herleitungen.

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

Beispiel

Gegeben seien die folgenden FDs F , leite mithilfe der Armstrong-Axiome weitere her.

- $A \rightarrow BC$
- $CD \rightarrow E$
- $B \rightarrow D$
- $E \rightarrow A$

Hergeleitete FDs

- $E \rightarrow A$ und $A \rightarrow BC$, dann $E \rightarrow BC$ (Transitivität)
- $B \rightarrow D$, dann $CB \rightarrow CD$ (Erweiterung/Verstärkung)
- $CB \rightarrow CD$ und $CD \rightarrow E$, dann $CB \rightarrow E$ (Transitivität)

Algorithmus zur Bestimmung der Attributhülle

Algorithmus `attrClosure(F , α)`

Input:

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

Algorithmus-Schritte:

Algorithmus attrClosure(F , α):

```

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

```

Korrektheit und Terminierung des Algorithmus

- Der Algorithmus berechnet die Attributhülle von α korrekt.
 - Bei Terminierung: Alle Teile der Attributhülle von α sind in `result`.
- Der Algorithmus terminiert (da die Menge der Attribute endlich ist und `result` bei jeder Änderung wächst).

Anwendungen des Algorithmus zur Attributhülle

- Testen, ob eine funktionale Abhängigkeit $\alpha \rightarrow \beta$ gilt.
 - Berechne α^+ (die Attributhülle von α) mit `attrClosure(F, α)`.
 - Wenn $\beta \subseteq \alpha^+$ (d.h. alle Attribute von β in der Attributhülle von α enthalten sind), dann gilt die funktionale Abhängigkeit $\alpha \rightarrow \beta$.
- Testen, ob eine Menge von Attributen $\kappa \subseteq R$ ein Superschlüssel ist.
 - Ein *Superschlüssel* ist eine Menge von Attributen, die eindeutig alle anderen Attribute einer Relation bestimmt.
 - Rufe `attrClosure(F, κ)` auf, um κ^+ (die Attributhülle von κ) zu erhalten.
 - Falls $\kappa^+ = R$ (d.h. die Attributhülle von κ alle Attribute des Schemas R enthält), dann ist κ ein Superschlüssel von R .

Beispiel

Relation R : $\{[A, B, C, D]\}$ $\alpha = \{A, D\}$

Menge an FDs: $F = \{A \rightarrow B, A \rightarrow C, CD \rightarrow A\}$

$\alpha^+ = result = \{$

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

}

Is $\{ A, D \}$ a super key? ja

Is $\{ A \}$ a super key? nein

Is $\{ D \}$ a super key? nein

Kanonische Überdeckung

Äquivalente FD-Mengen

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

Beobachtung:

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

Ziel:

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

Kanonische Überdeckung F_c (minimale Überdeckung)

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

Eigenschaften:

1. $F_c \equiv F$, also $F_c^+ = F^+$ (Äquivalenz, falls die Hüllen gleich sind).
 2. In F_c existieren **keine FDs** $\alpha \rightarrow \beta$, bei denen α oder β überflüssige Attribute enthalten.
- D.h. es muss gelten:

- (a) $\forall A \in \alpha: (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\} \not\equiv F_c$ (Das Entfernen eines Attributs aus der linken Seite oder das Ersetzen von α durch $\alpha - A$ verändert die Hülle von F_c)
- (b) $\forall B \in \beta: (F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\} \not\equiv F_c$ (Das Entfernen eines Attributs aus der rechten Seite oder das Ersetzen von β durch $\beta - B$ verändert die Hülle von F_c)
- **Überprüfung mit Hilfe der Attributhülle, ob $A \in \alpha$ in $\alpha \rightarrow \beta$ überflüssig ist:**
 - $A \in \alpha$ ist überflüssig, wenn $\beta \subseteq \text{attrClosure}((F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}, \alpha)$ (d.h., wenn β auch ohne A auf der linken Seite in Kombination mit den restlichen FDs abgeleitet werden kann).
- **Überprüfung mithilfe der Attributhülle, ob $B \in \beta$ in $\alpha \rightarrow \beta$ überflüssig ist:**
 - $B \in \beta$ ist überflüssig, wenn $B \in \text{attrClosure}((F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\}, \alpha)$ (d.h., wenn B auch dann abgeleitet werden kann, wenn es nicht auf der rechten Seite von $\alpha \rightarrow \beta$ steht, sondern aus den verbleibenden FDs in F und der modifizierten FD $\alpha \rightarrow (\beta - B)$).

3. Jede linke Seite einer funktionalen Abhängigkeit in F_c ist **einzigartig**.

- Durch Anwendung der **Vereinigungsregel** wird $(\alpha \rightarrow \beta)$ und $(\alpha \rightarrow \gamma)$ durch $\alpha \rightarrow \beta\gamma$ ersetzt. (Dies konsolidiert alle FDs mit gleicher linker Seite).

Algorithmus minimale Überdeckung

Hauptschritte:

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

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

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

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

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

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

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

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

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

Beispiel

Initiale Menge von Abhängigkeiten:

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

Momentane Menge von Abhängigkeiten:

$$F_c = \{A \rightarrow B, B \rightarrow C\}$$

- Schritt 1: Einzigartige linke Seiten
Keine Zusammenführung in diesem Beispiel notwendig
- Schritt 2: Linksreduktion
Ersetzte $AB \rightarrow C$ durch $A \rightarrow C$
- Schritt 3: Rechtsreduktion
Ersetzte $A \rightarrow C$ durch $A \rightarrow \emptyset$
- Schritt 4: Entferne Regeln der Form $\alpha \rightarrow \emptyset$
Entferne $A \rightarrow \emptyset$

Normalisierung durch Zerlegung der Relation

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

Normalformen

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

Gültige und verlustlose Zerlegungen

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

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

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

$$R = R_1 \bowtie R_2$$

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

Verlustlose Zerlegung

Formale Charakterisierung verlustloser Zerlegungen

Gegeben:

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

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

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

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

Beispielzerlegung

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

zerlegt in

customer		drinks	
pub	guest	guest	beer
Kowalski	Kemper	Kemper	pils
Kowalski	Eickler	Eickler	wheat beer
Innsteg	Kemper	Kemper	wheat beer

Rekonstruktion: $R_1 \bowtie R_2$

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

- Die Beziehung zwischen guest, pub und beer ging verloren.
- Die Verletzung der Verlustlosigkeit kann manchmal auch bedeuten, dass bei der Wiederherstellung zusätzliche Tupel entstehen.
- Also keine verlustlose Zerlegung.

Abhängigkeitsbewahrung

Zweites Kriterium für eine gute Zerlegung

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

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

Eine Zerlegung bewahrt Abhängigkeiten, wenn

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

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

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

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

$result = \alpha$

repeat

```
for each  $R_i$  in the decomposition do
   $t = (result \cap R_i)^+ \cap R_i$ 
   $result = result \cup t$ 
end for
```

until result changes no more

Beispiele

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

FDs

- {zip} → {city, region}
- {street, city, region} → {zip}

Zerlegung

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

Ist diese Zerlegung verlustlos?
Bewahrt diese Zerlegung Abhängigkeiten?

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

 $\pi_{zip, street}$ $\pi_{city, region, zip}$

street	
zip	street
15234	Goethestraße
60313	Goethestraße
60437	Galgenstraße

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

Die Zerlegung bewahrt die Abhängigkeiten **nicht**:

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

Gegeben

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

Ist das eine verlustlose Zerlegung?

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

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

JA:

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

Bewahrt diese Zerlegung Abhängigkeiten?

Hinweis (wir können alle Abhängigkeiten lokal überprüfen):

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

JA:

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

Gegeben

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

Ist das eine verlustlose Zerlegung?

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

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

JA:

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

Gegeben

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

Bewahrt diese Zerlegung Abhängigkeiten?

Hinweis (wir können alle Abhängigkeiten lokal überprüfen):

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

NEIN:

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

Zusammenfassung funktionaler Abhängigkeiten

Eigenschaften eines guten Datenbankentwurfs

- Keine Redundanz
- Keine Änderungsanomalien
- Alle Informationen können dargestellt werden

Warum Zerlegung?

- Verwandelt einen schlechten Entwurf in einen guten
 - Zerlege große Relationsschemata in mehrere kleinere

Eine gute Zerlegung

- ist verlustlos
- bewahrt Abhängigkeiten

Normalformen

Normalformen:

- legen Eigenschaften von Relationsschemata fest
- verbieten bestimmte Kombinationen von funktionalen Abhängigkeiten in Relationen
- sollen Redundanzen und Anomalien vermeiden
- Richtlinie, um gute Zerlegungen zu erhalten

Beispiele für Normalformen: **1NF**, **2NF**, **3NF**, **BCNF**, **4NF**, ...

Erste Normalform (1NF)

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

1NF:

Nicht 1NF:

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

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

Zweite Normalform (2NF)

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

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

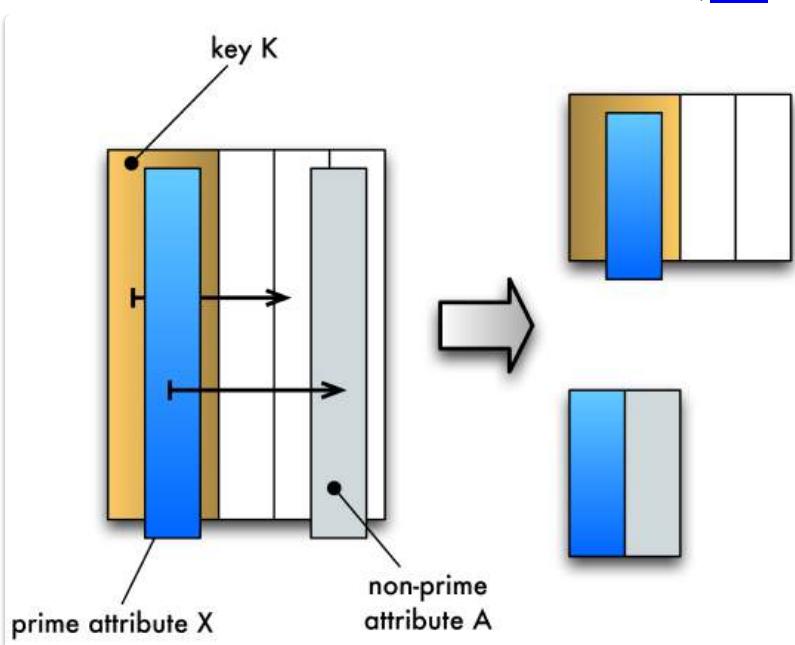
Definitionen:

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

2NF verhindert partielle Abhängigkeiten:

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

Eliminierung partieller Abhängigkeiten



studentCourses			
studID	courselD	name	semester
26120	5001	Fichte	10
27550	5001	Schopenhauer	6
27550	4052	Schopenhauer	6
28106	5041	Carnap	3
28106	5052	Carnap	3
28106	5216	Carnap	3
28106	5259	Carnap	3
...

Zerlegung in zwei Relationen:

- takes: {[studID, courselD]}
- student: {[studID, name, semester]}

Beide Relationen sind in 2NF.

Beispiel

studentCourses			
studID	courselD	name	semester
26120	5001	Fichte	10
27550	5001	Schopenhauer	6
27550	4052	Schopenhauer	6
28106	5041	Carnap	3
28106	5052	Carnap	3
28106	5216	Carnap	3
28106	5259	Carnap	3
...

- Kandidatenschlüssel: $\{ \{ \text{studID}, \text{courselD} \} \}$
- Schlüsselattribute (prim): $\{ \text{studID}, \text{courselD} \}$
- Nicht-Schlüsselattribute (nicht prim): $\{ \text{name}, \text{semester} \}$
- FDs: $\{ \text{studID} \} \rightarrow \{ \text{name} \}$ und $\{ \text{studID} \} \rightarrow \{ \text{semester} \}$

Partielle Abhängigkeit: studentCourses ist nicht in 2NF

Dritte Normalform (3NF)

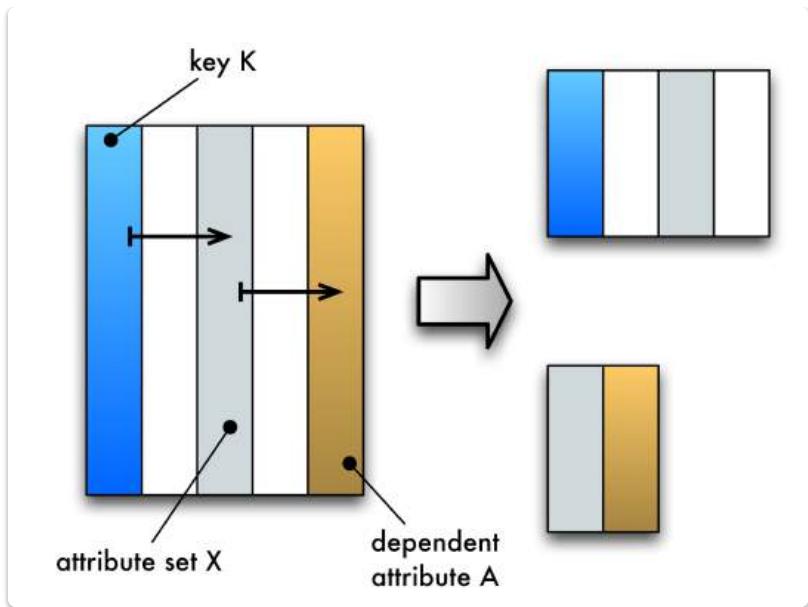
Ein Relationsschema R ist in **Dritter Normalform (3NF)**, wenn für jede für R geltende FD der Form $\alpha \rightarrow B$ mit Attribut $B \in R$ mindestens eine von drei Bedingungen gilt:

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

Eigenschaften der 3NF

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

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



- Nur Prim-Attribute können Endpunkte von transitiven Abhängigkeiten sein!

Beispiel

studentCourses			
studID	courselD	name	semester
26120	5001	Fichte	10
27550	5001	Schopenhauer	6
27550	4052	Schopenhauer	6
28106	5041	Carnap	3
28106	5052	Carnap	3
28106	5216	Carnap	3
28106	5259	Carnap	3
...

- Kandidatenschlüssel: { {studID, courselD} }
 - FDs: {studID} → {name}, {studID} → {semester}
- Hinweise (FD $\alpha \rightarrow B$)
- $B \in \alpha$, i.e., die FD ist trivial – beide FDs sind nicht trivial
 - α ist ein Superschlüssel von R – {studID} ist kein Superschlüssel
 - B ist prim – name und semester sind nicht prim

⇒ Die Relation ist nicht in 3NF

Boyce-Codd-Normalform (BCNF)

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

- $B \in \alpha$, d.h. die FD ist **trivial**.
- α ist **Superschlüssel** von R .

Eigenschaften der BCNF

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

Beispiel 3NF vs. BCNF

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

FDs

- ✓ {city, state} → {inhabitants}
- ✓ {state} → {governor}
- ✓ {governor} → {state}

Kandidatenschlüssel:

{city, state} and {city, governor}

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

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

Ist die Relation in BCNF? **NEIN**

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

BCNF Zerlegung

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

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

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

Zerlegungsalgorithmus für BCNF

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

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

Beispiel

cities: {[city, state, governor, inhabitants]}

Kandidatenschlüssel: {city, state} und {city, governor}

FDs:

- ① {state} → {governor}
- ② {city, state} → {inhabitants}
- ③ {governor} → {state}

\mathcal{R}_{i1} governments: {[state, governor]}

\mathcal{R}_{i2} cities: {[city, state, inhabitants]}

Diese Zerlegung ist verlustfrei und bewahrt Abhängigkeiten.

- FD1 und FD3 können governments zugewiesen werden.
- FD2 kann cities zugewiesen werden.

Zerlegung in BSNF

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

Kandidatenschlüssel: {street, city, region}

FDs

- | | |
|----------------------------------|---------------|
| ① {street, city, region} → {zip} | OK |
| ② {zip} → {city, region} | verletzt BCNF |

Zerlegung anhand von FD2:

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

Diese Zerlegung

- ist verlustfrei
- bewahrt aber nicht Abhängigkeiten! FD1 kann nicht überprüft werden, ohne den Join zu berechnen

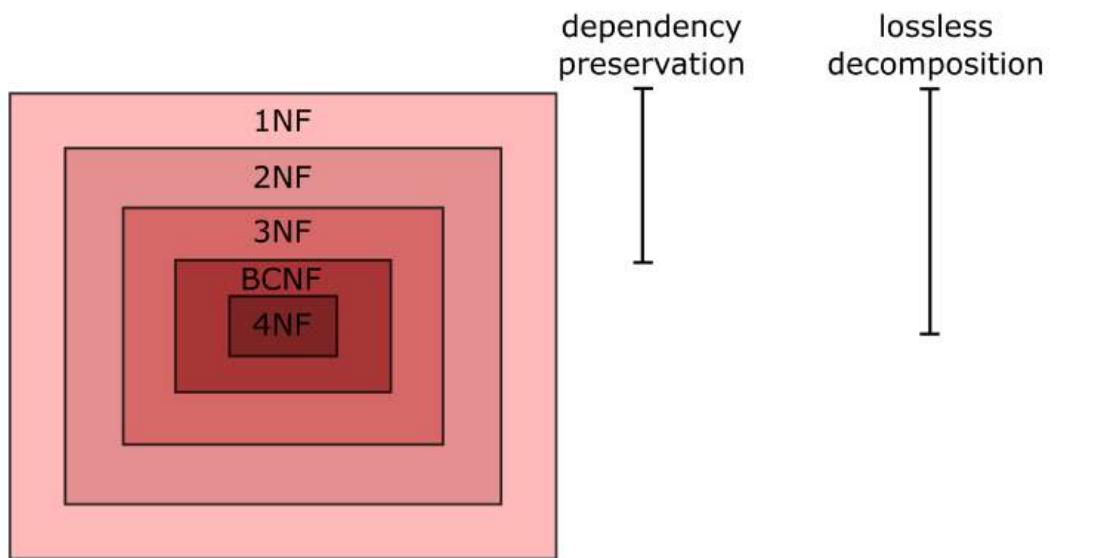
Übersicht über die Normalformen

Normalform	Kurzcharakteristik
1NF	nur atomare Attribute
2NF	keine partielle Abhangigkeit
3NF	keine transitive Abhangigkeit (Ausnahme: Primattribute)
BCNF	keine transitive Abhangigkeit

In der Praxis gibt man sich statt einer BCNF-Zerlegung, die zu einem Verlust der Abhangigkeitsbewahrung fuhren wurde, mit der dritten Normalform zufrieden.

Normalformen

- **Verlustlosigkeit** ist fur alle Zerlegungsalgorithmen in alle Normalformen garantiert.
- Die **Abhangigkeitsbewahrung** kann nur bis zur **dritten Normalform (3NF)** garantiert werden.



- **dependency preservation (Abhangigkeitsbewahrung):** Wenn wir eine Relation in kleinere Relationen zerlegen, bleiben alle ursprnglichen funktionalen Abhangigkeiten erhalten und knnen aus den zerlegten Relationen abgeleitet werden.
- **lossless decomposition (verlustlose Zerlegung):** Wenn wir eine Relation in kleinere Relationen zerlegen und diese spter wieder zusammenfgen, verlieren wir keine Informationen und erhalten die ursprngliche Relation zurck.

Zusammenfassung

Eigenschaften eines „guten“ relationalen Entwurfs

- Vermeidung von Redundanz (mehrfaches Speichern gleicher Informationen).
- Vermeidung von Anomalien (Probleme beim Einfgen, Lschen oder ndern von Daten).

- Vermeidung von Null-Werten (leere Felder, die oft auf unvollständige Informationen hindeuten).
- Informationen können dargestellt werden (alle notwendigen Informationen sind im Schema abbildbar).

Funktionale Abhängigkeiten (FDs)

- Das wichtigste Konzept beim relationalen Datenbankentwurf.
- **Armstrong-Axiome:** Eine Menge von Regeln, um aus einer gegebenen Menge von FDs alle weiteren FDs (die Hülle einer Menge von FDs) zu bestimmen.
- **Kanonische/minimale Überdeckung:** Eine minimale Menge von FDs, die die gleiche Hülle wie die ursprüngliche Menge an FDs hat. Das bedeutet, es gibt keine redundanten FDs in der minimalen Überdeckung.
- FDs sind ein Werkzeug, um einen „guten“ relationalen Entwurf zu garantieren.

Zerlegung

- Hilft, Relationenschemata zu verbessern.
- Versucht, funktionale Abhängigkeiten in Schlüsselabhängigkeiten zu verwandeln.
 - **Intuitivere Erklärung:** Das Ziel ist es, dass jede funktionale Abhängigkeit in einer Relation durch den Primärschlüssel der Relation erklärt wird. Das heißt, wenn wir den Schlüssel kennen, kennen wir auch alle anderen Attribute.
- Ein gutes ER-Modell und die Abbildung auf Relationen führt oft direkt zu Relationen in 3NF (oder höhere NF).

3NF (Dritte Normalform)

- **Verlustfrei** und **abhängigkeitsbewahrend**.
- Kann **immer erreicht werden**.

BCNF (Boyce-Codd-Normalform)

- **Verlustfrei**, aber **nicht immer abhängigkeitsbewahrend**.
 - **Intuitivere Erklärung:** Obwohl BCNF eine stärkere Normalform ist (besser in Bezug auf Redundanzvermeidung), kann es sein, dass bei einer Zerlegung in BCNF einige der ursprünglichen funktionalen Abhängigkeiten nicht mehr direkt aus den zerlegten Relationen abgeleitet werden können, ohne sie wieder zusammenzufügen.

Denormalisierung

- Prozess, bei dem die Normalisierung „zurückgenommen“ wird, um Anfragen schneller bearbeiten zu können.
 - **Intuitivere Erklärung:** Manchmal wird bewusst eine Redundanz in der Datenbank zugelassen, um die Performance bei Lesezugriffen zu verbessern, auch wenn dies zu

den Nachteilen der Normalisierung (Redundanz, Anomalien) führen kann. Dies ist ein Kompromiss zwischen Datenintegrität und Abfragegeschwindigkeit.

4. SQL

Einleitung

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

Bestandteile von SQL

SQL setzt sich aus mehreren Teilen zusammen:

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

Relationen vs. Tabellen

Mengen und Bags

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

Was ist der Unterschied?

Relationen (Mengen):

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

Tabellen (Bags, Multimengen) in SQL:

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

Grundlegende Datentypen in SQL

Datentypen

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

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

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

Tabellen erstellen

Tabelle erstellen

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

Tabelle erstellen (mit UNIQUE Constraint)

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

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

Tabellen kann man auch auf Basis von anderen Tabellen erstellen:

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

Primärschlüssel

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

Primärschlüssel als Spalten-Constraint

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

Primärschlüssel als Tabellen-Constraint

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

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

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

Fremdschlüssel

Gültige Werte des Fremdschlüssels müssen Werten des referenzierten Attributs entsprechen.

Fremdschlüssel als Tabellen-Constraint

```
CREATE TABLE course (
    courseId      integer,
    title         varchar(30) NOT NULL,
    ects          integer,
    taughtby      integer,
    PRIMARY KEY (courseId),
    FOREIGN KEY (taughtby) REFERENCES professor(empid)
);
```

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

Fremdschlüssel, der auf Primärschlüssel und UNIQUE-Constraint verweist

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

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

Defaultwerte

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

Defaultwert für ein Attribut definieren

```
CREATE TABLE wine (
    wineID      integer NOT NULL,
    name        varchar(20) NOT NULL,
```

```

color      varchar(10) DEFAULT 'red',
year       integer,
vineyard   varchar(20)
);

```

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

Zahlengeneratoren

Zahlengeneratoren erzeugen automatisch fortlaufende eindeutige IDs.

Sequenz erstellen und als Defaultwert nutzen

```

CREATE SEQUENCE serial START 101;

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

```

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

Tabellen verändern

Attribute hinzufügen

```
ALTER TABLE professor  
ADD COLUMN office integer;
```

Attribut löschen

```
ALTER TABLE professor  
DROP COLUMN name;
```

Attributtyp ändern

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

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

Tabelle löschen

```
DROP TABLE professor;
```

Tabelle leeren

```
TRUNCATE TABLE professor;
```

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

Daten einfügen

```

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

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

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

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

```

Daten löschen und ändern

Daten löschen

```

DELETE FROM student
WHERE semester > 13;

DELETE FROM student;

```

Daten ändern

```

UPDATE student
SET semester = semester + 1;

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

```

Rechteverwaltung

Rechte gewähren

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

Rechte entziehen

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

Rechte auf Tabellen, Spalten, ...:

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

Transaktionssteuerung

```
BEGIN;
```

kennzeichnet den **Beginn** einer Transaktion.

```
COMMIT;
```

schließt eine Transaktion ab.

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

```
ROLLBACK;
```

setzt eine Transaktion zurück.

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

Anfragesprache

Grundgerüst einer SQL-Anfrage (SFW-Block)

```
SELECT <Liste von Attributen>
FROM <Liste von Tabellen>
WHERE <Bedingung>;
```

- `SELECT <Liste von Attributen>`: Projektionsliste mit arithmetischen Operationen und Aggregatfunktionen
- `FROM <Liste von Tabellen>`: zu verwendende Tabellen, evtl. Umbenennungen
- `WHERE <Bedingung>`: Selektion und Join-Bedingungen, geschachtelte Anfragen

Relationale Algebra → SQL

Mehr dazu → 1. Relationale Algebra

- Projektion $\pi \rightarrow$ `SELECT`
- Kreuzprodukt $\times \rightarrow$ `FROM`
- Selektion $\sigma \rightarrow$ `WHERE`

Auswahl von Tabellen

Man kann Tupelvariablen für Relationen definieren.

```
SELECT *
FROM wine AS W;
```

```
SELECT *
FROM wine W;
```

W	wineID	name	color	year	vineyard
	1042	La Rose Grand Cru	red	1998	Château La Rose
	2168	Creek Shiraz	red	2003	Creek
	3456	Zinfandel	red	2004	Helena
	2171	Pinot Noir	red	2001	Creek
	3478	Pinot Noir	red	1999	Helena
	4711	Riesling Reserve	white	1999	Müller
	4961	Chardonnay	white	2002	Bighorn

Kreuzprodukt (Kartesisches Produkt)

Bei mehr als einer Tabelle in der `FROM`-Klausel wird das Kreuzprodukt gebildet.

```
SELECT *
FROM wine, producer;
```

Als Ergebnis werden **alle** Kombinationen generiert!

Tupelvariablen für mehrfachen Zugriff

Die Definition von Tupelvariablen erlaubt mehrfachen Zugriff auf eine Tabelle (z.B. Self-Join).

```
SELECT *
FROM wine w1, wine w2;
```

Die Spalten des Ergebnisses lauten:

```
w1.wineID, w1.name, w1.color, w1.year, w1.vineyard, w2.wineID, w2.name, w2.color,
w2.year, w2.vineyard
```

Natural Join

- Frühe SQL-Versionen:
 - Kein eigener Join-Operator.
 - Joinbedingung durch Prädikat in der `WHERE`-Klausel definiert.
- Neuere SQL-Versionen:
 - Kennen explizite Join-Operatoren.
 - `NATURAL JOIN` als Abkürzung für die ausführliche Anfrage mit Kreuzprodukt und anschließender Filterung.

Natürlicher Join als expliziter Operator

- Syntax:

```
SELECT *
FROM tabelle1 NATURAL JOIN tabelle2;
```

- Implizite Joinbedingung: Spalten mit gleichem Namen in beiden Tabellen werden für den Join verwendet.
- Ergebnisspalten:
 - Spalten mit eindeutigen Namen aus beiden Tabellen.
 - Joinspalten erscheinen nur einmal im Ergebnis.
- Ergebnistupel: Kombination von Tupeln aus beiden Tabellen, die in den Joinspalten übereinstimmende Werte haben.

Beispiel (aus den Slides)

Angenommen, wir haben die Tabelle `wine` mit den Spalten `wineID`, `name`, `color`, `year`, `vineyard` und die Tabelle `producer` mit den Spalten `producer`, `vineyard`, `region`.

```
SELECT *
FROM wine NATURAL JOIN producer;
```

- Die Joinbedingung basiert auf der Spalte `vineyard`, da sie in beiden Tabellen existiert.
- Die Spalte `vineyard` erscheint nur einmal im Ergebnis.

Wichtig: Ein `NATURAL JOIN` führt implizit einen Gleichheitsvergleich aller Spalten mit identischem Namen durch. Es ist wichtig, sich der Struktur der Tabellen bewusst zu sein, um unerwartete Join-Ergebnisse zu vermeiden, falls ungewollt Spalten mit gleichen Namen existieren.

Weitere Join-Varianten

Es gibt weitere Möglichkeiten, Joins auszudrücken:

USING-Klausel

- Syntax:

```
SELECT *
FROM tabelle1 JOIN tabelle2
USING (spalte1, spalte2, ...);
```

- Nach `USING` steht eine Liste von Spalten, über die der Join berechnet wird (Gleichheit).
- Funktioniert ähnlich wie `NATURAL JOIN`, erlaubt aber die explizite Angabe der Joinspalten, falls mehrere Spalten mit gleichen Namen existieren, aber nicht alle für den Join relevant sind.
- Joinspalten erscheinen nur einmal im Ergebnis.

ON-Klausel

- Syntax:

```
SELECT *
FROM tabelle1 JOIN tabelle2
ON bedingung;
```

- Nach `ON` kann ein beliebiger boolescher Ausdruck stehen (wie in der `WHERE`-Klausel).
- Ermöglicht komplexe Joinbedingungen, die nicht nur auf Gleichheit von Spalten basieren (Theta-Join).

Kreuzprodukt als expliziter Operator

Kreuzprodukt (Kartesisches Produkt)

- Kombiniert jedes Tupel der ersten Tabelle mit jedem Tupel der zweiten Tabelle.
- Syntax:

```
SELECT *
FROM tabelle1, tabelle2;
```

oder explizit:

```
SELECT *
FROM tabelle1 CROSS JOIN tabelle2;
```

- Ergebnisspalten: Alle Spalten aus beiden Tabellen.
 - Anzahl der Ergebnistupel: Produkt der Anzahl der Tupel in den beiden Tabellen ($|Tabelle1| \times |Tabelle2|$).
 - In den meisten Fällen ist ein reines Kreuzprodukt ohne anschließende Filterung (WHERE - Klausel oder Join-Bedingung) wenig sinnvoll, da es zu sehr großen Ergebnismengen führen kann. Es bildet aber die Grundlage für andere Join-Operationen.
-

Tupelvariable

In Zwischenergebnissen

- "Zwischenergebnistabellen" aus SQL-Operationen oder einem SFW-Block können durch Tupelvariablen mit einem Namen versehen werden (Aliasing).
- Syntax:

```
SELECT attribute
  FROM relation AS alias;
```

Das Schlüsselwort `AS` ist optional.

Verwendung

- Ermöglicht es, auf Attribute der resultierenden Tabelle über den Aliasnamen zuzugreifen.
- Besonders nützlich bei Joins, um die Herkunft von Attributen zu kennzeichnen oder um mehrdeutige Attributnamen aufzulösen.

Beispiel

```
SELECT result.vineyard
  FROM (wine NATURAL JOIN producer) AS result;
```

In diesem Beispiel erhält das Ergebnis des `NATURAL JOIN` zwischen `wine` und `producer` den Aliasnamen `result`. Anschließend wird das Attribut `vineyard` über den Alias `result.vineyard` ausgewählt.

Wichtiger Hinweis

Wenn eine Tupelvariable (Alias) definiert wird, dann können Spalten nur noch mit dem Variablenamen referenziert werden und nicht mehr mit dem Tabellennamen!

Beispiel für eine ungültige Anfrage ohne Alias

```
SELECT name, year, vineyard
  FROM wine, producer
 WHERE wine.vineyard = producer.vineyard;
```

- Ist dies eine gültige SQL-Anfrage? Nein.
- Die Referenz ist nicht eindeutig: Das Resultat des Joins hat zwei Spalten mit dem Namen `vineyard`.

Richtiger Ausdruck mit Alias

```
SELECT w.name, w.year, w.vineyard  
FROM wine AS w, producer AS p  
WHERE w.vineyard = p.vineyard;
```

Hier werden die Tabellen `wine` und `producer` mit den Aliassen `w` bzw. `p` versehen, um eindeutig auf die Spalte `vineyard` zugreifen zu können.

Die SELECT-Klausel

Festlegung der Projektionsattribute

- Syntax:

```
SELECT [DISTINCT] projektionsliste
      FROM ...
```

Projektionsliste

- Liste von Spalten.
- Spezialfall: `*` steht für alle Spalten der Tabellen in der `FROM`-Klausel.
- Arithmetische Ausdrücke bestehend aus Konstanten und Spalten der Tabellen (z.B., `price * 1.19`).
- Aggregatfunktionen über Spalten der Tabellen (mehr dazu später).

Duplikateliminierung

- Die Standardeinstellung von `SELECT` ist, alle ausgewählten Tupel im Ergebnis beizubehalten, auch wenn sie Duplikate sind. Das Ergebnis ist somit ein Multimenge (Bag).

Beispiel ohne `DISTINCT`

```
SELECT name
      FROM wine;
```

name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Pinot Noir
Riesling Reserve
Chardonnay

Verwendung von `DISTINCT`

- Das Schlüsselwort `DISTINCT` vor der Projektionsliste bewirkt, dass Duplikate im Ergebnis eliminiert werden. Das Ergebnis ist eine Menge (Set) von eindeutigen Tupeln.

Beispiel mit `DISTINCT`

```
SELECT DISTINCT name  
FROM wine;
```

- Entspricht der Projektion der Relationenalgebra.

name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Riesling Reserve
Chardonnay

Sortierung

- Die `ORDER BY`-Klausel dient zur Sortierung der Ergebnismenge nach einer oder mehreren Spalten.
- Syntax:

```
SELECT spalte1, spalte2, ...
FROM tabelle
ORDER BY spalte_sortier1 [ASC|DESC], spalte_sortier2 [ASC|DESC], ...;
```

- Reihenfolge:
 - `ASC` : aufsteigend (ascending) - Standard.
 - `DESC` : absteigend (descending).
- Es können mehrere Spalten für die Sortierung angegeben werden. Die Sortierung erfolgt zuerst nach der ersten Spalte, dann innerhalb gleicher Werte der ersten Spalte nach der zweiten Spalte usw.

Beispiel

```
SELECT empid, name, rank
FROM professor
ORDER BY rank DESC, name ASC;
```

Das Ergebnis wird zuerst absteigend nach dem `rank` sortiert. Innerhalb der gleichen Ränge werden die Professoren aufsteigend nach ihrem `name` sortiert.

Die WHERE-Klausel

- Die WHERE -Klausel filtert die Tupel der in der FROM -Klausel angegebenen Tabellen basierend auf einer Bedingung.
- Syntax:

```
SELECT ...
FROM tabelle
WHERE bedingung;
```

- Die bedingung beschreibt Bedingungen, die für alle Ergebnistupel gelten müssen.
- Mögliche Operatoren und Ausdrücke in der WHERE -Klausel:
 - Vergleiche von Attributen mit anderen Attributen und/oder Konstanten (=, >, <, >=, <=, <>).
 - Bereichsoperatoren (BETWEEN, NOT BETWEEN).
 - Mengenoperatoren (IN, NOT IN).
 - Mustervergleich (LIKE, NOT LIKE).
 - Nullwertprüfung (IS NULL, IS NOT NULL).
- Bedingungen in der WHERE -Klausel können logisch mit AND, OR, NOT kombiniert werden.
- Jede Bedingung, die für einen Theta-Join definiert werden kann, kann in der WHERE -Klausel verwendet werden.

Anfrage mit einer Tabelle

```
SELECT empid, name
FROM professor
WHERE rank = 'C4';
```

professor	
empid	name
2125	Socrates
2126	Russel
2136	Curie
2137	Kant

professor			
empid	name	rank	office
2125	Socrates	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

Wichtiger Hinweis: SQL legt nicht fest, in welcher Reihenfolge Selektion, Projektion und Join ausgeführt werden. Der Anfrageoptimierer des Datenbankmanagementsystems (DBMS) bestimmt die effizienteste Ausführungsreihenfolge.

Anfrage mit mehreren Tabellen

```
SELECT name, title
FROM professor, course
WHERE professor.empid = course.taughtby AND title = 'Bioethik';
```

- Welcher Professor liest „Bioethik“ (SQL und relationale Algebra)?

Ausdruck in relationaler Algebra

$$\pi_{name,title}(\sigma_{empid=taughtby \wedge title='Bioethik'}(professor \times course))$$

Übersetzung von SQL-Anfragen in relationale Algebra

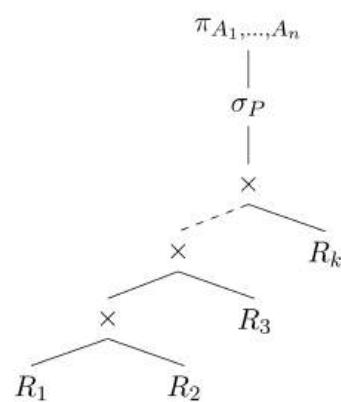
Allgemeine Form einer (geschachtelten) SQL-Anfrage

Allgemeine Form einer
(ungeschachtelten) SQL-Anfrage

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

Übersetzung in relationale Algebra

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



Mengenoperationen

- Mengenoperationen erfordern **Vereinigungskompatibilität**:
 - Gleiche Anzahl von Attributen mit kompatiblen Domänen.
 - Beide Relationen sind gleich (beide sind auf charakterbasierende Wertebereiche (unabhängig von der Länge des Strings))
 - Beide sind numerische Wertebereiche (unabhängig vom genauen Typ), z.B. integer oder float.
- Ergebnisschema: Spaltennamen der **ersten** Tabelle.

UNION

- Vereinigung zweier Mengen. Entfernt Duplikate im Endergebnis.
- Beispiel:

```
(SELECT A, B, C FROM R1)
UNION
(SELECT A, C, D FROM R2);
```

Duplikate und Mengenoperationen

- Bei Mengenoperationen ist Duplikateliminierung (`DISTINCT`) default!

UNION vs. UNION ALL

- `UNION` : Entfernt Duplikate aus dem Ergebnis.

```
(SELECT name FROM assistant)
UNION
(SELECT name FROM assistant);
```

- `UNION ALL` : Behält alle Duplikate im Ergebnis bei.

```
(SELECT name FROM assistant)
UNION ALL
(SELECT name FROM assistant);
```

Schachtelung von Anfragen

- Für Vergleiche mit Wertemengen sind Unteranfragen notwendig.
- Standardvergleiche in Verbindung mit den Quantoren `ALL` (\forall) und `ANY` (\exists).
- Spezielle Prädikate für den Zugriff auf Mengen: `IN` und `EXISTS`.

Unkorrelierte Unteranfragen

- Notation: `attribut IN (SFW-Block)`

Beispiel 1: Vergleich von Wert mit Menge von Werten

```
SELECT name
FROM professor
WHERE empid IN (SELECT taughtby
                  FROM course);
```

Beispiel 2: Welche Weine wurden in der Region "Bordeaux" produziert (mit Hilfe von `IN`)?

Gegeben seien die Tabellen:

- `wine (wineID, name, color, year, vineyard)`
- `producer (vineyard, area, region)`

```
SELECT name
FROM wine
WHERE vineyard IN (SELECT vineyard
                     FROM producer
                     WHERE region = 'Bordeaux');
```

Können wir die gleiche Anfrage formulieren, ohne `IN` zu verwenden?

Ja, mit einem Join:

```
SELECT name
FROM wine
NATURAL JOIN producer
WHERE region = 'Bordeaux';
```

Negation des IN-Prädikats

Simulation des Differenzoperators

Der relationale Algebra Ausdruck:

$$\pi_{vineyard}(producer) - \pi_{vineyard}(wine)$$

entspricht der folgenden SQL-Anfrage:

```
SELECT vineyard
FROM producer
WHERE vineyard NOT IN (
    SELECT vineyard
    FROM wine
);
```

- Das `NOT IN`-Prädikat überprüft, ob ein Wert *nicht* in der Menge der Werte ist, die von der Unterabfrage zurückgegeben werden.
- In diesem Fall werden alle `vineyard`-Werte aus der `producer`-Tabelle ausgewählt, die *nicht* in der Menge der `vineyard`-Werte aus der `wine`-Tabelle enthalten sind.

Korrelierte Unteranfragen

- Eine korrelierte Unteranfrage ist eine Unteranfrage, die sich auf eine Spalte aus der äußeren Abfrage bezieht.
- Die Unteranfrage wird für jede Zeile der äußeren Abfrage neu ausgewertet.

Beispiel 1: Verwendung von `EXISTS`

```
SELECT name
FROM professor p
WHERE EXISTS (SELECT *
               FROM course v
               WHERE v.taughtby = p.empid);
```

- **Was wird hier berechnet?** Die Namen jener Professoren, die irgendeine Lehrveranstaltung (LVA) abhalten.
- Für jeden Professor in der äußeren Abfrage prüft die Unterabfrage, ob es mindestens einen Eintrag in der Tabelle `course` gibt, dessen `taughtby`-Wert mit dem `empid` des aktuellen Professors übereinstimmt.

Beispiel 2: Auswirkung der `SELECT`-Liste in der Unteranfrage

```
SELECT name
FROM professor p
WHERE EXISTS (SELECT 42
               FROM course v
               WHERE v.taughtby = p.empid);
```

- **Was passiert, wenn wir das `SELECT *` in der Unteranfrage ändern?**
- **Erhalten wir nun andere Ergebnisse?** Nein.
- Das `EXISTS`-Prädikat prüft lediglich auf die Existenz von Zeilen, die die Bedingung erfüllen. Die tatsächlich selektierten Spalten in der Unteranfrage sind irrelevant.
- **Ist das Ergebnis der inneren Anfrage leer?** Nicht notwendigerweise.
- **Für jedes Ergebnistupel in der inneren Anfrage wird 42 zurückgegeben.** Entscheidend ist, ob *mindestens ein* Tupel die `WHERE`-Bedingung der inneren Anfrage erfüllt. Wenn ja, ist die `EXISTS`-Bedingung für die aktuelle Zeile der äußeren Abfrage wahr.

Quantoren: IN vs. EXISTS

IN

- **Bedeutung:** Ist das "linke Tupel" in der "rechten Menge" enthalten?
- **Funktionsweise:** Vergleicht einen Wert (oder ein Tupel von Werten) mit der Menge der Ergebnisse, die von der Unterabfrage zurückgegeben werden. Die Bedingung ist wahr, wenn der Wert (oder das Tupel) in dieser Menge enthalten ist.
- **Beispiel:**

```
SELECT ...
WHERE (studid, courseid) IN (SELECT studid, courseid FROM takes);
```

EXISTS

- **Bedeutung:** Ist die "rechte Seite" leer? (Gibt die Unterabfrage mindestens eine Zeile zurück?)
- **Funktionsweise:** Prüft, ob die Unterabfrage mindestens eine Zeile zurückgibt, die die WHERE -Bedingung innerhalb der Unterabfrage erfüllt. Die Bedingung ist wahr, wenn die Unterabfrage mindestens eine Zeile zurückgibt, unabhängig von den selektierten Spalten.
- **Beispiel:**

```
SELECT ...
WHERE EXISTS (SELECT * FROM takes WHERE ...);
```

Wichtiger Unterschied: IN vergleicht Werte, während EXISTS auf die Existenz von Zeilen prüft. EXISTS ist oft effizienter, besonders bei großen Unterabfragen, da es stoppt, sobald die erste passende Zeile gefunden wurde.

Quantor: ALL und ANY

- Vergleich eines Wertes mit einer Menge von Werten.
- Das Prädikat mit `ALL` ist wahr, wenn die Bedingung für *alle* Werte in der Menge der Unterabfrage erfüllt ist.

Beispiel All:

```
SELECT name
FROM student
WHERE semester >= ALL (SELECT semester
                        FROM student);
```

- Dann gibts noch `ANY`
- Das ist wahr, wenn mindestens eins davon erfüllt ist.

Beispiel Any:

```
SELECT name
FROM student
WHERE semester >= ANY (SELECT semester
                        FROM student);
```

Vergleich All und Any

Tabellen:

- `wine(wineID, name, color, year, vineyard)`
- `producer(vineyard, area, region)`

Beispiel 1: Finde den ältesten Wein (verwende ALL).

```
SELECT *
FROM wine
WHERE year <= ALL (
    SELECT year FROM wine
);
```

- `ALL`: Die Bedingung ist wahr, wenn sie für *alle* Werte in der Unterabfrage zutrifft.
- Hier: Wählt alle Weine aus, deren `year`-Wert kleiner oder gleich dem `year`-Wert *jedes* anderen Weins ist (also der älteste Wein bzw. die ältesten Weine).

Beispiel 2: Finde alle Weingüter, die Rotweine herstellen (verwende ANY).

```
SELECT *
FROM producer
WHERE vineyard = ANY (
    SELECT vineyard FROM wine
    WHERE color = 'red'
);
```

- ANY : Die Bedingung ist wahr, wenn sie für *mindestens einen* Wert in der Unterabfrage zutrifft.
 - Hier: Wählt alle Weingüter aus, deren `vineyard`-Wert mit dem `vineyard`-Wert *mindestens eines* Rotweins übereinstimmt.
-

Unteranfragen in der WHERE-Klausel ohne Quantoren

Beispiel: Vergleich eines Wertes mit einem anderen Wert

```
SELECT *
FROM grades
WHERE grade < (SELECT AVG(grade) FROM grades);
```

- Hier: Wählt alle Einträge aus der Tabelle `grades` aus, deren `grade`-Wert kleiner ist als der Durchschnitt aller `grade`-Werte in derselben Tabelle.
- Die Unterabfrage `(SELECT AVG(grade) FROM grades)` wird einmalig ausgeführt und liefert einen einzelnen Wert (den Durchschnitt).
- Dieser einzelne Wert wird dann mit dem `grade`-Wert jeder Zeile der äußeren Abfrage verglichen.

Unteranfragen in der SELECT-Klausel

- Die Unteranfrage wird für jedes Lösungstupel der äußeren Abfrage evaluiert.
- Die Unteranfrage ist korreliert.

Beispiel: Berechne einen Wert für jedes äußere Tupel.

```
SELECT empid, name, (SELECT SUM(ects)
                      FROM course
                      WHERE taughtby = empid) AS teachingLoad
FROM professor;
```

- Hier: Wählt die `empid` und den `name` aus der Tabelle `professor` aus.
- Für jede Zeile in `professor` wird die Unterabfrage ausgeführt.
- Die Unterabfrage `(SELECT SUM(ects) FROM course WHERE taughtby = empid)` berechnet die Summe der `ects` aller Kurse, die von dem aktuellen Professor (`empid` der äußeren Abfrage) unterrichtet werden.
- Das Ergebnis dieser Summe wird als Alias `teachingLoad` in die Ergebnismenge aufgenommen.
- Die Unterabfrage ist **korreliert**, da sie sich auf den Wert `empid` der äußeren Abfrage bezieht.

COALESCE()

Aufgabe: Gib eine Liste aller Studierenden aus inkl. deren Noten für abgeschlossene Kurse. Für Studierende, die keinen Kurs besucht haben, sollen `courseID` und `grade` den Wert `0` annehmen.

```
SELECT s.studID, s.name,
       COALESCE(g.courseID, 0),
       COALESCE(g.grade, 0)
  FROM student s LEFT OUTER JOIN grades g
    ON s.studID = g.studID;
```

- `COALESCE(Ausdruck1, Ausdruck2, ...)` : Gibt den ersten Ausdruck zurück, der nicht `NULL` ist. Sind alle Ausdrücke `NULL`, wird `NULL` zurückgegeben.
- Im Beispiel:
 - `COALESCE(g.courseID, 0)` : Wenn `g.courseID` für einen Studenten `NULL` ist (weil er keinen Kurs belegt hat), wird `0` zurückgegeben. Andernfalls wird der tatsächliche `g.courseID`-Wert verwendet.
 - `COALESCE(g.grade, 0)` : Wenn `g.grade` für einen Studenten `NULL` ist, wird `0` zurückgegeben. Andernfalls wird die tatsächliche `g.grade` verwendet.
- `LEFT OUTER JOIN` : Stellt sicher, dass *alle* Zeilen aus der linken Tabelle (`student`) im Ergebnis enthalten sind. Wenn es keine übereinstimmende Zeile in der rechten Tabelle (`grades`) gibt, werden die Spalten aus `grades` als `NULL` behandelt.

Achtung: Die Datentypen der Ausdrücke in `COALESCE()` müssen kompatibel sein. Im Beispiel wird davon ausgegangen, dass `courseID` und `grade` numerische Typen sind oder implizit in numerische Typen konvertiert werden können.

Allquantifizierte Anfragen

Aufgabe: Welche Studierenden haben *alle* 4-ECTS Lehrveranstaltungen gehört?

Tabellen:

- student(studID, name, semester)
- takes(studID, courseID)
- course(courseID, title, ects, taughtBy)

Relationale Algebra:

$$\text{takes} \div \pi_{\text{courseID}}(\sigma_{\text{ects}=4}(\text{course}))$$

- $\sigma_{\text{ects}=4}(\text{course})$: Selektiert alle Kurse mit 4 ECTS-Punkten.
- $\pi_{\text{courseID}}(\sigma_{\text{ects}=4}(\text{course}))$: Projiziert die `courseID`s der 4-ECTS-Kurse.
- \div : Divisionsoperator. $A \div B$ liefert alle Tupel t aus A , sodass für *jedes* Tupel u in B das Tupel (t, u) in A existiert.
- Im Kontext: Liefert alle `studID`s aus `takes`, sodass für *jede* `courseID` eines 4-ECTS-Kurses ein Eintrag mit dieser `studID` und der `courseID` in `takes` existiert.

Allquantifizierte Anfragen - Nicht direkt in SQL

SQL stellt **keinen Allquantor** direkt zur Verfügung.

Realisierung durch Logische Äquivalenz (mittels 2 x NOT EXISTS):

Die Anfrage "Finde alle Studierenden, die alle 4-ECTS Lehrveranstaltungen gehört haben" ist logisch äquivalent zu "Finde alle Studierenden, für die es *keine* 4-ECTS Lehrveranstaltung gibt, die sie *nicht* gehört haben."

Dies kann in SQL mit zwei `NOT EXISTS`-Klauseln umgesetzt werden

Umformulierung in äquivalente Anfrage mit Negation und Existenzquantor (Prädikatenlogik)

$$\forall x F(x) \Leftrightarrow \neg \exists x (\neg F(x))$$

Allquantifizierte Anfragen - "Logische Umformung"

Umformulierung in äquivalente Anfrage mit Negation und Existenzquantor (Prädikatenlogik):

$$\forall x F(x) \Leftrightarrow \neg \exists x (\neg F(x))$$

- \forall : Allquantor ("für alle")
- \exists : Existenzquantor ("es existiert mindestens ein")

- \neg : Negation ("nicht")
- $F(x)$: Eine Aussage über x

Beispiel: Welche Studierenden haben alle 4-ECTS Lehrveranstaltungen gehört?

\iff Suchen Sie jene Studierenden, für die gilt: sie haben alle 4-ECTS Lehrveranstaltungen gehört.

\iff Suchen Sie jene Studierenden, für die **nicht** gilt: es gibt eine 4-ECTS Lehrveranstaltung, die der/die Studierende **nicht** gehört hat.

Diese logische Umformung ist die Grundlage für die Implementierung allquantifizierter Anfragen in SQL mithilfe von `NOT EXISTS`.

SQL-Umsetzung folgt nun direkt aus:

Suchen Sie jene Studierende, für die **nicht** gilt: es gibt eine 4-ECTS Lehrveranstaltung, für die **nicht** gilt: die/der Studierende hat diese Lehrveranstaltung gehört.

```
SELECT s.*  
FROM student s  
WHERE NOT EXISTS (SELECT *  
                   FROM course c  
                   WHERE c.ects = 4  
                   AND c.courseID NOT IN (SELECT t.courseID  
                                         FROM takes t  
                                         WHERE t.studID = s.studID));
```

- Die äußere `NOT EXISTS`-Klausel prüft für jeden Studenten `s` aus der Tabelle `student`, ob die innere `SELECT`-Anweisung **keine** Zeilen zurückliefert.
- Die innere `SELECT`-Anweisung findet alle 4-ECTS-Kurse (`WHERE c.ects = 4`), deren `courseID` **nicht** in der Menge der `courseID`'s enthalten ist, die der aktuelle Student `s` in der Tabelle `takes` belegt hat (`WHERE t.studID = s.studID`).
- Wenn die innere `SELECT`-Anweisung **keine** Zeilen zurückliefert, bedeutet das, dass der Student für jeden 4-ECTS-Kurs einen Eintrag in der Tabelle `takes` hat, also alle 4-ECTS-Kurse gehört hat. In diesem Fall ist die Bedingung der äußeren `NOT EXISTS`-Klausel wahr, und der Student wird ausgewählt.

Mächtigkeit des SQL-Kerns

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

Joins

Join Varianten:

- CROSS JOIN
- NATURAL JOIN
- JOIN oder INNER JOIN
- LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN

Standard Formulierung (für INNER JOIN):

```
SELECT *
FROM R1, R2
WHERE R1.A = R2.B;
```

- Kombiniert Zeilen aus `R1` und `R2`, bei denen der Wert der Spalte `A` in `R1` gleich dem Wert der Spalte `B` in `R2` ist.
- Nur übereinstimmende Zeilen aus beiden Tabellen werden im Ergebnis berücksichtigt.

Alternative Formulierung (expliziter INNER JOIN):

```
SELECT *
FROM R1 JOIN R2 ON R1.A = R2.B;
```

- Semantisch identisch zur Standardformulierung des Inner Joins.
- Die `JOIN`-Klausel in Verbindung mit der `ON`-Klausel spezifiziert explizit die Join-Bedingung.

Right outer Join

```
SELECT *
FROM grades g RIGHT OUTER JOIN student s ON g.studid = s.studid;
```

g.studid	g.courseid	g.empid	g.grade	s.studid	s.name	s.semester
⊥	⊥	⊥	⊥	24002	Xenokrates	18
25403	5041	2125	2.0	25403	Jonas	12
⊥	⊥	⊥	⊥	26120	Fichte	10
⊥	⊥	⊥	⊥	26830	Aristoxenos	8
27550	4630	2137	2.0	27550	Schopenhauer	6
28106	5001	2126	1.0	28106	Carnap	3
⊥	⊥	⊥	⊥	29120	Theophrastos	2
⊥	⊥	⊥	⊥	29555	Feuerbach	2

Für jedes Tupel welches auf der rechten Seite ist, haben wir eine Ausgabe auch wenn sie keine Noten haben.

Das gibt es auch als **Left outer Join** und **Full outer Join**

Aggregatfunktionen

Wie können wir die folgende Anfrage in SQL formulieren?

- Den durchschnittlichen Preis aller Artikel im Angebot
- Gesamtumsatz aller verkauften Produkte

Aggregatfunktionen berechnen neue Werte für eine Spalte.

Verfügbare Aggregatfunktionen:

`AVG`, `MAX`, `MIN`, `COUNT`, `SUM`

Duplikate

Der Argumentspalte (außer im Falle `COUNT(*)`) kann optional ein `DISTINCT` oder ein `ALL` hinzugefügt werden.

- `DISTINCT`: Bevor die Aggregatfunktion ausgewertet wird, werden Duplikate entfernt.
- `ALL`: Duplikate werden für die Auswertung herangezogen (Standard/Default).

Null-Werte werden vor der Auswertung entfernt (außer im Falle `COUNT(*)`).

Beispiel

Anzahl von Weinen

```
SELECT COUNT(*) AS number
FROM wine;
```

Die Anzahl von unterschiedlichen Regionen, wo Weine produziert werden

```
SELECT COUNT(DISTINCT region)
FROM producer;
```

Die Namen und Jahre der Weine, die älter sind als der Durchschnitt

```
SELECT name, year
FROM wine
WHERE year < (SELECT AVG(year) FROM wine);
```

Aggregatfunktionen in der WHERE-Klausel

⇒ Einsatz in Konstanten-Selektionen der `WHERE`-Klausel möglich.

Beispiel: Alle Weingüter, die nur einen Wein produzieren.

```
SELECT * FROM producer e
WHERE 1 = (SELECT COUNT(*) FROM wine w
            WHERE w.vineyard = e.vineyard);
```

- Die Unterabfrage `(SELECT COUNT(*) FROM wine w WHERE w.vineyard = e.vineyard)` zählt für jedes Weingut `e.vineyard` die Anzahl der Weine in der Tabelle `wine`, die von diesem Weingut produziert werden.
- Das Ergebnis dieser Unterabfrage ist ein einzelner numerischer Wert (die Anzahl der Weine).
- Die äußere `WHERE`-Klausel vergleicht diesen einzelnen Wert mit der Konstanten `1`. Nur Weingüter, für die die Anzahl der produzierten Weine gleich 1 ist, werden im Ergebnis ausgewählt.

Verschachtelung

Nicht erlaubt!, daher:

Falsch:

```
SELECT f1(f2(A)) AS result
FROM R ...;
```

- Direkte Schachtelung von Aggregatfunktionen (`f1` angewendet auf das Ergebnis von `f2(A)`) ist in SQL nicht zulässig.

Möglich (durch Verwendung einer Subquery):

```
SELECT f1(temp) AS result
FROM ( SELECT f2(A) AS temp FROM R ... ) AS subquery;
```

- Um eine ähnliche Funktionalität zu erreichen, kann eine Subquery verwendet werden.
- Zuerst wird die innere Aggregatfunktion `f2(A)` in der Subquery berechnet und das Ergebnis unter dem Alias `temp` gespeichert.
- Anschließend kann die äußere Aggregatfunktion `f1` auf diese temporäre Spalte `temp` angewendet werden.

Gruppierung

Notation:

```
SELECT ...
FROM ...
(WHERE ... GROUP BY attributliste )
(HAVING bedingung );
```

Aggregatsfunktion in Kombination mit Gruppierung

Berechnung der Funktionen pro Gruppe:

```
SELECT taughtBy, SUM(ects)
FROM course
GROUP BY taughtBy;
```

- GROUP BY taughtBy : Gruppiert die Tupel der Tabelle course nach dem Wert des Attributs taughtBy .
- Alle Tupel, die den gleichen Wert für das Attribut taughtBy haben, werden zu einer Gruppe zusammengefasst.
- Die Aggregatfunktion SUM(ects) wird dann für jede dieser Gruppen separat berechnet.
- Das Ergebnis enthält für jeden taughtBy -Wert die Summe der ects der von dieser Person unterrichteten Kurse.

Beispiel

Was ist das Problem mit folgendem Ausdruck?

```
SELECT rank, COUNT(empId), name
FROM professor
GROUP BY rank;
```

- SQL erzeugt ein Lösungstupel pro Gruppe.
- Alle Spalten in der SELECT -Klausel müssen entweder in der GROUP BY -Klausel aufgeführt werden oder in einer Aggregatfunktion involviert sein.
- Im obigen Beispiel ist die Spalte name weder in der GROUP BY -Klausel (rank) enthalten noch wird sie in einer Aggregatfunktion verwendet. Da mehrere Professoren denselben rank haben können, ist unklar, welcher name in der Ergebnismenge für diese Gruppe angezeigt werden soll. Dies führt zu einem Fehler in den meisten SQL-Implementierungen.

Jeder Professor hat einen eigenen Namen und wir können nur einen Wert pro Gruppe haben, daher hat das Programm keine Ahnung was es ausgeben soll.

Man kann die Anfrage **retten** indem man den **Namen weglässt**.

Weitere Beispiele:

```
SELECT COUNT(*)
FROM course
GROUP BY taughtBy
```

richtig

Das geht, weil wir hier nachschauen, wie viele individuelle taughtBy Werte es gibt.

```
SELECT taughtBy, COUNT(*)
FROM course
GROUP BY ects
```

falsch

Das geht nicht, weil die `GROUP BY`-Klausel nicht mit den ausgewählten Spalten (`taughtBy` und `COUNT(*)`) übereinstimmt

```
SELECT taughtBy, COUNT(*)
FROM course
GROUP BY ects, taughtBy
```

richtig

Das stimmt, weil die `GROUP BY`-Klausel jetzt alle nicht-aggregierten Spalten enthält, die im `SELECT`-Teil der Abfrage stehen. Lass uns das genauer anschauen:

- `SELECT taughtBy, COUNT(*)`: Du wählst den Namen des Dozenten (`taughtBy`) und die Anzahl der Zeilen (`COUNT(*)`) aus der Tabelle `course` aus.
- `GROUP BY ects, taughtBy`: Du gruppierst die Zeilen nach *zwei* Spalten: `ects` und `taughtBy`.

Having-Klausel

```
SELECT taughtBy, name, SUM(ects)
FROM course, professor
WHERE taughtBy = empID AND rank = 'C4'
GROUP BY taughtBy, name
HAVING AVG(ects) >= 3;
```

Die HAVING-Klausel drückt eine weitere Bedingung aus, die Gruppen erfüllen müssen, um im Resultat vorzukommen.

Was wird hier berechnet? Was sind die Schritte?

Ausführen einer Anfrage mit Group by und Having

FROM course, professor

course × professor							
courseid	title	ects	taughtBy	empID	name	rank	office
5001	Basics	4	2137	2125	Socrates	C4	226
5041	Ethics	4	2125	2125	Socrates	C4	226
...
4630	Constructive Criticism	4	2137	2137	Kant	C4	7



WHERE taughtBy = empID AND rank = 'C4'

Dann bekommen wir das hier:

course × professor							
courseid	title	ects	taughtBy	empID	name	rank	office
5001	Basics	4	2137	2137	Kant	C4	7
5041	Ethics	4	2125	2125	Socrates	C4	226
5043	Theory of Cognition	3	2126	2126	Russel	C4	232
5049	DBS	2	2125	2125	Socrates	C4	226
4052	Logics	4	2125	2125	Socrates	C4	226
5052	Theory of Science	3	2126	2126	Russel	C4	232
5216	Bioethics	2	2126	2126	Russel	C4	232
4630	Constructive Criticism	4	2137	2137	Kant	C4	7



GROUP BY taughtBy, name

und führen `GROUP BY` aus:

course × professor							
taughtBy	name	courseid	title	ects	empID	rank	office
2125	Socrates	5041	Ethics	4	2125	C4	226
		5049	DBS	2	2125	C4	226
		4052	Logics	4	2125	C4	226
2126	Russel	5043	Theory of Cognition	3	2126	C4	232
		5052	Theory of Science	3	2126	C4	232
		5216	Bioethics	2	2126	C4	232
2137	Kant	5001	Basics	4	2137	C4	7
		4630	Constructive Criticism	4	2137	C4	7



`HAVING AVG(ects) >= 3`

und jetzt noch `HAVING` dann haben wir das:

course × professor							
taughtBy	name	courseid	title	ects	empID	rank	office
2125	Socrates	5041	Ethics	4	2125	C4	226
		5049	DBS	2	2125	C4	226
		4052	Logics	4	2125	C4	226
2137	Kant	5001	Basics	4	2137	C4	7
		4630	Constructive Criticism	4	2137	C4	7



`SUM(ects)`

Und jetzt berechnen wir noch die Summe:

"Pasted image 20250428144757.png" could not be found.

Und am Ende geben wir dann nur das aus, was in unserem `Select` steht:

taughtBy	name	SUM(ects)
2125	Socrates	10
2137	Kant	8

Null-Werte

Manchmal sehr überraschende Anfrageergebnisse, wenn Null-Werte vorkommen.

```
SELECT COUNT(semester)
FROM student
WHERE semester < 13 OR semester >= 13;
```

```
SELECT COUNT(semester) FROM student;
```

Beide Anfragen liefern das gleiche Ergebnis, weil Tupel mit Null-Werten in der Spalte `semester` nicht gezählt werden.

Erklärung:

- `COUNT(Spalte)` zählt nur die Zeilen, in denen die angegebene `Spalte` einen **nicht-NULL** Wert hat.
- In der ersten Anfrage filtert die `WHERE` -Klausel nach `semester` -Werten, die kleiner als 13 oder größer oder gleich 13 sind. Diese Bedingung schließt `NULL` -Werte nicht explizit aus oder ein. Da `NULL` weder kleiner als 13 noch größer oder gleich 13 ist, werden Zeilen mit `NULL` in der Spalte `semester` durch die `WHERE` -Klausel nicht gefiltert, aber von `COUNT(semester)` dennoch nicht gezählt.
- Die zweite Anfrage zählt alle nicht-NULL Werte in der Spalte `semester` über die gesamte Tabelle.
- Da `COUNT(semester)` in beiden Fällen nur nicht-NULL Werte in der Spalte `semester` berücksichtigt, liefern beide Anfragen dasselbe Ergebnis (die Anzahl der Studierenden, deren Semester-Wert nicht `NULL` ist).

Auswertung von Null-Values

Arithmetische Ausdrücke:

- Null-Werte werden "propagiert".
- `null + 1` \Rightarrow `null`
- `null * 0` \Rightarrow `null`

\Rightarrow Sobald irgendwo `null` steht kommt `null` raus

Vergleichsoperatoren:

- SQL hat eine dreiwertige Logik: `true`, `false` und `unknown`.
- Wenn mindestens ein Argument `null` ist, dann ist das Ergebnis `unknown`.
- `studid = 5` \Rightarrow `unknown` immer, wenn `studid null` ist.

⇒ Sobald null in einem Vergleich vorkommt kommt unknown raus und nicht mehr true oder false

Test, ob ein Attribut den Wert null hat:

- Attribut IS NULL

nur so kann man testen ob gleich null ist.

Zusammenfassend:

Logische Ausdrücke werden gemäß folgender Tabellen berechnet

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

Null-Werte in der WHERE-Klausel und Gruppierung

WHERE-Klausel:

- Die WHERE -Klausel reicht nur Tupel weiter, die zu true evaluieren.
- Tupel, die zu unknown evaluieren (z.B. bei Vergleichen mit NULL), sind nicht Teil des Resultats.

Gruppierung (GROUP BY):

- null wird als eigener, distinkter Wert interpretiert.
- Tupel mit null in der Gruppierungsspalte werden in einer eigenen Gruppe zusammengefasst.

Berechnung der Vorgänger

Aufgabe: Welche Vorlesungen müssen besucht werden, um die Vorlesung "Theory of Science" verstehen zu können?

Tabellen:

- `requires: { predecessor, successor }`
- `course: { courseid, title, ects, taughtBy }`

```
SELECT predecessor
FROM requires, course
WHERE successor = courseid
AND title = 'Theory of Science';
```

Erläuterung:

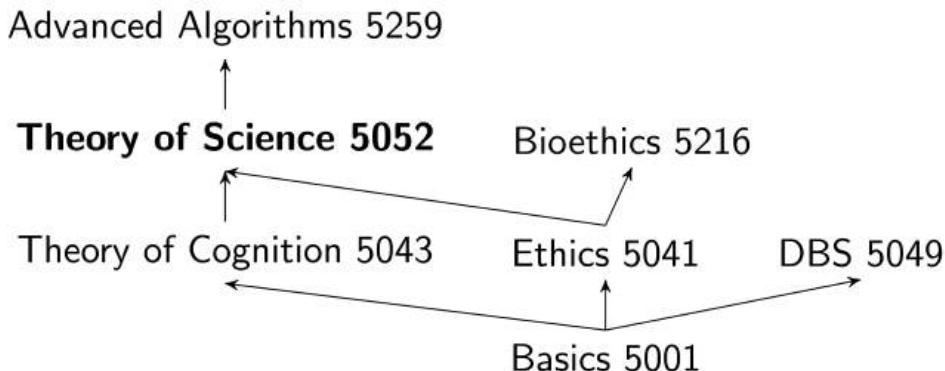
- Die Anfrage verbindet die Tabellen `requires` und `course` über die Bedingung `successor = courseid`.
- Anschließend filtert sie nach den Einträgen, bei denen der `title` der Vorlesung in der Tabelle `course` gleich 'Theory of Science' ist.
- Die `SELECT`-Klausel gibt dann den `predecessor` (die vorausgesetzte Vorlesung) für diese Vorlesung aus.

Wichtiger Hinweis: Hier werden allerdings nur die **direkten** Vorgänger berechnet. Um alle transitiven Vorgänger (Vorgänger von Vorgängern usw.) zu finden, wären rekursive Anfragen notwendig, die im Standard-SQL nicht direkt unterstützt werden (können aber in einigen Datenbankmanagementsystemen mit speziellen Erweiterungen realisiert werden).

Beispiel

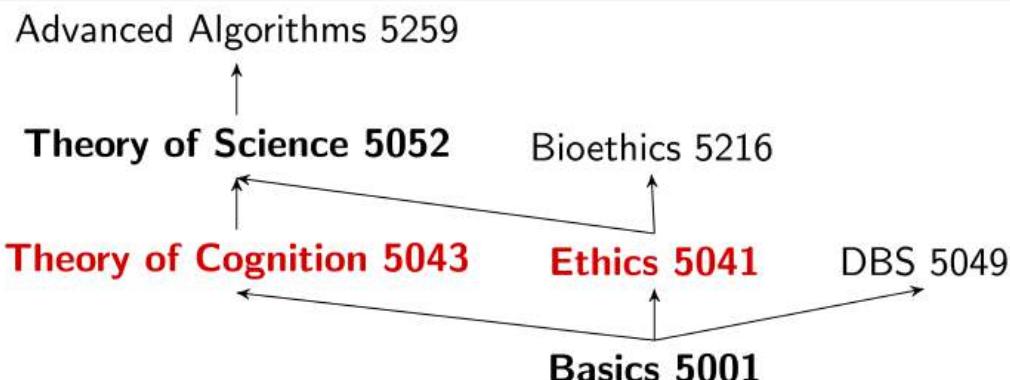
```
SELECT r2.predecessor
FROM course c, requires r1, requires r2
WHERE c.title='Theory of Science' AND
      c.courseid = r1.successor AND
      r1.predecessor = r2.successor;
```

Aus welchen Vorlesungen besteht das Ergebnis dieser Anfrage?



```
SELECT r2.predecessor
FROM course c, requires r1, requires r2
WHERE c.title='Theory of Science' AND
      c.courseid = r1.successor AND
      r1.predecessor = r2.successor;
```

Aus welchen Vorlesungen besteht das Ergebnis dieser Anfrage?



Vorgänger der Tiefe n

```
SELECT r1.predecessor
FROM requires r1,
     ...
     requires rn_minus_1,
     requires rn,
     course c
WHERE r1.successor = r2.predecessor AND
      ... AND
      rn_minus_1.successor = rn.predecessor AND
      rn.successor = c.courseid AND
      c.title = 'Theory of Science';
```

Wie berechnet man die ganze Liste von Vorgängern beliebiger Tiefe?

Tiefe 1 UNION Tiefe 2 UNION Tiefe 3 UNION ...

Rekursion in SQL

Beispiel 1: Summe der Zahlen von 1 bis 100

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

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

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

Anfrage für Voraussetzungen von Kursen

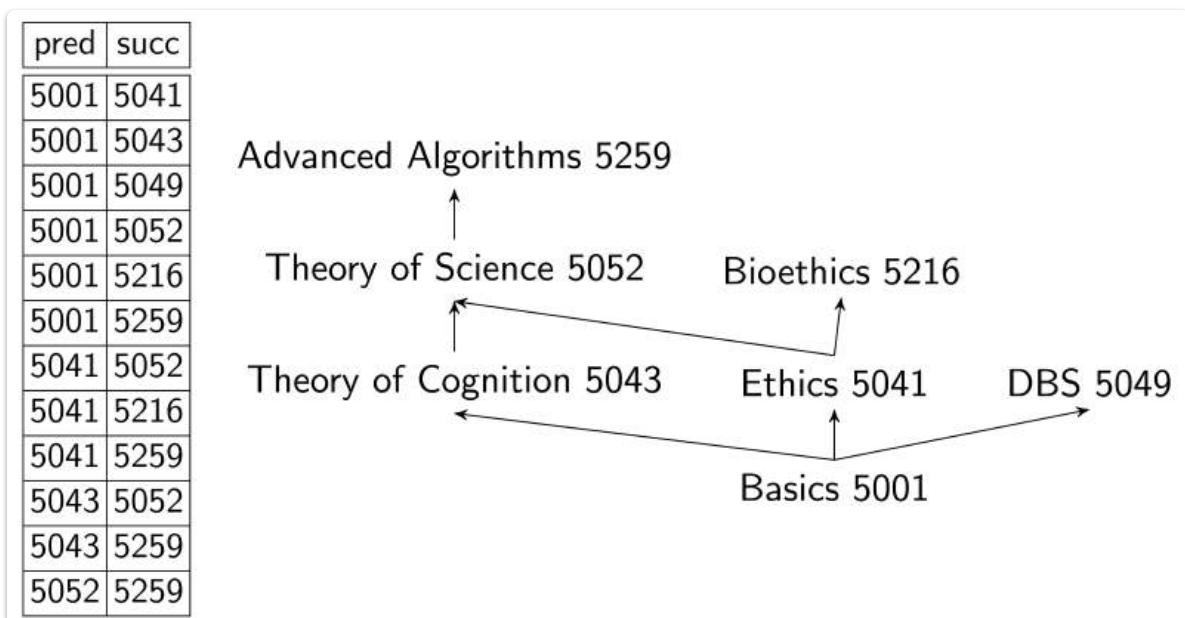
Beispiel 2: Transitive Voraussetzungen von Kursen

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

- `WITH RECURSIVE transitiveCourse (pred, succ) AS (...)`: Definiert eine rekursive CTE namens `transitiveCourse` mit den Spalten `pred` (Voraussetzung) und `succ` (Nachfolger).

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

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



Unendliche Rekursion verhindern

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

Beispiel zur Verhinderung unendlicher Rekursion durch Tiefenbegrenzung:

```

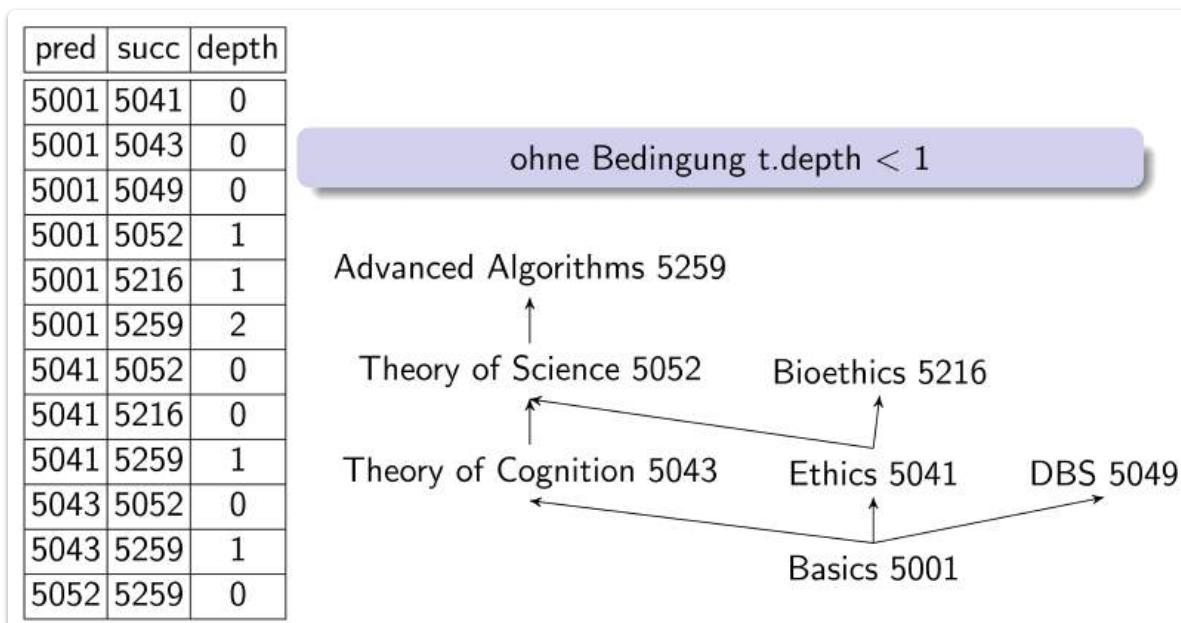
WITH RECURSIVE transitiveCourse (pred, succ, depth) AS (
  SELECT predecessor, successor, 0
  FROM requires
  UNION ALL
  SELECT DISTINCT tc.pred, r.successor, tc.depth + 1
  FROM transitiveCourse tc, requires r
  WHERE tc.succ = r.predecessor
  AND tc.depth < 1 -- Begrenzung der Rekursionstiefe
)
 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

```
)
SELECT *
FROM transitiveCourse
ORDER BY (pred, succ) ASC;
```

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

Hinweis: Der Wert `< 1` in diesem Beispiel ist eine sehr strenge Begrenzung und würde nur direkte Voraussetzungen und deren unmittelbare Voraussetzungen berücksichtigen. In realen Szenarien wäre eine höhere Tiefenbegrenzung oder eine andere Abbruchlogik erforderlich, abhängig von der Struktur der Daten. Die meisten DBMS erlauben auch eine Konfiguration der maximalen Rekursionstiefe auf Systemebene.

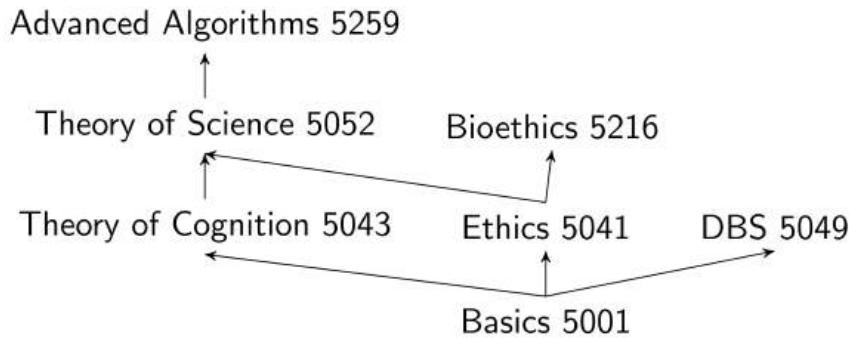
Ohne der Bedingung:



Mit der Bedingung:

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

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



Noch ein Beispiel von Rekursion

Alle Voraussetzungen einer bestimmten Vorlesung

```

WITH RECURSIVE transitiveCourse (pred, succ) AS (
    SELECT predecessor, successor
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
)
SELECT c2.title
FROM transitiveCourse tv, course c1, course c2
WHERE tv.succ = c1.courseid
AND c1.title = 'Theory of Science'
AND tv.pred = c2.courseid;
    
```

Was genau ist das Resultat dieser Anfrage?

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

Erläuterung der Anfrage:

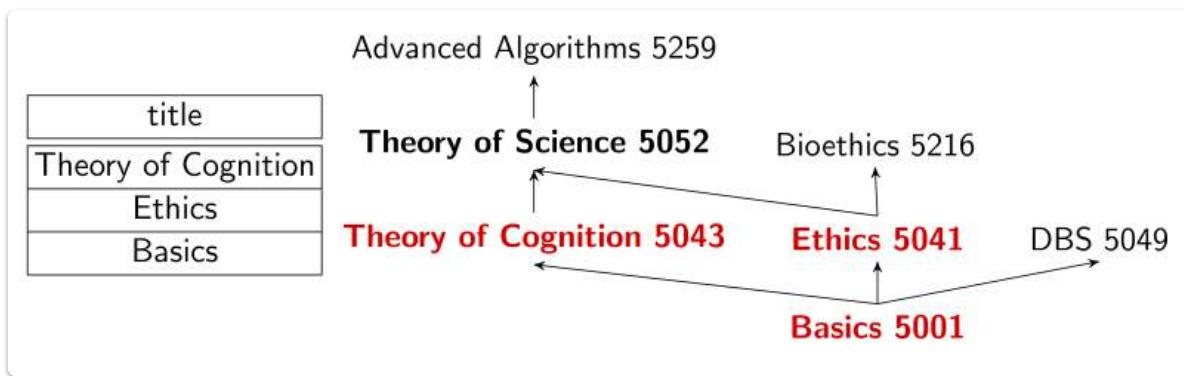
1. Rekursive CTE `transitiveCourse`:

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

2. Hauptabfrage:

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

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



Anzahl der Ergebnisse begrenzen

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

SQL 2003: Window Function

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

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

SQL 2008: Fetch-First-Klausel

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

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

Nicht-Standard-Syntax

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

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

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

Unterstützt von Oracle

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

Unterstützt von MS SQL server

LIMIT wird in der Prüfung akzeptiert.

Abstraktionsebenen eines Datenbanksystems

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



Datenunabhängigkeit:

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

Mehr zu **Abstraktion** siehe: [EP2](#)

Beispiele für Views:

Beispiel 1: View `profsAndtheirCourses`

```

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

SELECT * FROM profsAndtheirCourses;
  
```

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

Beispiel 2: View `ectsPerStud`

```

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

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

SELECT sum FROM ectsPerStud;
```

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

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

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

Views ändern

`CREATE OR REPLACE VIEW`:

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

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

`DROP VIEW`:

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

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

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

Alternative:

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

Sichten vs. materialisierte Sichten

(Dynamische) Sicht (View)

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

Materialisierte Sicht (Materialized View)

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

Was ist die bessere Wahl? Laufzeit vs. Updates

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

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

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

Updates und Views

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

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

Was ist hier das Problem?

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

Noch ein Beispiel:

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

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

Tabellen:

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

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

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

Änderbarkeit von Sichten in SQL

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

in SQL-92

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

in SQL-99

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

Änderbarkeit von Views in SQL

Views in Postgres sind nicht materialisiert!

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

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

ERROR: cannot insert into a view.

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

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

Integritätsbedingungen

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

Welche Integritätsbedingungen haben wir in der Vorlesung schon kennengelernt?

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

Statische Integritätsbedingungen

Statische Integritätsbedingungen müssen von jedem Zustand der Datenbank erfüllt werden.

Beispiele:

- NOT NULL -Constraint:

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

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

- CHECK -Constraint (Einschränkung des Wertebereichs):

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

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

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

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

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

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

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

Dynamische Integritätsbedingungen

Referentielle Integrität

Referentielle Integrität bedeutet

Fremdschlüssel müssen auf existierende Tupel verweisen oder einen Nullwert enthalten.

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

```
insert into course
values (5100, 'Spying for Dummies', 4, 007);
```

Wie kann dieses „insert“ verhindert werden?

Festlegung von Schlüsseln

Kandidatenschlüssel

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

Primärschlüssel

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

Fremdschlüssel

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

Verhalten bei Änderungsoperationen

Dynamische Integritätsbedingungen müssen von Zustandsänderungen erfüllt werden.

Bei Änderung von referenzierten Daten haben wir mindestens 3 Optionen

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

In Postgres außerdem

- Auf Defaultwert setzen (SET DEFAULT)

Beispiel

Update on table R

S		R	
	α		κ
	κ_1		κ_1
	κ_2		κ_2
...

UPDATE R

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

DELETE FROM R

WHERE $\kappa = \kappa_1$

Wie soll damit umgegangen werden?

Da gibts mehrere Optionen:

Option 1

CREATE TABLE S
(...,
 α integer REFERENCES R(κ));

S		R	
	α		κ
	κ_1		κ_1
	κ_2		κ_2
...

Ergebnis der Änderungsoperation: DB bleibt unverändert

Option 2

CREATE TABLE S

(...,
 α integer REFERENCES R(κ)
ON UPDATE CASCADE);

S		R	
	α		$\underline{\kappa}$
	κ'_1		κ'_1
	κ_2		κ_2
...

CREATE TABLE S

(...,
 α integer REFERENCES R(κ)
ON DELETE CASCADE);

S		R	
	α		$\underline{\kappa}$
	κ_2		κ_2
...

Ergebnis der Änderungsoperation:
Schlüssel in R und S geändert.

Ergebnis der Änderungsoperation:
Schlüssel in R und S gelöscht.

Option 3:

CREATE TABLE S

(...,
 α integer REFERENCES R(κ)
ON UPDATE SET NULL);

S		R	
	α		$\underline{\kappa}$
	null		κ'_1
	κ_2		κ_2
...

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

CREATE TABLE S

(...,
 α integer REFERENCES R(κ)
ON DELETE SET NULL);

S		R	
	α		$\underline{\kappa}$
	null		...
	κ_2		κ_2
...

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

Kaskadierendes Löschen

Referentielle Integrität: ON DELETE CASCADE

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

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

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

Beispiel siehe Slides: [hier](#)

Komplexe Konsistenzbedingung

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

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

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

Zusammenfassung

- SQL ist mehr als eine Anfragesprache.
- DDL, DML, DCL, TCL, DQL.
- SQL (DQL) ist eine deklarative Anfragesprache.
- SFW-Block ist die Grundlage.
- SQL bietet Operationen an, aus relationaler Algebra bekannt.
- Duplikate.
- Komplexe Bedingungen (=, <>, !=), >, <, >=, <=, LIKE, BETWEEN, IN, AND, OR, NOT, IS, ...).
- Geschachtelte Anfragen:
 - Korreliert und unkorreliert.
 - IN, EXISTS, ANY, ALL.
- Gruppierung und Aggregation:
 - SELECT -Klausel kann nur Spalten referenzieren, die in der GROUP BY -Klausel vorkommen oder in Aggregatfunktionen.
 - HAVING, um gewisse Gruppen zu selektieren.
- Temporäre Tabellen (WITH).
- Null-Werte führen möglicherweise zu unerwarteten Ergebnissen.
- Rekursive Anfragen.
- Größe des Ergebnisses beschränken (LIMIT, FETCH FIRST n ROWS ONLY, ...).
- Views können wie „Makros“ verwendet werden.
- Materialisierte Sichten für schnellere Anfragebearbeitung.
- Integritätsbedingungen, um Inkonsistenzen in den Daten zu vermeiden.
- CASCADE vorsichtig verwenden.
- Trigger sind mächtiger als CHECK -Bedingungen aber auch „gefährlicher“.

5. Transaktionen

Einführung

Beispiel einer typischen Bankanwendung:

- ① Lese den Kontostand von A in die Variable a : $\text{read}(A, a)$;
- ② Reduziere den Kontostand um 50 Euro: $a := a - 50$;
- ③ Schreibe den neuen Kontostand in die Datenbank: $\text{write}(A, a)$;
- ④ Lese den Kontostand von B in die Variable b : $\text{read}(B, b)$;
- ⑤ Erhöhe den Kontostand um 50 Euro: $b := b + 50$;
- ⑥ Schreibe den neuen Kontostand in die Datenbank: $\text{write}(B, b)$;

Jetzt ist natürlich die Frage, was hier schiefgehen kann.

Die Antwort dafür ist:

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

Transaktion

Was ist eine Transaktion?

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

Eigenschaften von Transaktionen: ACID

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

Atomicity (Atomarität)

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

Consistency (Konsistenz)

- **Prinzip:** Die Ausführung einer Transaktion in Isolation (ohne gleichzeitige andere Transaktionen) bewahrt den konsistenten Zustand der Datenbank.
- **Gewährleistung:**
 - Einhaltung definierter Constraints (z.B. Datentypen, Wertebereiche).
 - Durchführung von Checks (Überprüfungen von Bedingungen).
 - Einhaltung von Assertions (Zusicherungen über den Datenbankzustand).
- **Anwendungsdefinition:** Die Semantik der Konsistenz wird auch durch die Anwendung selbst definiert.
 - **Beispiel:** Bei einer Banküberweisung muss die Summe des abgebenden und empfangenden Kontos gleich bleiben – es darf kein Geld "entstehen" oder "verschwinden". Die Gesamtsumme aller beteiligten Konten muss vor und nach der Transaktion identisch sein.

Isolation

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

Durability (Dauerhaftigkeit)

- **Prinzip:** Änderungen, die von einer erfolgreich abgeschlossenen Transaktion (Commit) an der Datenbank vorgenommen wurden, müssen dauerhaft gespeichert bleiben und dürfen auch bei Systemfehlern (z.B. Stromausfall, Festplattenausfall) nicht verloren gehen.

- **Gewährleistung:**

- Typischerweise durch die Verwendung von Log-Einträgen (Protokolldateien), in denen die durchgeführten Änderungen persistent protokolliert werden.
 - Im Fehlerfall können die Änderungen anhand des Logs wiederhergestellt werden (Recovery-Mechanismen).
-

Operationen auf Transaktionsebene

begin of transaction (BOT)

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

In SQL BEGIN;

commit

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

In SQL COMMIT;

rollback oder abort

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

In SQL ROLLBACK;

"autocommit" Modus

Jeder Befehl wird in einer eigenen Transaktion ausgeführt.

Einfache Konsistenzprüfung (Checks)

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

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

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

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

Savepoints

Transaktionen von langer Dauer können zusätzlich Savepoints definieren

`SAVEPOINT savepoint_name;`

Definiert einen Punkt/Zustand innerhalb einer Transaktion.

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

`ROLLBACK TO savepoint_name;`

Setzt die aktive Transaktion zurück bis zum Savepoints savepoint_name

Man kann einfach Zwischenspeicherungen machen.

Ein Beispiel dazu wäre:

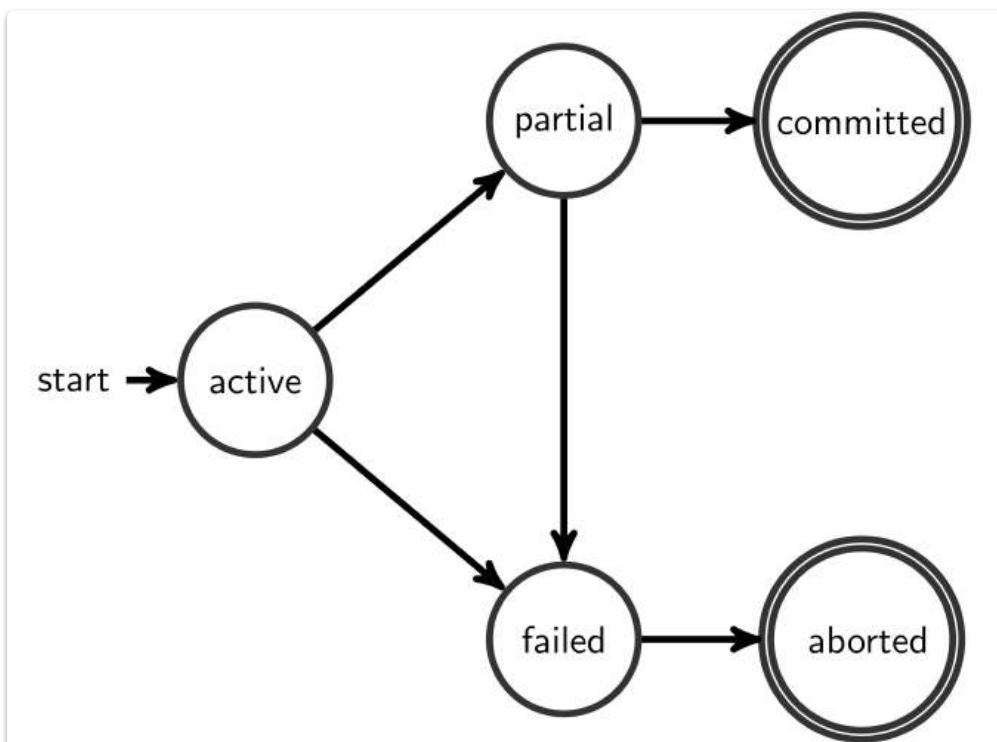
```
BEGIN;
INSERT INTO tab VALUES ...
SAVEPOINT A;
INSERT INTO tab VALUES...
SAVEPOINT B;
SELECT * FROM tab;
ROLLBACK TO A;
SELECT * FROM tab;
```

Beenden einer Transaktion

Für den Abschluss einer Transaktion gibt es drei Möglichkeiten:

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

Zustandsdiagramm für Transaktionen



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

Realisierung vom DBMS

Die beiden wichtigsten Komponenten der Transaktionsverwaltung sind:

Mehrbenutzersynchronisation (Isolation)

- **Ziel:** Semantische Korrektheit bei Nebenläufigkeit.
- **Vorteil:** Hoher Durchsatz.
- **Serialisierbarkeit:** Ergebnis nebenläufiger Ausführung soll serieller Ausführung entsprechen.

- **Isolation Levels:** Schwächere Stufen für mehr Nebenläufigkeit möglich.

Recovery (Atomicity und Durability)

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

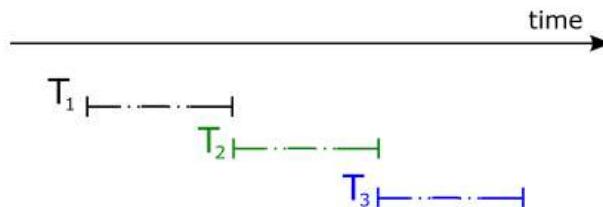
Schedules und Serialisierbarkeit

Nebenläufigkeit (Parallelität)

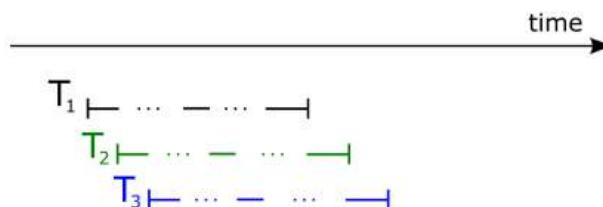
Das „I“ in ACID.

Ausführung der drei Transaktionen T_1 , T_2 , und T_3

(a) im Einzelbetrieb



(b) im (nebenläufigen) Mehrbenutzerbetrieb mit verzahnter Ausführung



Probleme bei nebenläufiger Ausführung

1. **Lost Updates** (verlorengegangene Änderung durch Überschreiben)

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

2. **Dirty Read** (Abhängigkeit von nicht freigegebenen Änderungen)

Steps	T_1	T_2
1.	read(A, a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A, a_1)	
4.		read(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A, a_2)
7.	read(B, b_1)	
8.	...	
9.	abort	

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

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

4. Phantomproblem (Abhängigkeit von neuen/gelöschten Tupeln)

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

Nebenläufigkeit und Korrektheit

Zentrales System mit gleichzeitigem Zugriff durch mehrere Benutzer

- Datenbank besteht aus zwei Einträgen: X und Y
- Einziges Kriterium für Korrektheit: $X = Y$
- Folgende Transaktionen:

$$\begin{array}{ll} T_1 & X \leftarrow X + 1 \\ & Y \leftarrow Y + 1 \end{array} \quad \begin{array}{ll} T_2 & X \leftarrow 2 * X \\ & Y \leftarrow 2 * Y \end{array}$$

- Initial: $X=10$ und $Y=10$.
- T_1 gefolgt von $T_2 \Rightarrow X = 22$ and $Y = 22$
- T_2 gefolgt von $T_1 \Rightarrow X = 21$ and $Y = 21$

Beispiel von [hier](#) bis [hier](#).

Formale Definition eines Schedules

[DBS-7_transaktionen, p.19](#), [DBS-7_transaktionen, p.19](#)

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

Arten von Schedules

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

Gültiger Schedule

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

Beispiele

schedule S_0		schedule $S_{0'}$		schedule S_1	
T_1	T_2	T_1	T_2	T_1	T_2
read(X, x)		read(X, x)		read(X, x)	
$x \leftarrow 2x$		$x \leftarrow 2x$		$x \leftarrow x+1$	
write(X, x)		write(X, x)		write(X, x)	
read(Y, y)	read(X, x)	read(Y, y)		read(Y, y)	
$y \leftarrow 2y$	$x \leftarrow x+1$	$y \leftarrow 2y$		$y \leftarrow y+1$	
write(Y, y)	write(X, x)	write(Y, y)		write(Y, y)	
read(X, x)				read(X, x)	
$x \leftarrow x+1$				$x \leftarrow 2x$	
write(X, x)				write(X, x)	
read(Y, y)		read(Y, y)		read(Y, y)	
$y \leftarrow y+1$		$y \leftarrow y+1$		$y \leftarrow y+1$	
write(Y, y)		write(Y, y)		write(Y, y)	

• X = 21, Y = 21
• Serieller Schedule
• X = 21, Y = 21
• Nebenläufiger Schedule
• X = 22, Y = 21
• Ungültiger Schedule

Korrektheit

Definition D1:

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

Definition D2 (Ergebnisäquivalenz):

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

Beispiel

schedule S_2	
T_3	T_4
read(X, x)	
$x \leftarrow x+1$	
write(X, x)	read(X, x)
	$x \leftarrow 2x$
	write(X, x)
read(Y, y)	read(Y, y)
$y \leftarrow y+1$	$y \leftarrow 2y$
write(Y, y)	
	write(Y, y)

Initial: $X = 10$ und $Y = 10$
 $\Rightarrow X = 20$ und $Y = 20$

- S_2 ist nicht ergebnisäquivalent zu einer seriellen Ausführung von T_3 , T_4
- Aber der finale Datenbankzustand ist konsistent – obwohl es einige Lost Updates gibt.

Die bessere Wahl ist Definition D2:

Die Ausführungsreihenfolge ist **korrekt**, wenn sie zu einer **seriellen Ausführung ergebnisäquivalent** ist.

Gegeben ist eine Menge von n nebenläufigen Transaktionen. Wie können wir effizient auf Korrektheit prüfen?

Im Folgenden gehen wir von vereinfachenden Annahmen aus

- **Nur Reads und Writes** werden verwendet, um die Korrektheit festzustellen.
- Diese Annahme ist strenger als Definition D2, da noch weniger Schedules als korrekt gelten.

Konflikt Serialisierbarkeit

Mögliche Konflikte zwischen Transaktionen

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

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

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

Konflikt

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

Konflikt

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

Konflikt

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

Kein Konflikt

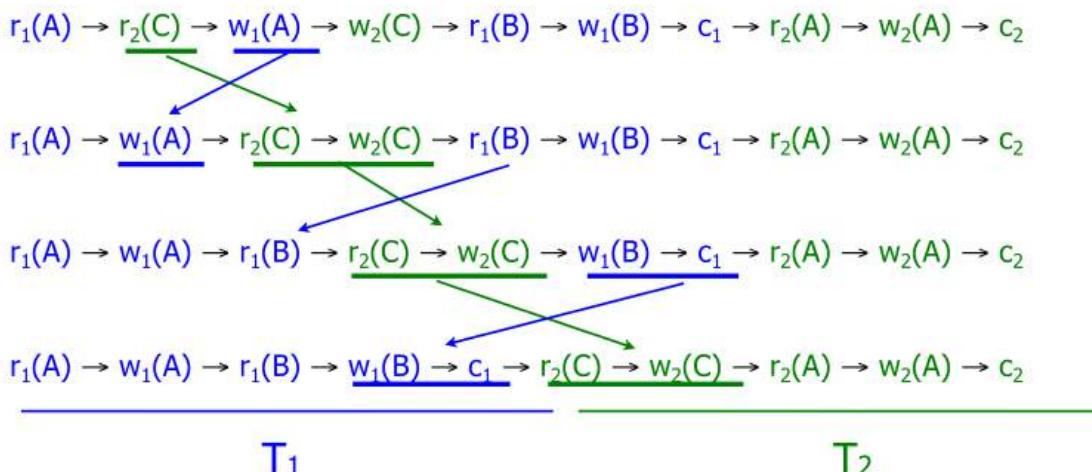
Definition (D4) Conflict Serializability

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

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

Beispiel Konfliktäquivalenz von 2 Schedules

Die Transformation zeigt, dass der initiale Schedule konfliktäquivalent zu einem seriellen Schedule ist. Daher ist dieser auch conflict serializable.



c ist die Abkürzung für commit, r (read), w (write)

Weitere Beispiele:

schedule S_A	
T_1	T_2
read(Y, y)	read(X, x) write(X, x)
write(Y, y)	

Conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	read(X, x) write(X, x)
write(X, x)	

Nicht conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	read(X, x)
write(X, x)	

Conflict serializable

schedule S_D	
T_7	T_8
read(X, x)	write(X, x)
write(X, x)	

Nicht conflict serializable

Conflict Graph (Precedence Graphs)

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

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

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

Beispiel für einen Conflict Graph des Schedules S_1



Serialisierbarkeit feststellen

Gegeben:

- Ein Schedule S
- Ein Conflict Graph

Wie feststellen, ob S conflict serializable ist?

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

Intuition:

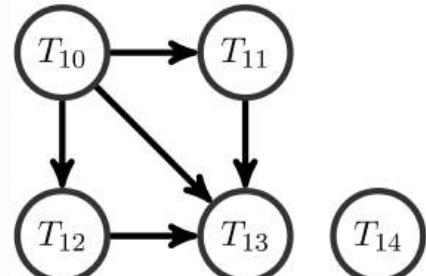
- Ein **Konflikt** zwischen zwei Transaktionen erzwingt eine bestimmte Ausführungsreihenfolge.
- Die Conflict Serializability entspricht der Existenz einer **topologischen Sortierung** des Conflict Graphen.

Warum Conflict Serializability?

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

Beispiel für Conflict Graph

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



Wir haben 5 Knoten weil wir 5 einzelne Transaktionen haben.

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

So macht man das dann für alle anderen Kanten auch.

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

Was sagt uns der Graph jetzt?

- er hat keinen Zyklus
- --> Er ist Konflikt serialisierbar

verschiedene Ergebnisse

es gibt jetzt verschiedene Reihenfolgen die funktionieren:

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

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

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

Beziehungen zwischen Schedules

Alle Schedules

Schedules die DB in einem konsistenten Zustand belassen (D1)

Schedules äquivalent zu einem seriellen Schedule (D2)

Sichtenserialisierbare Schedules (D3)

Conflict Serializable Schedules (D4)

Serielle Schedules

Recoverable Schedules und Cascadeless Schedules

Transaktionen können fehlschlagen

- Abort vom User aus
- Rollback
- Irgendwelche Fehler

Recoverable Schedules

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

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

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

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

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

recoverable
recoverable

Cascading Rollbacks (Kaskadierendes Rücksetzen)

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

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

Ist recoverable weil nicht committed wurde

Cascadeless Schedules

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

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

Dieser Schedule ist auch recoverable

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

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

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

Vorteil:

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

Jeder cascadeless Schedule ist auch recoverable.

Cascading Rollbacks:

- Können schnell zeitaufwändig werden.

Schlussfolgerung:

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

Zusammenfassung: Transaktionen und Schedules

- **Transaktionen:** Erhalten den konsistenten Zustand der Datenbank.
- **Serielle Ausführung:** Eine serielle Ausführung einer Menge von Transaktionen erhält einen konsistenten Zustand.
- **Nebenläufige Ausführung:** Die einzelnen Ausführungsschritte mehrerer Transaktionen können verzahnt sein.
- **Serialisierbarkeit:** Ein nebenläufiger Schedule ist serialisierbar, wenn er äquivalent zu einem seriellen Schedule ist.

Conflict Serializability

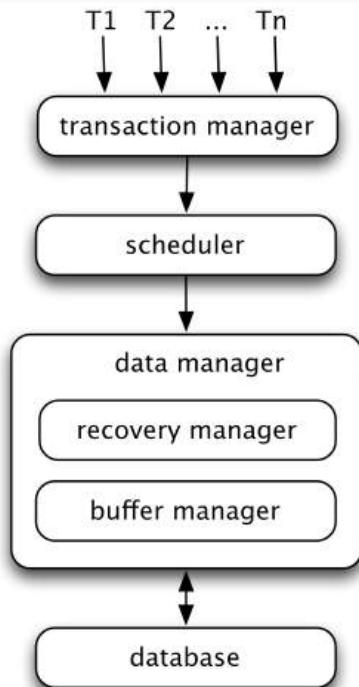
- **Bevorzugte Methode:** Aufgrund praktikabler Implementierungsmöglichkeiten.
- **Conflict Graphs:** Werden zur Bestimmung der Conflict Serializability verwendet. Ein Schedule ist conflict serializable, wenn sein Conflict Graph azyklisch ist.

Recoverability und Cascadeless Schedules

- Schedules müssen **recoverable** sein.
 - Ein Schedule ist recoverable, wenn für jedes Transaktionspaar T_i und T_j , bei dem T_j Daten liest, die von T_i geschrieben wurden, T_i vor T_j committet wird.
 - Schedules sollten **cascadeless** sein.
 - Jeder cascadeless Schedule ist auch recoverable.
 - Ein Schedule ist cascadeless, wenn für jedes Transaktionspaar T_i und T_j , bei dem T_j Daten liest, die von T_i geschrieben wurden, T_i vor dem Lesezugriff von T_j bereits committed ist.
 - Cascading Rollbacks können durch cascadeless Schedules verhindert werden.: Transaktionen und Schedules
-

Datenbank Scheduler

Der Datenbank-Scheduler



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

Synchronisationsverfahren

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

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

Sperrbasierte Synchronisation

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

Privilegien bei Locks

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

Verträglichkeitsmatrix (Kompatibilitätsmatrix)

	NL	S	X
S	OK	OK	-
X	OK	-	-

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

Beispiel

Ein Beispiel

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

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

Resultat bei serieller Ausführung:

T_{16} zeigt 300

Bei diesem Schedule:

T_{16} zeigt 250

Ursache des Problems:

Der Exclusive Lock auf B wurde zu früh freigegeben.

Initial: A = 100 und B = 200

Probleme bei zu früher Eingabe

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

Probleme bei später Sperrfreigabe

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

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

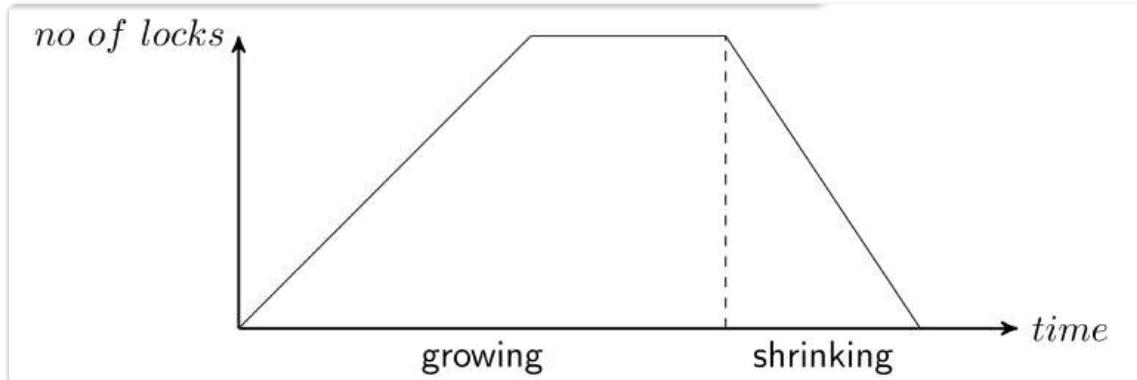
- Späte Sperrfreigaben verhindern Non-Serializable Schedules, aber sie erhöhen die Chancen von **Deadlocks**.
- Lernen Sie damit zu leben!

Das Zwei-Phasen-Sperrprotokoll (2PL)

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

2. 2. Phase (Freigabephase, shrinking phase):

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



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

Beispiele: 2PL: Ja oder nein?

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

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

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

Eigenschaften des Zwei-Phasen-Sperrprotokolls

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

Cascading Rollbacks

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

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

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

Striktes und rigoroses Zwei-Phasen-Sperrprotokoll

Striktes 2PL

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

Rigoroses 2PL

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

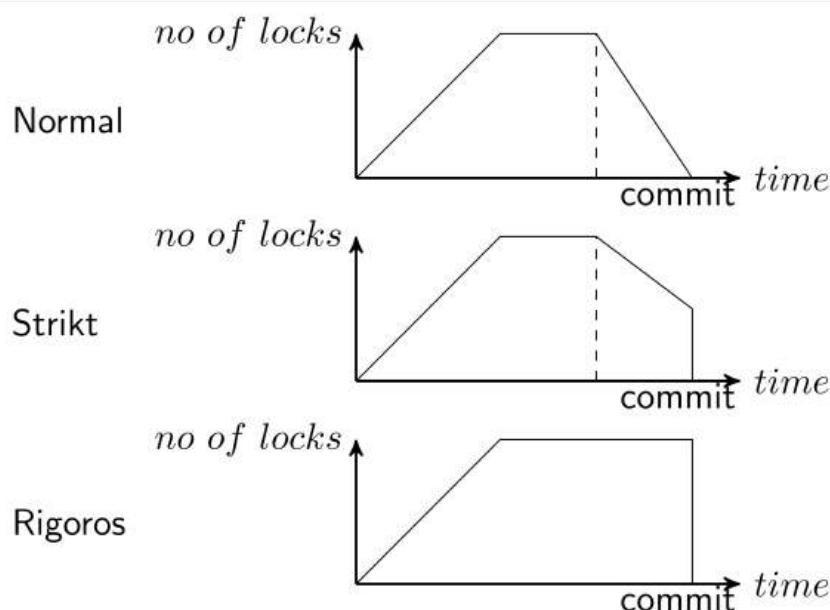
Gemeinsamer Vorteil

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

Nachteil

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

Übersicht 2PL Protokolle



Vorteil von rigorosem 2PL gegenüber striktem 2PL

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

Konvertierung von Sperren

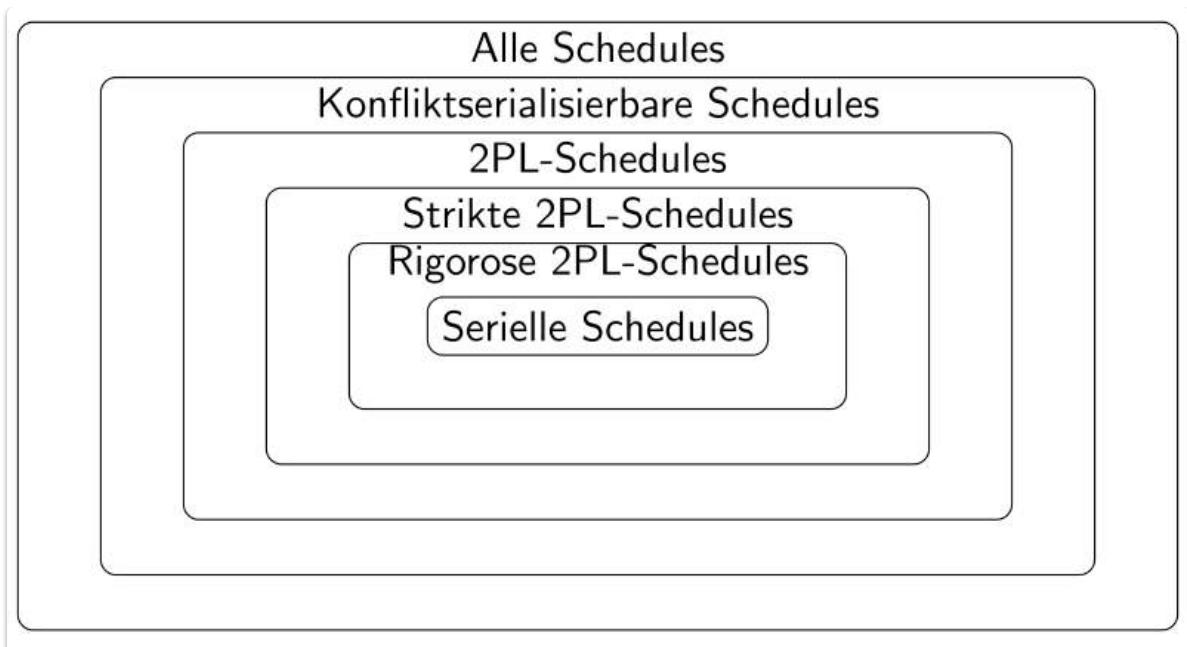
- **Ziel:** 2PL anwenden, aber einen höheren Grad an Nebenläufigkeit erlauben.
- **Erste Phase (Growing Phase):**
 - S-Lock erhalten (`lock_s`)

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

Beispiele zu den Arten

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

Übersicht über 2PL-Schedules



Erkennung von Deadlocks

Deadlocks

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

T_1	T_2	
lock_X(A) read(A) write(A) lock_X(B) ...	lock_S(B) read(B) lock_S(A) ...	T_1 muss warten auf T_2 T_2 muss warten auf T_1 \Rightarrow Deadlock

- **Erläuterung des Deadlocks im Beispiel:**

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

- **Lösungen für Deadlocks:**

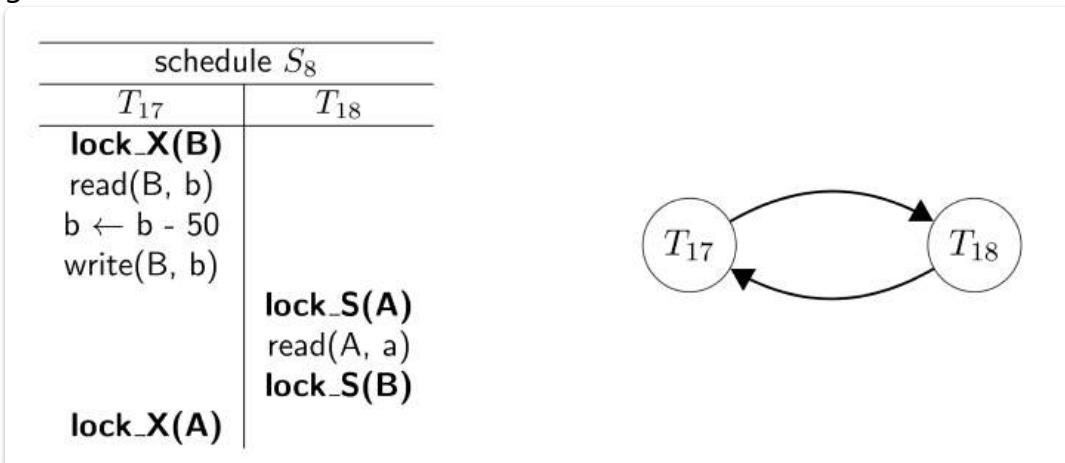
- **Erkennung von Deadlocks und anschließend Recovery:** Das System erkennt, wenn ein Deadlock aufgetreten ist, und ergreift Maßnahmen, um ihn aufzulösen (z.B.

Abbruch einer der beteiligten Transaktionen).

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

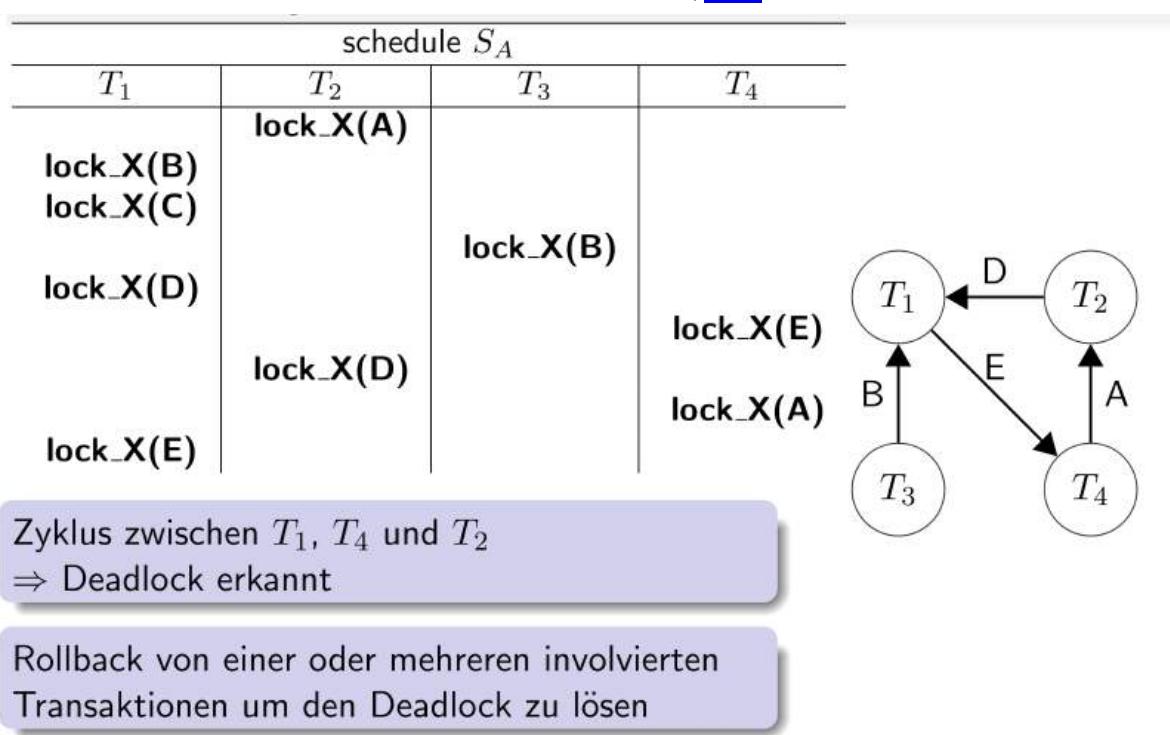
Erkennung von Deadlocks

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



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

Beispiel



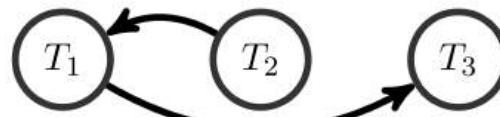
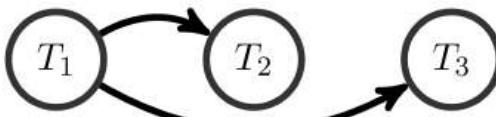
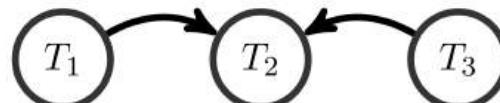
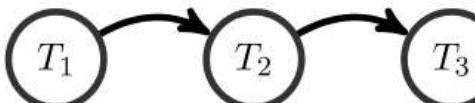
Rollback Kandidaten

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

Wertgraph

Welcher Wartegraph passt zu diesem Schedule?

schedule S_A		
T_1	T_2	T_3
	lock_X(A)	
lock_X(A) lock_X(B)		lock_X(B) lock_X(C)



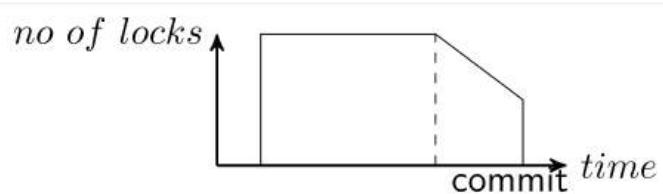
Heir passt nur Option 3 (links unten), weil T_3 wartet auf T_2 und muss auch auf T_3 warten. Da es keinen Zyklus gibt, sieht das gut aus und wir haben keine Probleme / kein Deadlock

Vermeidung von Deadlocks

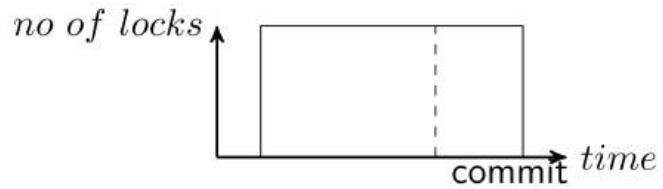
Konservatives 2PL-Protokoll

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

konservatives striktes 2PL



konservatives rigoroses 2PL



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

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

Zusammenfassung Mehrbenutzersynchronisation

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

Recovery

Fehlerklassifikation

- **Aspekte:**
 - **Atomarität:**
 - Transaktionen können einen Abort ausführen (Rollback). Entweder werden alle Operationen einer Transaktion erfolgreich abgeschlossen (Commit), oder keine von ihnen hat einen dauerhaften Effekt auf die Datenbank (Abort).
 - **Dauerhaftigkeit:**
 - Was muss passieren, wenn das DBMS abstürzt? Nach einem Commit einer Transaktion müssen die Änderungen dauerhaft in der Datenbank gespeichert bleiben, auch bei Systemfehlern.
- **Garantie des DBMS:** Das DBMS garantiert, dass eine Transaktion
 - entweder fertig wird und ein permanentes Resultat liefert (committed)
 - oder keinen Effekt auf die Datenbank hat (aborted).
- **Rolle der Recovery-Komponente:** Die Rolle der Recovery-Komponente ist es, die Atomarität und Dauerhaftigkeit von Transaktionen trotz etwaiger Systemfehler zu garantieren.

Wie kann Dauerhaftigkeit garantiert werden?

Noch nicht auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **noch nicht** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem Blackout (Systemausfall) in diesem Moment?
- Welche Daten sind tatsächlich in der Datenbank auf der Festplatte gesichert? Die Änderungen der gerade committeten Transaktion **könnten verloren sein**, wenn sie noch nicht persistent gespeichert wurden.

Teilweise auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **teilweise** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem Blackout (Systemausfall) in diesem Moment?
- Welche Daten sind in der Datenbank? Es ist unklar, ob alle Änderungen der committeten Transaktion persistent gespeichert wurden, da der Schreibvorgang möglicherweise unterbrochen wurde. Dies kann zu einer **inkonsistenten Datenbank** führen, bei der einige, aber nicht alle Änderungen der Transaktion übernommen wurden.

Vollständig auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **vollständig** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem **Hardware-Fehler**, z.B. dem **Verlust einer Festplatte**?

- Welche Daten sind in der Datenbank? Obwohl die Daten der Transaktion auf *einer* Festplatte gespeichert wurden, sind sie bei einem Ausfall dieser spezifischen Festplatte verloren.
- **Schlussfolgerung:** Das alleinige Schreiben auf eine Festplatte garantiert keine vollständige Dauerhaftigkeit gegen Hardware-Fehler. Es sind zusätzliche Mechanismen zur Datensicherung und Redundanz erforderlich.

Vollständig auf mehreren Festplatten

- **Szenario:**
 - Eine Transaktion verändert Daten im Arbeitsspeicher.
 - Die Daten wurden **vollständig auf mehrere Festplatten** geschrieben (Redundanz).
 - Ein Commit wird durchgeführt.
 - Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.
- **Problem:**
 - Was passiert, wenn es ein schwerwiegendes Ereignis wie Feuer, Flut, Erdbeben oder ähnliches gibt, das den physischen Standort der Datenbank betrifft und **alle Festplatten zerstört oder unzugänglich macht?**
 - Welche Daten sind in der Datenbank? In diesem Fall wären die Daten, obwohl redundant auf mehreren lokalen Festplatten gespeichert, **verloren**, da alle Kopien gleichzeitig zerstört wurden.
- **Schlussfolgerung:** Um gegen solche katastrophalen Ereignisse gewappnet zu sein, sind **externe Backups** und **Disaster-Recovery-Pläne** unerlässlich, bei denen Daten an einem geografisch entfernten Ort gesichert werden.

Alle Festplatten kaputt

- **Szenario:**
 - Eine Transaktion verändert Daten im Arbeitsspeicher.
 - Die Daten wurden vollständig auf mehrere Festplatten geschrieben, und die Festplatten wurden auf mehreren **geografisch verschiedenen Computerzentren** verteilt.
 - Ein Commit wird durchgeführt.
 - Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.
- **Problem:**
 - Was passiert, wenn es ein extrem unwahrscheinliches, aber theoretisch mögliches Ereignis wie Feuer, Flut, Erdbeben oder ähnliches **gleichzeitig bei allen Computerzentren** gibt, sodass alle Standorte und damit alle Kopien der Daten verloren gehen?
 - Welche Daten sind in der Datenbank? In diesem extremen Szenario wären **alle Daten verloren**.

- **Schlussfolgerung:** Obwohl die Wahrscheinlichkeit eines solchen globalen Datenverlusts durch geografische Verteilung extrem gering ist, zeigt es die fundamentale Abhängigkeit von der physischen Existenz der Datenträger. Um absolute Sicherheit zu gewährleisten, bräuchte es hypothetisch widerstandsfähige Speicherorte außerhalb unseres Planeten oder ähnliche futuristische Lösungen. In der praktischen Realität stellt die geografische Verteilung über mehrere Rechenzentren jedoch einen extrem hohen Grad an Ausfallsicherheit dar.

Dauerhaftigkeit (Durability)

- **Relativität:** Dauerhaftigkeit ist **relativ** und hängt von der Anzahl der Datenkopien und deren geografischer Verteilung ab. Je mehr Kopien an unterschiedlichen Orten existieren, desto höher ist die Wahrscheinlichkeit, dass Daten auch bei Ausfällen einzelner Komponenten oder Standorte erhalten bleiben.
- **Garantien:** Garantien für Dauerhaftigkeit sind nur möglich, wenn:
 - wir **zuerst** die Kopien der Daten persistent aktualisieren (z.B. auf Festplatten schreiben)
 - und **erst dann** den Benutzer darüber informieren, dass der Commit der Transaktion erfolgreich war. Dieses Prinzip wird oft durch das **Write-Ahead Logging (WAL)** sichergestellt.
- **Annahme:** Wir nehmen deshalb an, dass die WAL-Regel erfüllt ist. Das bedeutet, dass Änderungen an der Datenbank zuerst in einem Logfile protokolliert werden, bevor sie in die eigentliche Datenbank geschrieben werden. Dieses Log ermöglicht es dem System, nach einem Absturz den Zustand der Datenbank wiederherzustellen.
- **Variationen der WAL-Regel (Strategien zur Erhöhung der Dauerhaftigkeit):**
 - **Log-Based Recovery:** Die Wiederherstellung der Datenbank nach einem Fehler basiert auf der Analyse des Transaktionslogs.
 - **Volle Redundanz:** Spiegeln aller Daten auf mehreren Computern (Festplatten, Rechenzentren), die alle die gleichen Operationen ausführen. Dies bietet hohe Fehlertoleranz und Datenverfügbarkeit.

Fehlerklassifikation

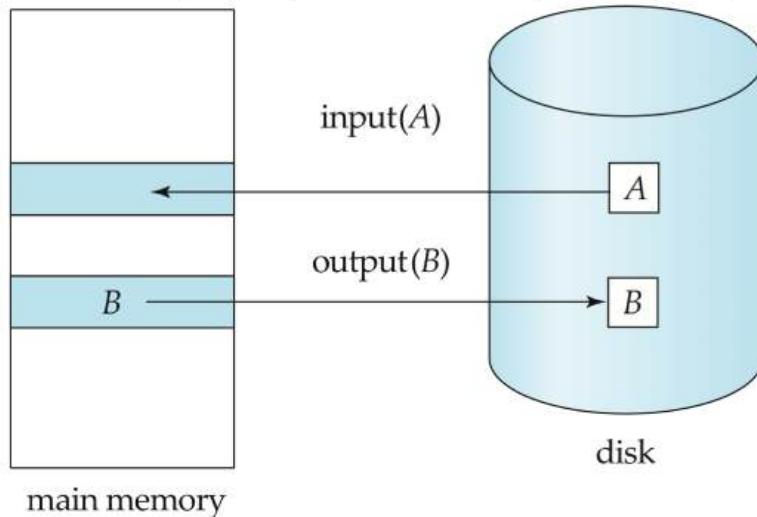
- **Transaktionsfehler (Fehler einer Transaktion, die noch nicht committed wurde):**
 - **Maßnahme:** Rückgängigmachen der Änderungen (Rollback) der betroffenen Transaktion, um die Atomarität zu gewährleisten.
- **Systemabsturz (Fehler mit Hauptspeicherverlust):**
 - **Anforderungen an die Recovery:**
 - Änderungen von Transaktionen, die **committed** wurden, müssen erhalten bleiben (Dauerhaftigkeit).
 - Änderungen von Transaktionen, die **nicht committed** wurden, müssen rückgängig gemacht werden (Atomarität).
- **Festplattenfehler:**

- **Maßnahme:** Recovery basiert in der Regel auf Archiven oder Datenbank-Dumps (regelmäßige Sicherungskopien der Datenbank), die auf anderen Speichermedien gespeichert sind. Zusätzlich können Transaktionslogs verwendet werden, um Änderungen seit dem letzten Backup wiederherzustellen (Log-Based Recovery).

Datenspeicher

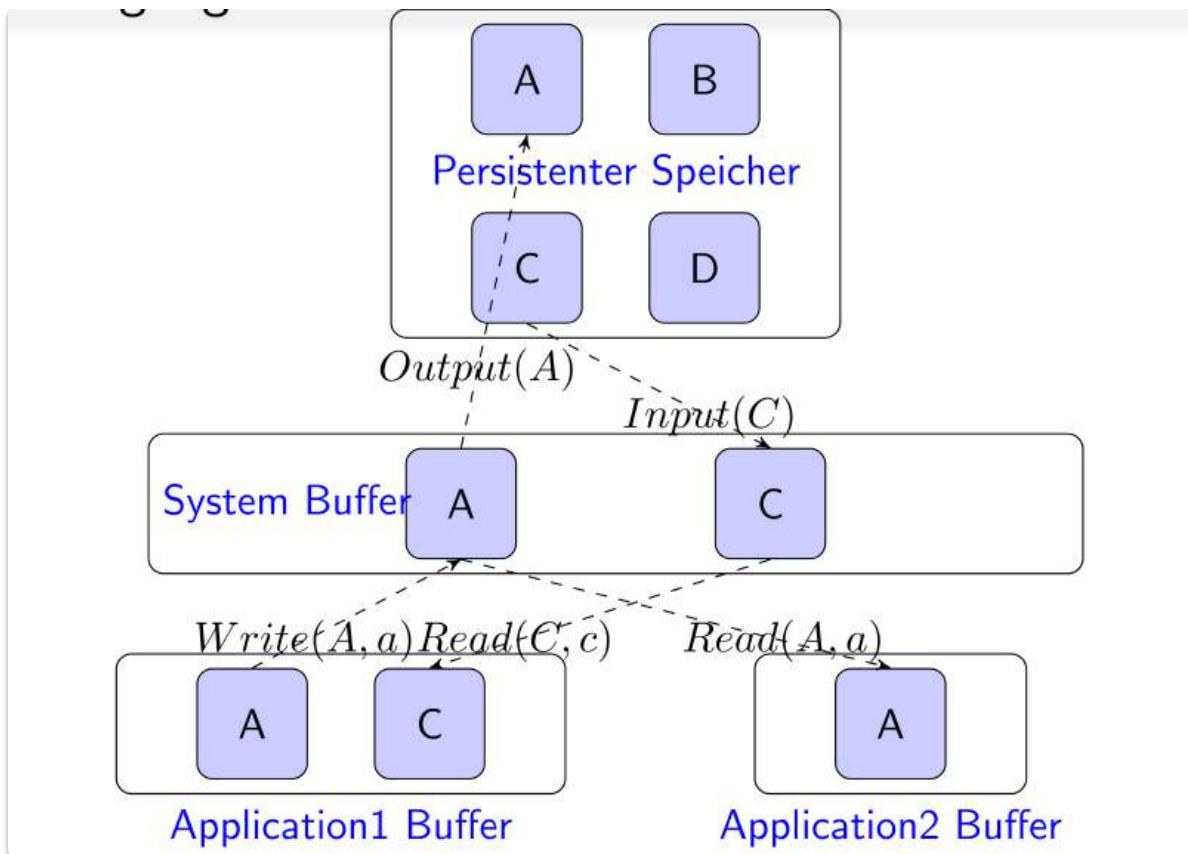
Zweistufige Speicherhierarchie

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



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

Bewegung der Daten



Speicheroperationen

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

Log-Einträge

Die WAL-Regel für Log-Based Recovery

WAL (Write Ahead Logging)

- **Regel 1 (Commit-Regel):** Bevor eine Transaktion in den Commit-Zustand wechselt, müssen **alle zugehörigen Log-Einträge** auf einen stabilen Speicher (z.B. Festplatte) geschrieben worden sein, **inklusive dem Commit-Log-Eintrag** selbst. Dies stellt sicher, dass ein Commit auch nach einem Systemabsturz nachvollzogen werden kann.
- **Regel 2 (Write-Regel):** Bevor eine modifizierte Seite (oder ein Block) im Arbeitsspeicher in die Datenbank (nichtflüchtiger Speicher) geschrieben werden kann, müssen **alle zugehörigen Log-Einträge** für diese Änderung bereits auf einem stabilen Speicher geschrieben worden sein. Dies gewährleistet, dass die Log-Informationen zur Wiederherstellung verfügbar sind, falls ein Absturz während des Schreibens der Daten in die Datenbank erfolgt.

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

Logging

Im normalen Betrieb

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

Struktur eines Log-Eintrags (Log Record)

- **Update-Eintrag:**
 - Format: $[TID, DID, old, new]$
 - **TID:** ID der Transaktion, die die Änderung verursacht hat.

- **DID:** ID des Datenobjekts. Dies kann die genaue Speicheradresse auf der Festplatte angeben (z.B. Seite, Block, Offset).
 - **old:** Wert des Datenobjekts vor der Änderung.
 - **new:** Wert des Datenobjekts nach der Änderung.
- **Zusätzliche Steuerungseinträge:**
- **Start-Eintrag:** [TID start] - Markiert den Beginn der Transaktion mit der ID TID.
 - **Commit-Eintrag:** [TID commit] - Markiert das erfolgreiche Ende (Commit) der Transaktion mit der ID TID.
 - **Abort-Eintrag:** [TID abort] - Markiert das abgebrochene Ende (Abort) der Transaktion mit der ID TID.

Beispiele

Generelles Log-Einträge-Beispiel

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

Finde die Fehler

[T1 commit]
[T1, B, 300, 400]
[T1 start]

Commit vor Start

[T1 start]
[T1, B, 300, 400]
[T1 commit]
[T1, C, 40, 540]

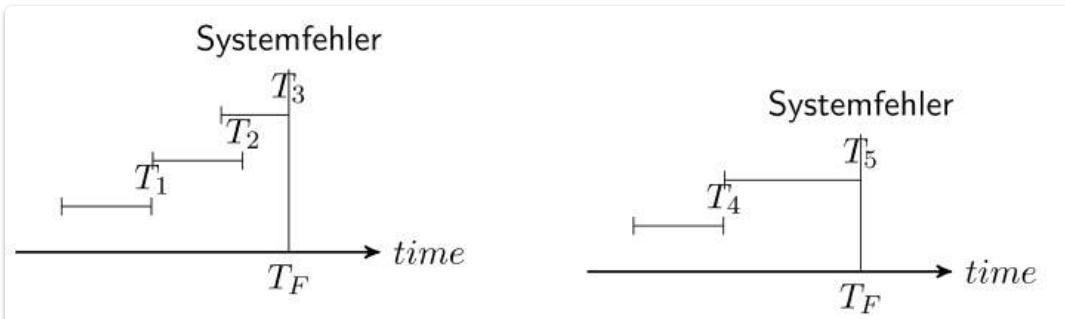
Es darf keine Log-Einträge für T1 nach dem Commit geben.

[T1 start]	[T1 start]
[T1, C, 40, 10]	[T1, C, 40, 10]
[T1 start]	[T1, B, 300, 400]
[T1, B, 300, 400]	[T1 commit]
[T1 commit]	Korrekt!

Mehrere Start-Einträge für T1

Log-Based Recovery

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



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

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

Recovery Algorithmus

- **Grundlegende Schritte:**

- **Reproduzieren (redo) der Resultate von Transaktionen, die committed wurden:**
Anhand des Logs werden die Änderungen der erfolgreich abgeschlossenen Transaktionen erneut angewendet, um sicherzustellen, dass ihre Effekte in der Datenbank erhalten bleiben.
- **Rückgängigmachen (undo) von Transaktionen, die noch nicht committed wurden:**
Anhand des Logs werden die Änderungen unvollständiger Transaktionen entfernt, um die Atomarität zu gewährleisten. Die Datenbank wird in einen Zustand zurückversetzt, als ob diese Transaktionen nie begonnen hätten.

- **Anmerkungen:**

- In einem Multitasking-System müssen eventuell mehr als eine Transaktion rückgängig gemacht werden, falls mehrere Transaktionen zum Zeitpunkt des Fehlers aktiv waren.
- **Idempotenz der Recovery:** Wenn ein Systemabsturz während der Recovery-Phase auftritt, muss auch ein erneuter Recovery-Anlauf korrekte und konsistente Resultate liefern. Das Recovery-Verfahren muss idempotent sein, d.h., es darf keine schädlichen oder inkonsistenten Nebeneffekte haben, wenn es mehrmals ausgeführt wird.

Log-Based Recovery

Datenbank	Log-Einträge
A 100	[T1 start]
B 300	[T1, B, 300, 400]
C 5	[T1, C, 5, 10]
D 60	[T2 start]
E 80	[T2, E, 80, 480]
	[T1, A, 100, 560]
	[T1 commit]
	[T2, A, 560, 570]
	[T2, D, 60, 530]

Wie würden Sie die Log-Informationen verwenden (systematisch), um die Datenbank nach einem Absturz wiederherzustellen?

Drei Recovery-Phasen

Phase 1:

- **Redo (Vollständige Wiederholung der Historie)**
 - Alle Log-Einträge werden Schritt für Schritt in chronologischer Reihenfolge durchgegangen (vorwärts).
 - Alle im Log verzeichneten Änderungen werden in derselben Reihenfolge auf die Datenbank angewendet (Redo).

- **Bestimmen der "Undo"-Transaktionen:**

- Für jeden `[T_i start]` Eintrag im Log wird die Transaktion T_i zur "Undo List" hinzugefügt. Dies sind potenziell unvollständige Transaktionen.
- Wenn ein `[T_i commit]` oder `[T_i abort]` Eintrag für eine Transaktion T_i gefunden wird, wird T_i von der "Undo List" entfernt, da diese Transaktion entweder erfolgreich abgeschlossen oder explizit abgebrochen wurde. Die Transaktionen, die am Ende dieser Phase noch in der "Undo List" sind, waren zum Zeitpunkt des Systemfehlers aktiv und nicht committed.

Phase 2:

- **Undo (Rollback aller Transaktionen in der "Undo List")**

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

Compensation-Log-Einträge

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

werden. In diesem Fall werden sie erzeugt, um die bereits erfolgten Änderungen der Transaktion zu neutralisieren.

Beispiel

Ein langes Beispiel dafür gibt es von [DBS-7_transaktionen part 2, p.90](#) bis [hier](#) in den Slides.

ARIES

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

Zusammenfassung Recovery

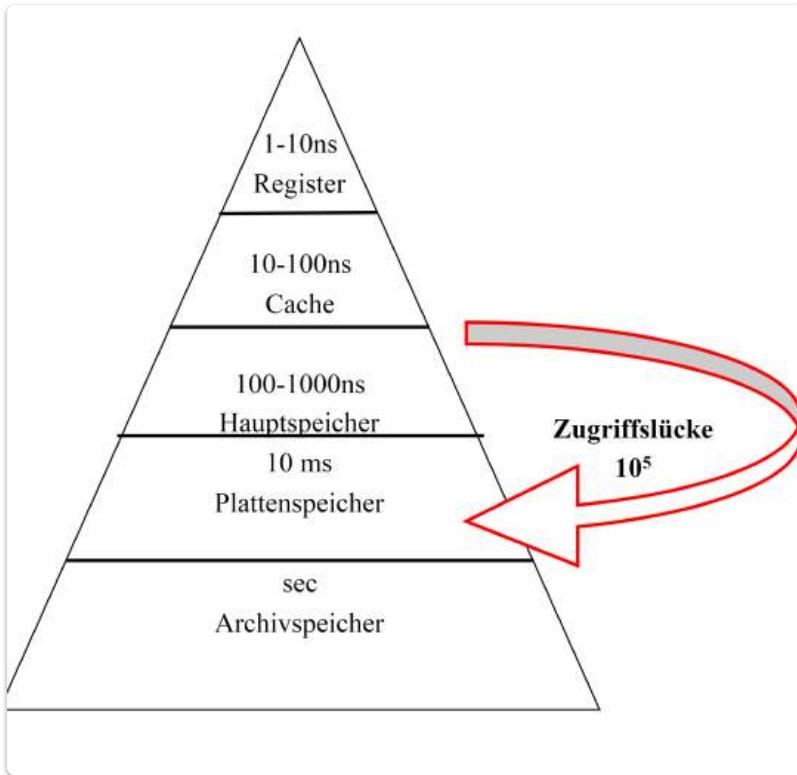
- **Ziel:** Sicherstellen von Atomarität und Dauerhaftigkeit trotz Systemfehlern und Abstürzen.
- **Dauerhaftigkeit ist relativ** und hängt von der Redundanz und der geografischen Verteilung der Daten ab.
- **WAL-Regel (Write Ahead Logging):** Log-Einträge müssen persistent gespeichert sein, bevor die zugehörigen Datenbankänderungen geschrieben oder ein Commit durchgeführt wird.
- **Log-Based Recovery:** Ein Verfahren zur Wiederherstellung der Datenbank mithilfe eines Transaktionslogs.
 - Alle Änderungen müssen in eine Log-Datei geschrieben werden.
 - Eine Transaktion führt ein Commit genau dann durch, wenn der Commit-Eintrag im Log geschrieben wurde.

6. Physischer Datenbankentwurf

Slides: [DBS-8_Physischer Datenbankentwurf.pdf](#)

Dateiorganisation - Speicherorganisation

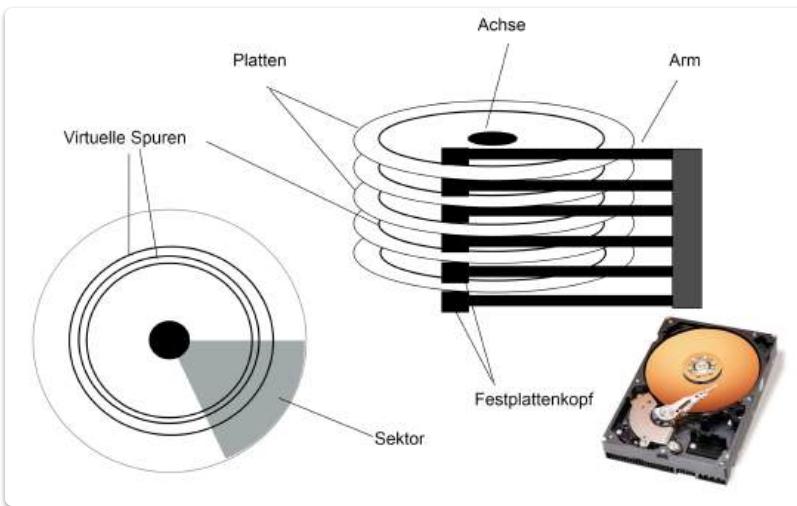
Speicherebenen



- **Primärspeicher (Volatile Storage)**
 - z.B. Cache, Hauptspeicher, Register
 - Verliert den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeiten:
 - Register: 1-10 ns
 - Cache: 10-100 ns
 - Hauptspeicher: 100-1000 ns
- **Sekundärspeicher (Non-Volatile Storage)**
 - z.B. Festplatte
 - Behält den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeit: ca. 10 ms (deutlich langsamer als Primärspeicher)
- **Tertiärspeicher (Non-Volatile Storage)**
 - z.B. Magnetbänder (Archivspeicher)
 - Behält den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeit: sec (noch langsamer als Sekundärspeicher)

- Je höher das Level (im Mehrebenenmodell), desto schneller der Zugriff.
- **Zugriffslücke:** Die Diskrepanz in der Zugriffszeit zwischen Hauptspeicher und Plattspeicher ist sehr groß (10^5).

Magnetische Festplatte



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

Zugriffsoptimierung für Festplatten

Für jeden Speicherzugriff auf die Festplatte fallen folgende Zeiten an:

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

Optimierung des Festplattenzugriffs

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

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

Funktionale Anforderungen an Datenbanksysteme

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

Performanzanforderungen an Datenbanksysteme

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

Dateiorganisation - Dateien und Tupel (Records)

Dateiorganisation

Speichern von Datenbanken auf Festplatten

- Eine Datenbank wird als Menge von **Dateien (Files)** gespeichert.
- Jede Datei enthält eine Menge von **Tupeln (Records)**.
- Ein Tupel/Record enthält eine bestimmte Anzahl von **Feldern (Fields)**.

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

Größe eines Records

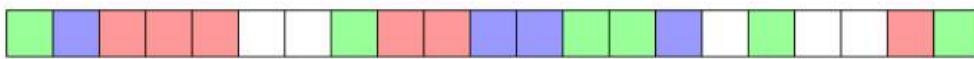
- **Feste Größe (Fixed Size):** Alle Records in einer Datei haben die gleiche Länge. Dies erleichtert die Berechnung der Speicheradresse eines bestimmten Records.
- **Variable Größe (Variable Size):** Records in einer Datei können unterschiedliche Längen haben. Dies erfordert zusätzliche Informationen,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

Dateien auf Massenspeichern (normalerweise Festplatten)

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

Beispiel

Blöcke in einem Dateisystem sind nicht unbedingt zusammenhängend (contiguous).



Geschwindigkeit

- Geschwindigkeit des Lesezugriffs auf einen Block (ein I/O-Zugriff):
 - ≈10 msec für einen nicht zusammenhängenden Block
 - ≈1 msec für einen zusammenhängenden Block
- DBMS/OS kann Blöcke reorganisieren, um den Zusammenhang wiederherzustellen.

Feste Größe (Fixed Size Records)

- Alle Tupel haben die **gleiche Länge (Größe)**, auch wenn sie nicht den gesamten reservierten Speicherplatz benötigen.

Löschen eines Tupels i

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

Verschieben von Tupeln:

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

Markieren der Lücken:

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

Variable Größe (Variable Size Records)

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

Alternativen für variable Größe

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

des Blocks).

Organisation von Tupeln in Dateien

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

Heap Files

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

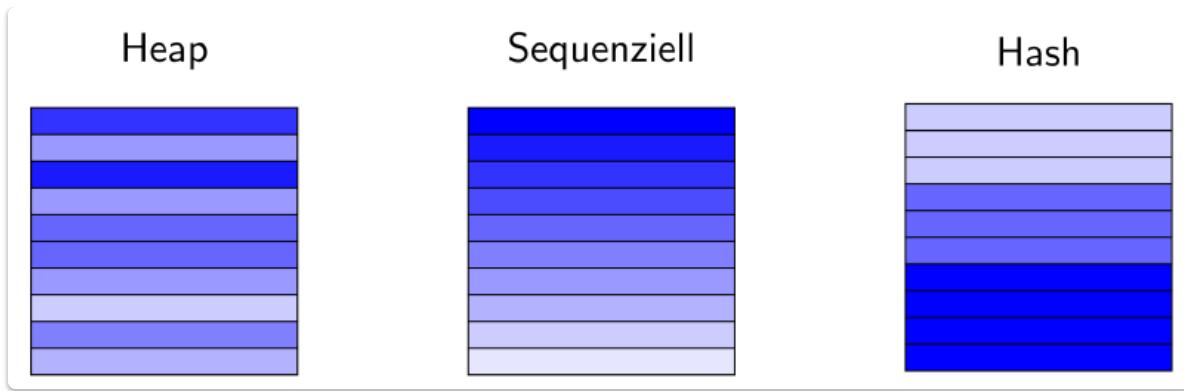
Sequentielle Dateiorganisation

- Tupel werden **sequenziell in Reihenfolge des Search Keys** (z.B. ein bestimmtes Attribut) abgelegt.
- Die physische Reihenfolge der Datensätze entspricht der logischen Ordnung basierend auf dem Suchschlüssel.
- Dies ermöglicht effizientere Bereichsabfragen basierend auf dem Suchschlüssel.
- Einfügungen und Löschungen können aufwendig sein, da möglicherweise viele Datensätze verschoben werden müssen, um die Reihenfolge beizubehalten.

Hash Files

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

Overview



Heap

- **Keine bestimmte Reihenfolge** innerhalb der Tabelle. Die Tupel sind in keiner spezifischen Ordnung gespeichert.
- **Suche:** Alle Tupel nacheinander durchsuchen (**Linear Scan**), bis das gesuchte Tupel gefunden wird. Dies kann ineffizient sein, insbesondere bei großen Tabellen.
- **Einfügen:** Finde einen freien Slot (entweder am Ende der Datei oder eine zuvor freigegebene Lücke) und füge das neue Tupel dort ein. Das Einfügen ist in der Regel effizient.

3	Larsen	E117
6	Rose	E167
8	Lazy	E176
1	Aaen	E111
4	Ravn	E161
7	Torp	E171
2	Dolog	E116
5	Srba	E166

Sequenziell

- Tabelle ist anhand eines **Search-Keys (Suchschlüssels)** sortiert, z.B. ID-Attribut. Die physische Reihenfolge der Tupel entspricht der Sortierreihenfolge des Suchschlüssels.
- Der Search-Key muss **nicht unbedingt der Primärschlüssel** sein, ist aber meistens ein eindeutiges Attribut.
- **Suche: Binäre Suche** bei Suche anhand des Search-Keys möglich, was deutlich effizienter ist als der Linear Scan bei Heap Files (logarithmische Komplexität).
- **Einfügen:** Erfordert möglicherweise eine **Reorganisation der Datei**, um die Sortierreihenfolge beizubehalten. Neue Tupel müssen an der richtigen Position eingefügt werden, was das Verschieben anderer Tupel erfordern kann.

1	Aaen	E111
2	Dolog	E116
3	Larsen	E117
4	Ravn	E161
5	Srba	E166
6	Rose	E167
7	Torp	E171
8	Lazy	E176

Hashing

- **Hashfunktion bestimmt Reihenfolge:** Eine Hashfunktion wird auf den Search-Key angewendet, um die Speicheradresse (genauer gesagt, die Bucket- oder Blocknummer) für das Tupel zu bestimmen. Tupel mit ähnlichen Hashwerten werden im selben Bucket gespeichert.
- **Suche:** Benutze die Hashfunktion mit dem Wert des Search-Keys (z.B. ID-Attribut), um den **richtigen Block/Seite direkt zu finden**. Im Idealfall ist ein direkter Zugriff auf den Speicherort möglich.
- **Einfügen:** Benutze die Hashfunktion mit dem Wert des Search-Keys, um den **richtigen Block zu finden und füge das Tupel dort hinzu**. Es kann zu Kollisionen kommen, wenn unterschiedliche Suchschlüssel auf denselben Bucket abgebildet werden. Diese müssen durch spezielle Techniken (z.B. separate Verkettung) behandelt werden.

3	Larsen	E117
6	Rose	E167
4	Ravn	E161
1	Aaen	E111
7	Torp	E171
5	Srba	E166
2	Dolog	E116
8	Lazy	E176

Indexstrukturen

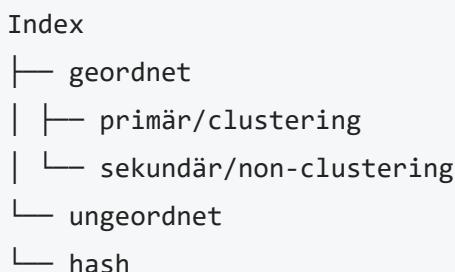
Annahmen (im Kontext von Indexstrukturen)

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

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

Übersicht: Indexstrukturen

Klassifikation von Indexen:



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

Variationen:

- **Single-Level Index:** Der Index besteht aus einer einzigen Ebene.

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

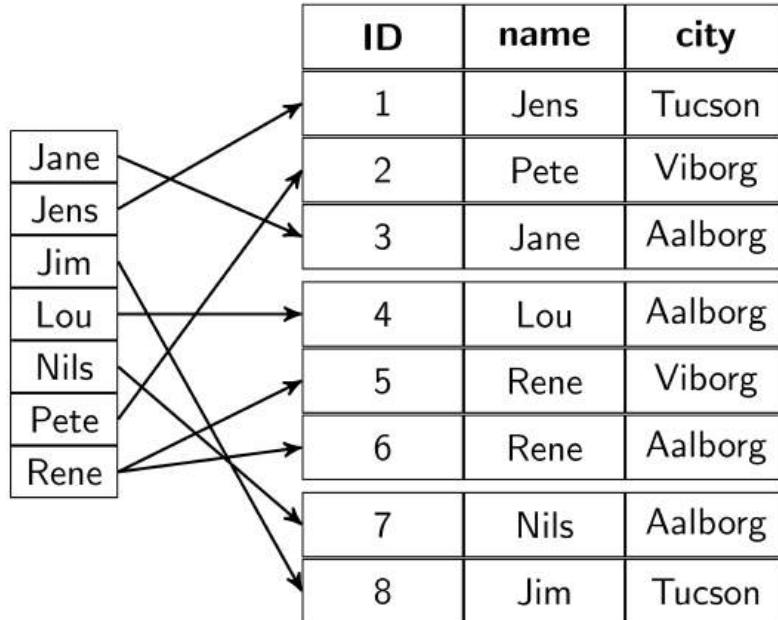
Es ist möglich, **mehrere Indexe auf der gleichen Tabelle zu definieren**, um unterschiedliche Arten von Abfragen effizient zu unterstützen. Die Wahl der Indexe hängt von den häufigsten Abfragemustern ab.

Primary Sparse Index

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

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

Secondary Dense Index



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

Indexstrukturen - Geordnete Indexe

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

Primär (Primary) vs. Sekundär (Secondary)

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

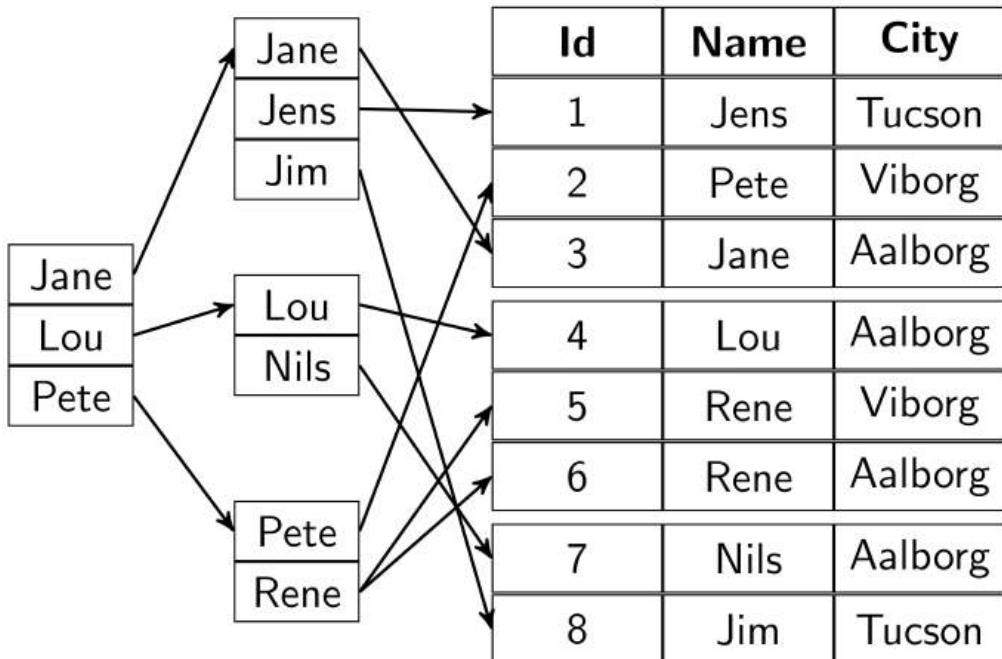
Dense vs. Sparse

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

Tradeoff

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

Multi-Level Index



- Ziel: der äußere Index (sparse) passt in den Hauptspeicher
- Der Index kann mehr als 2 Ebenen haben

Einschränkungen von geordneten Dateistrukturen

- **Einfügen und Löschen:**
 - Kann zu einer **teuren Reorganisation von mehreren Ebenen von sortierten Dateien** führen, um die Sortierreihenfolge beizubehalten. Dies ist besonders aufwendig bei sequenzieller Dateiorganisation.

B+-Bäume

- **Balancierte Suchbäume:** Stellen sicher, dass der Suchpfad von der Wurzel zu jedem Blattknoten ungefähr gleich lang ist, was eine effiziente Suche garantiert (logarithmische Komplexität).
- **Die Anzahl von Lookups/Levels ist für alle Einträge gleich:** Dies sorgt für eine vorhersagbare und gute Suchperformance, unabhängig vom gesuchten Schlüssel.
- **Etwas Platz auf jeder Seite/Block lassen:** Um zukünftige Einfügungen zu ermöglichen, ohne sofort eine Reorganisation durchführen zu müssen. Dies verbessert die Performance bei häufigen Einfügeoperationen. B+-Bäume sind eine häufig verwendete Indexstruktur in Datenbanksystemen aufgrund ihrer Effizienz bei Such-, Einfüge- und Löschoperationen sowie bei Bereichsabfragen

Kombinationen von Konzepten (Indexarten)

	dense	sparse
primary (clustering)	✓	✓

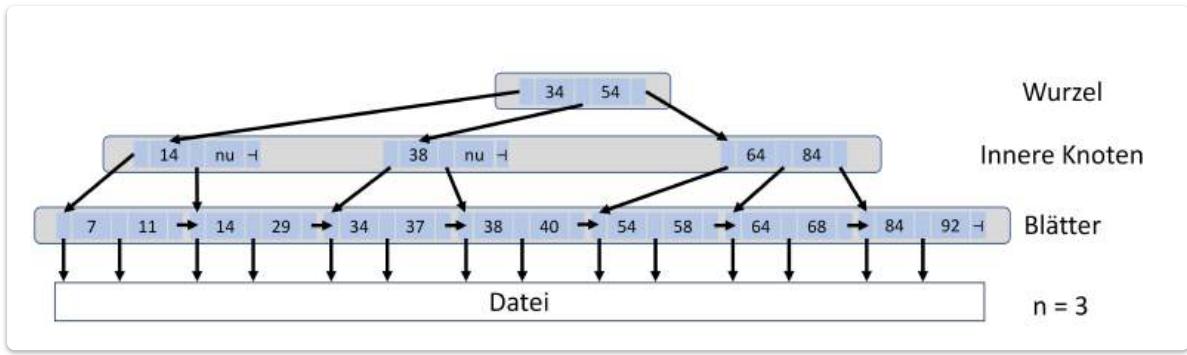
	dense	sparse
secondary (non-clustering)	✓	%

Ein Secondary Sparse Index ist nicht sinnvoll!

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

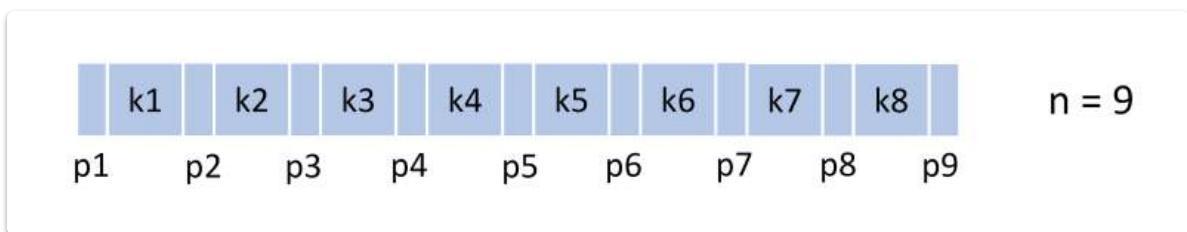
Indexstrukturen - B^+ Bäume

B^+ Baum Beispiel



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

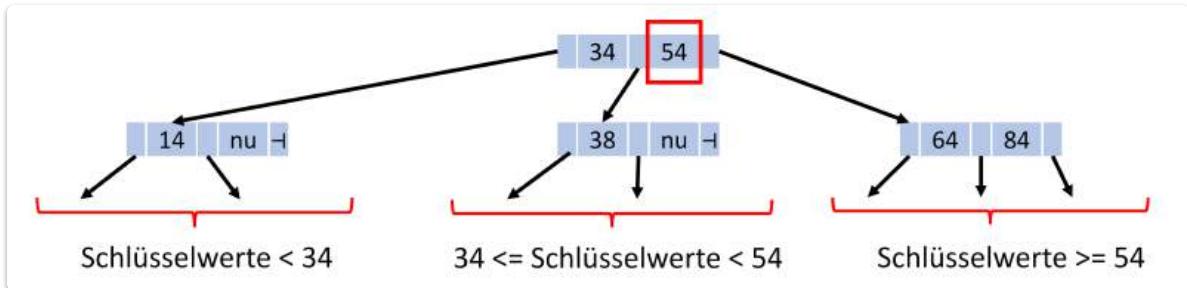
B^+ -Baum Knotenstruktur



- **Wurzel, innere Knoten und Blätter haben die gleiche Struktur.**
- **Jeder Knoten hat n Pointer p_i .** Hier ist $n = 9$. Die Pointer in internen Knoten zeigen auf Kindknoten. In Blattknoten zeigen die Pointer auf Datensätze (oder Speicheradressen der Datensätze) oder, im Fall des letzten Pointers, auf den nächsten Blattknoten.
- **Jeder Knoten hat at most ($n - 1$) Search-Key-Werte k_j .** Hier also maximal $9 - 1 = 8$ Schlüsselwerte. Die Schlüsselwerte in internen Knoten dienen als Wegweiser, um den richtigen Unterbaum für die Suche zu finden. In Blattknoten sind sie die tatsächlichen Suchschlüssel der gespeicherten Datensätze (oder eine Kopie davon).

- Der letzte Pointer auf Blattebene zeigt auf den nächsten Blattknoten in Search-Key-Reihenfolge. Dies ist ein wichtiges Merkmal von B+-Bäumen, das effiziente sequentielle Durchläufe (Bereichsabfragen) der Daten ermöglicht, ohne zum Wurzelknoten zurückkehren zu müssen. Die Blattknoten sind also einfach verkettet.

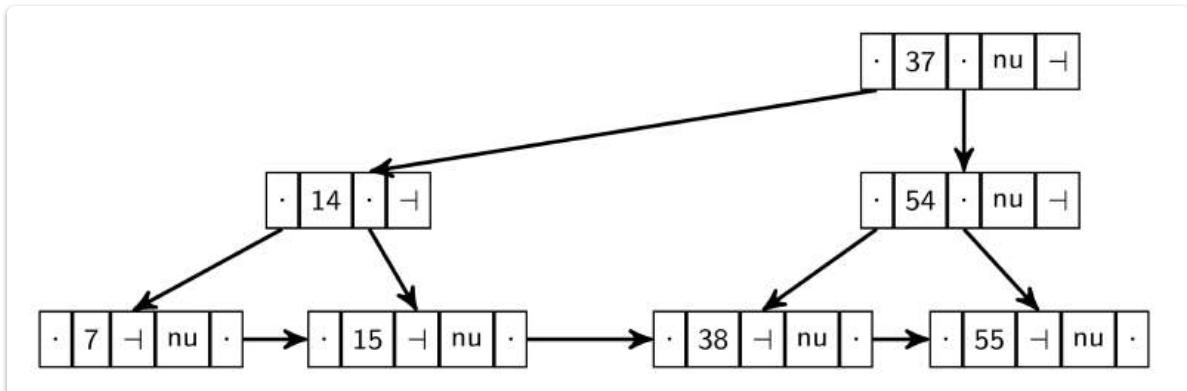
B+-Baum Eigenschaften



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

Ein minimaler B+-Baum

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



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

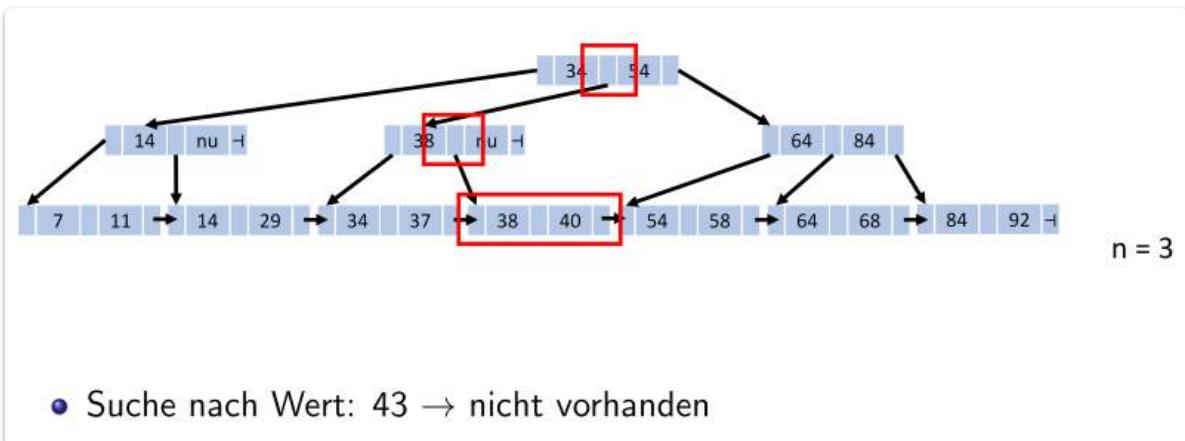
Der gezeigte Baum illustriert diese Minimalbedingungen. Beachte die Verkettung der Blattknoten für sequenzielle Zugriffe.

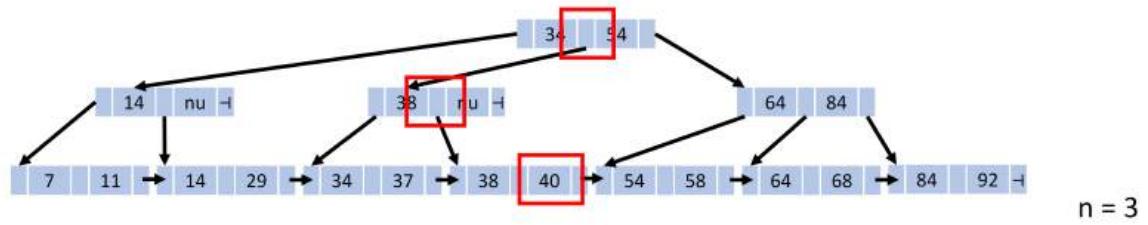
Verwendung von B⁺-Bäumen in DBMS

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

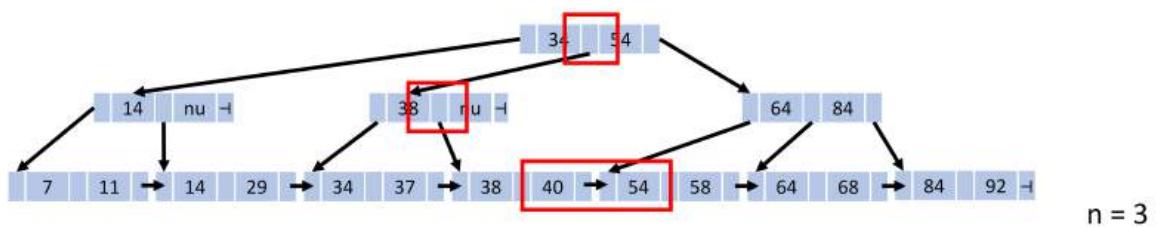
Uniqueness-Constraints: Uniqueness-Constraints auf Attributen in einer Datenbank werden durch B⁺-Bäume realisiert → **Primärschlüssel** (B⁺-Bäume eignen sich gut zur Durchsetzung von Eindeutigkeitsbedingungen und für effiziente Bereichsabfragen).

Beispiel zum Lookup



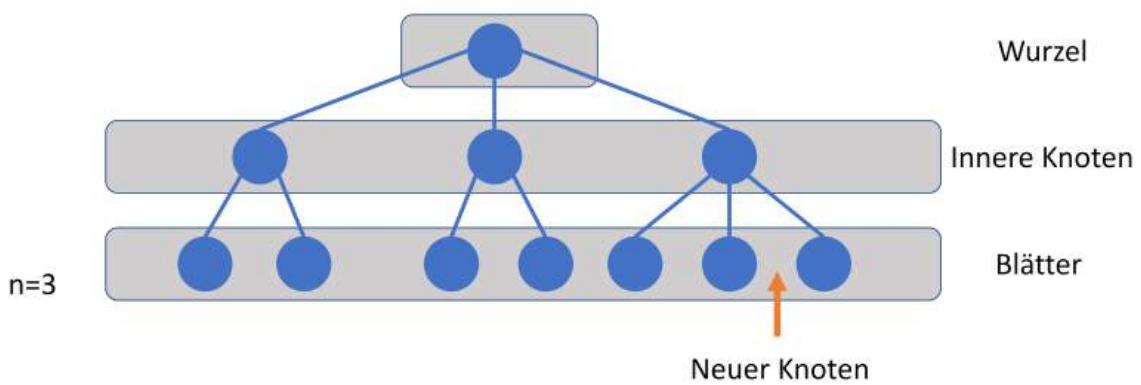


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden

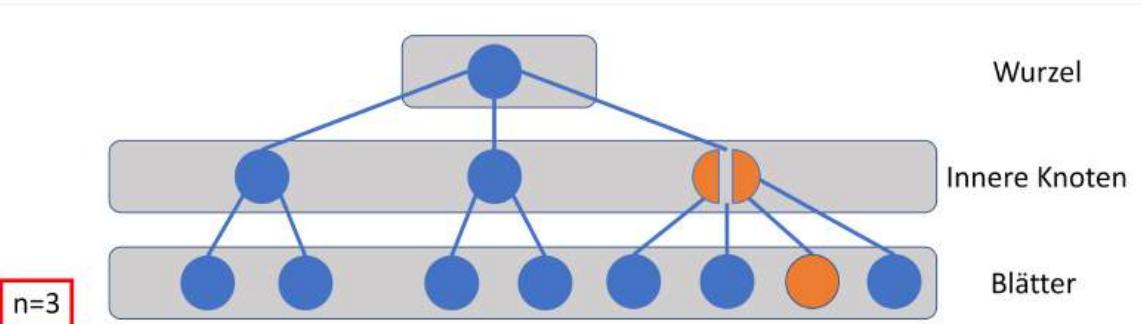


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden
- Suche nach Bereich: 40-55 → gefunden

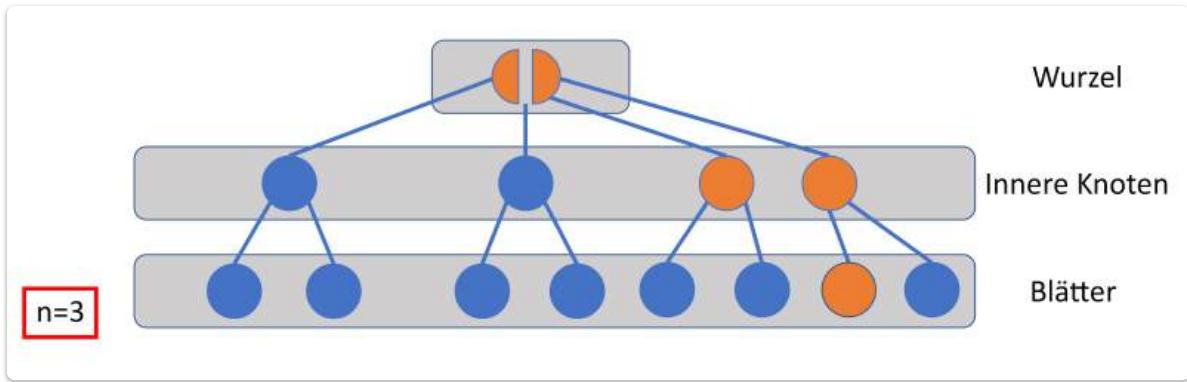
Einfügen beim B^+ Baum



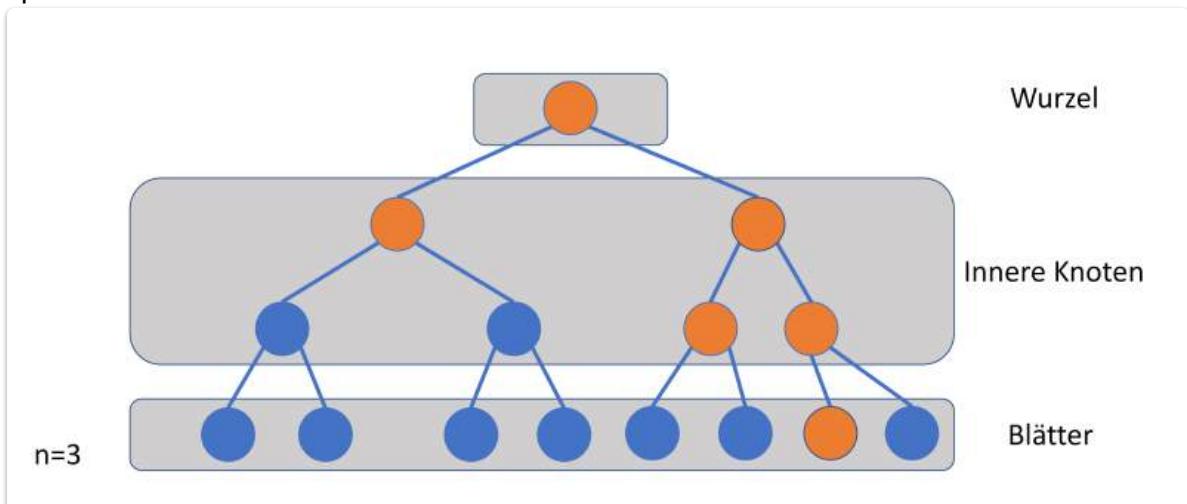
Wir wollen hier den neuen Knoten einfügen



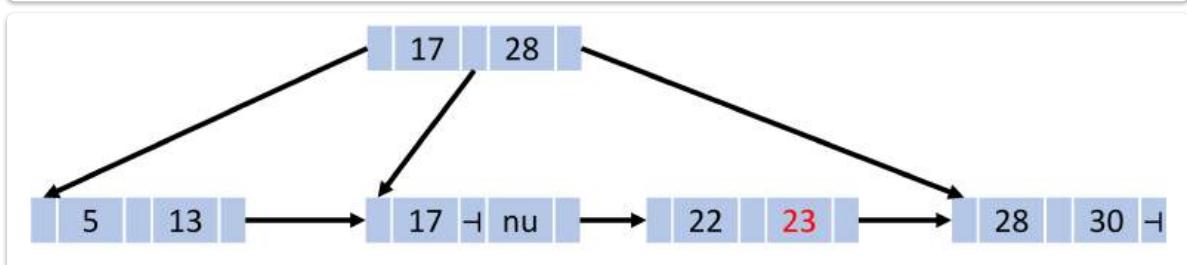
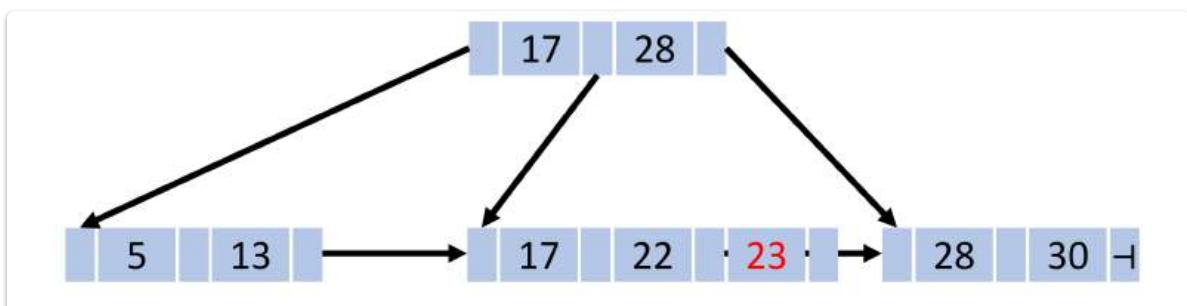
Da unser $n = 3$ müssen wir eine Ebene darüber in 2 teilen

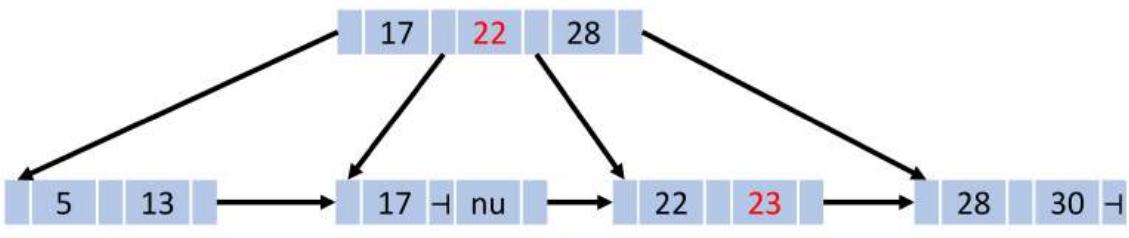


Dann haben wir das selbe Problem in einer Ebene darüber. Da müssen wir auch ein mal spalten und haben am Ende eine Ebene mehr



Beispiel: Einfügen von 23





Mehr Beispiele: [DBS-8_Physischer Datenbankentwurf, p.32](#)

B⁺-Baum Löschen

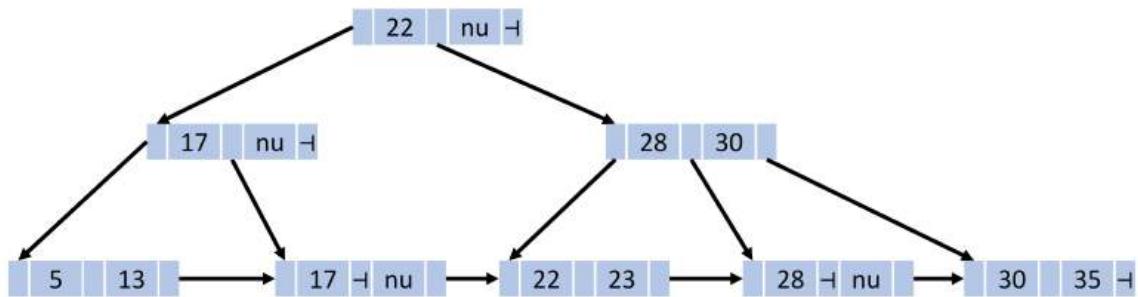
Löschen von Schlüsselwert k:

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

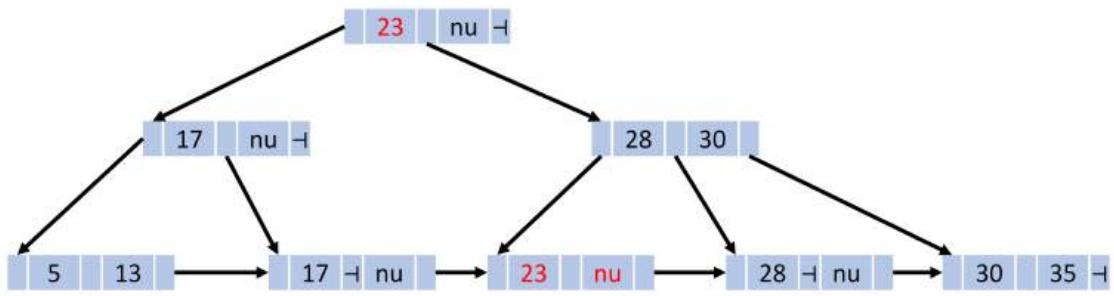
Beispiele:

Löschen von 22

n=3, Löschen von 22

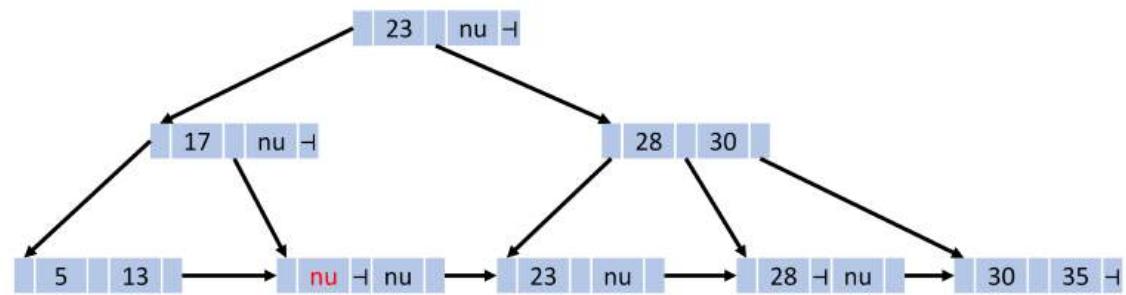


n=3, Löschen von 22

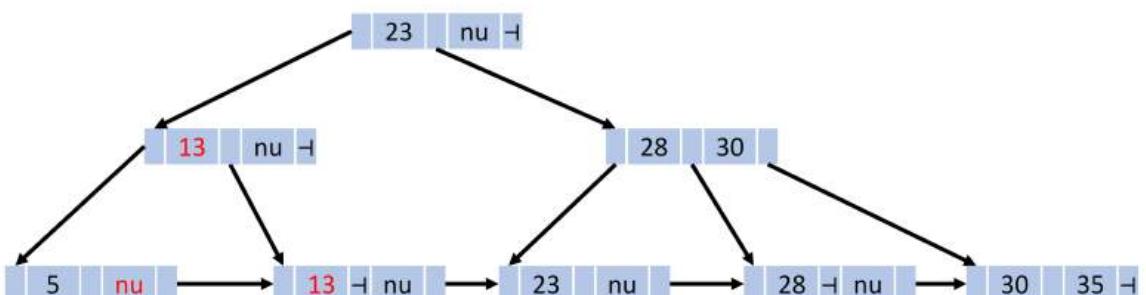


Löschen von 17

n=3, Löschen von 17

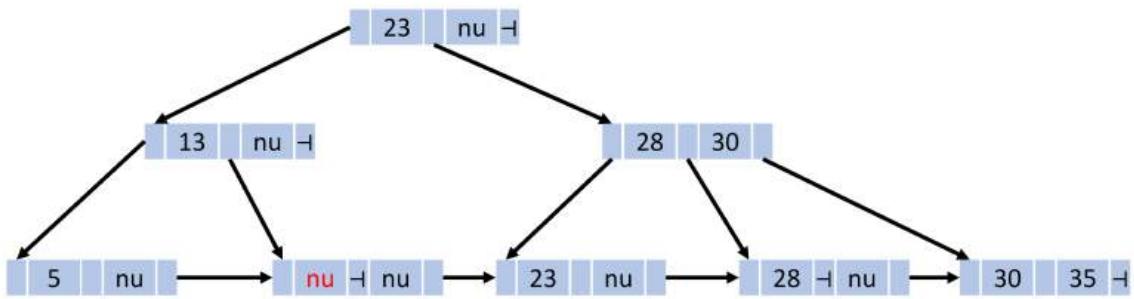


n=3, Löschen von 17

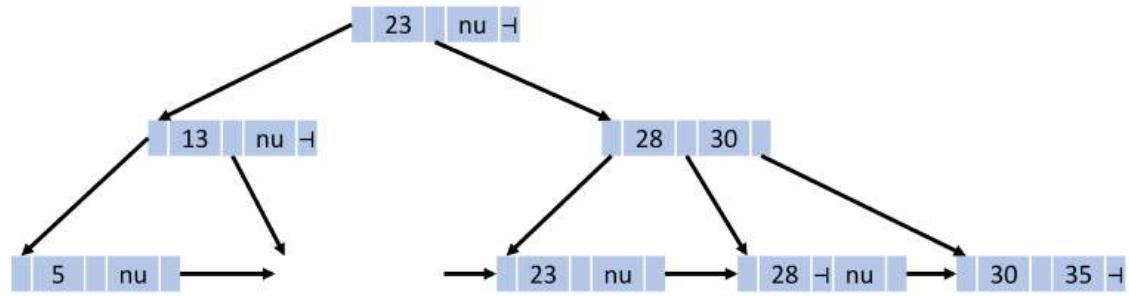


Löschen von 13

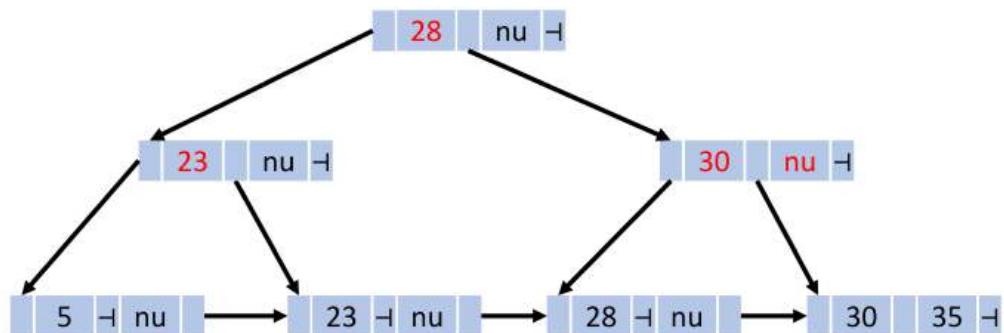
n=3, Löschen von 13



n=3, Löschen von 13

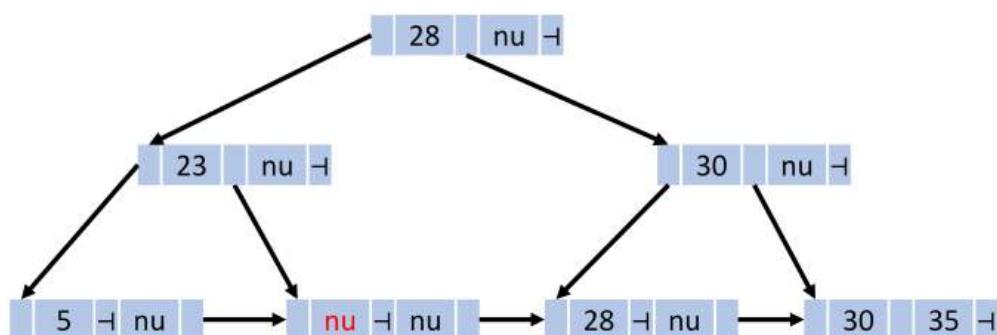


n=3, Löschen von 13

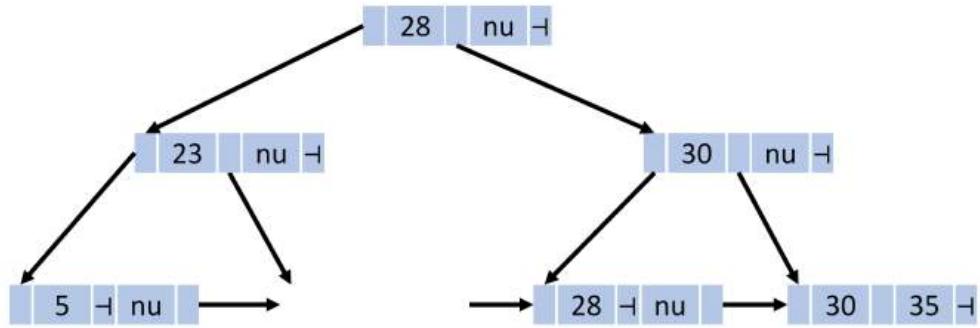


Löschen von 23

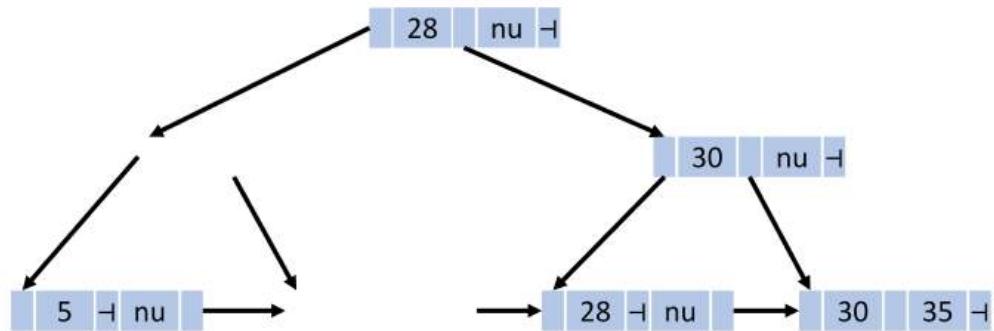
n=3, Löschen von 23



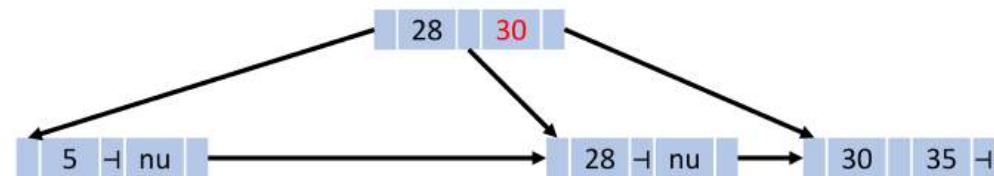
n=3, Löschen von 23



n=3, Löschen von 23



n=3, Löschen von 23

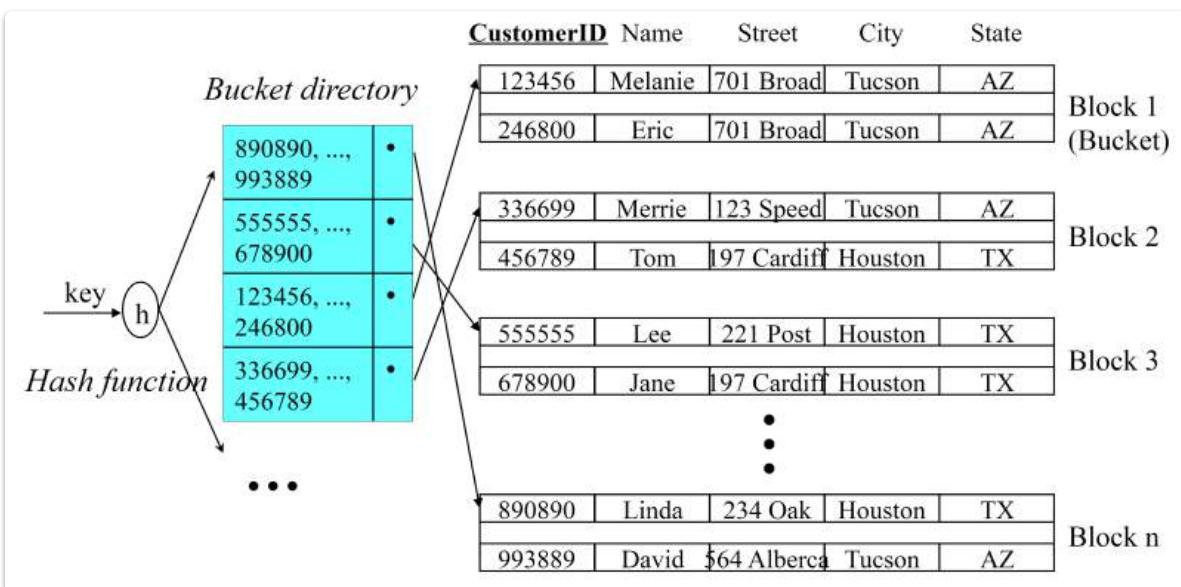


Indexstrukturen - Hashing

Statischer Hash-Index

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

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



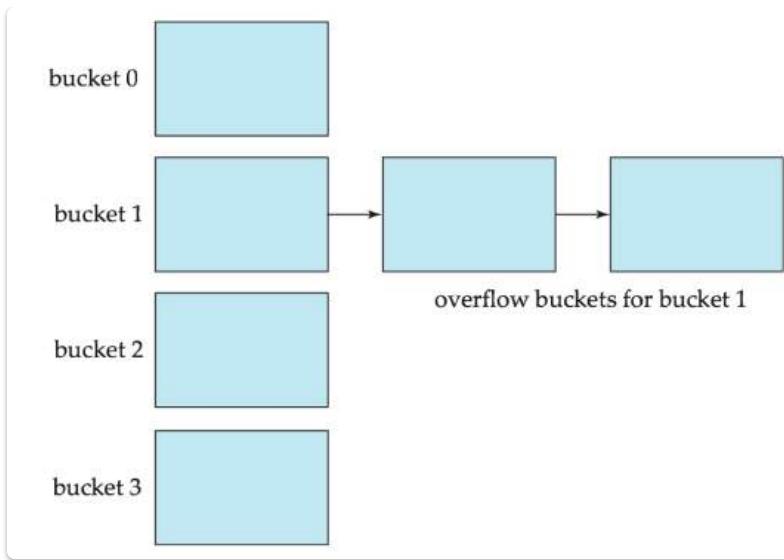
(Die Abbildung zeigt eine Tabelle mit Kundendaten, eine Hashfunktion h , ein Bucket Directory, das Hashes auf Buckets abbildet, und die eigentlichen Datenblöcke (Buckets) mit den Datensätzen.)

Suche:

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

Gute Performance hängt von einer guten Hashfunktion ab:

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



Statischer Hash-Index

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

Static Hashing

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

Datenbanken wachsen und schrumpfen mit der Zeit!

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

Lösungsansätze:

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

Design Tuning

Optimierungsgegenstände:

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

Grundsätzliche Fragestellungen:

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

Nutzung von Indexen

Manchmal werden existierende Indexe nicht verwendet:

- **System-Katalog hat veraltete Informationen** (der Query-Optimierer trifft Entscheidungen basierend auf möglicherweise falschen Statistiken).
- **Optimierer könnte annehmen, dass die Tabelle klein ist** (bei kleinen Tabellen kann ein Full Table Scan effizienter sein als die Indexnutzung).
- **"Gute" und "schlechte" Typen von Anfragen (bezüglich Indexnutzung):**
 - `SELECT * FROM EMP WHERE salary/12 > 4000` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).
 - `SELECT * FROM EMP WHERE salary > 48000` (Bereichsabfragen können Indexe gut nutzen).
 - `SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G'` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).
 - `SELECT * FROM EMP WHERE name LIKE 'G%'` (führende Wildcards können Indexnutzung einschränken, nachfolgende Wildcards oft nicht).
 - `SELECT * FROM EMP WHERE name = 'Smith'` (Punktuelle Suchen auf indizierten Spalten sind ideal für Indexe).
 - `SELECT * FROM EMP WHERE salary IS NULL` (Indexe können oft auch für IS NULL-Bedingungen genutzt werden).
- **Geschachtelte SQL-Anfragen** (die Effizienz der Indexnutzung kann von der Struktur der geschachtelten Anfrage abhängen).
- **Negationen** (`NOT`, `!=`) (können die Indexnutzung erschweren).
- **Anfragen mit OR** (die Indexnutzung bei OR-Bedingungen hängt von der Verfügbarkeit von Indexen auf den beteiligten Spalten ab und wie der Optimierer die Anfrage

umformuliert).

Zusammenfassung

- Speicherhierarchie und Festplatten.
- Tupel mit variabler Länge machen die Dateiorganisation komplexer.
- Keine Dateiorganisation ist die beste für alle Anwendungen.
- Clustering vs. non-clustering Index.
- Sparse vs. dense Index.
- Single-level and multi-level Index.
- Der B⁺-Baum ist der wichtigste Index für Datenbanksysteme.
- Tuning hängt von vielen Aspekten ab, z.B. Query Load, Charakteristika der Daten, Systemvoraussetzungen etc.

Kompromiss zwischen Speicherplatz und Zeit: Intelligente Nutzung von zusätzlichem Speicherplatz erhöht im Allgemeinen die Performance (z.B. durch Caching von Indexen).

Kompromiss zwischen Select-Anfragen und Updates: Wenn Maßnahmen zur Beschleunigung von Select-Anfragen getroffen werden, geschieht dies meist zum Nachteil von Updates – und umgekehrt (z.B. viele Indexe beschleunigen Suchen, verlangsamen aber Schreiboperationen).

7. Anfrageoptimierung

Ausführen einer SQL-Anfrage

Klauseln und ihre Reihenfolge

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

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

Ausführungsreihenfolge einer SQL-Anfrage

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

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

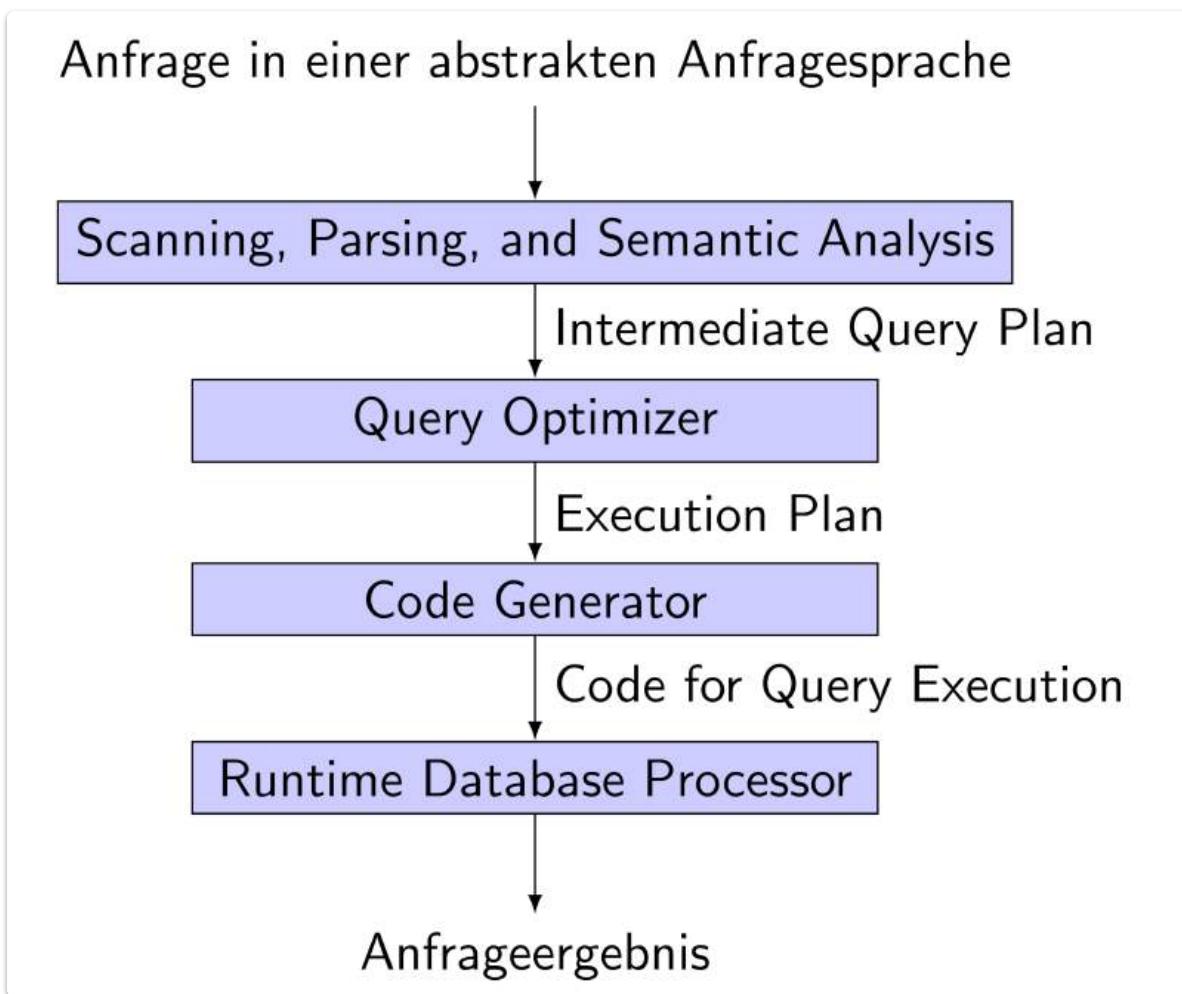
- **Erklärung:** Das endgültige Ergebnis wird nach den in `sort list` angegebenen Spalten sortiert.

SQL ist deklarativ!

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

Schritte der Anfragebearbeitung

Die Verarbeitung einer Anfrage in einem Datenbanksystem durchläuft mehrere Schritte:



1. Anfrage in einer abstrakten Anfragesprache (z.B. SQL):

- Dies ist die vom Benutzer eingegebene Anfrage.

2. Scanning, Parsing, und Semantic Analysis:

- **Scanning (Lexikalische Analyse):** Der Anfragetext wird in Tokens (kleinste syntaktische Einheiten, z.B. Schlüsselwörter, Operatoren, Bezeichner) zerlegt.
- **Parsing (Syntaktische Analyse):** Die Tokens werden gemäß der Grammatik der Anfragesprache in einen **Parse Tree** (Syntaxbaum) oder eine **Anfragespezifikation** (Query Specification) übersetzt.

umgewandelt. Dabei wird geprüft, ob die Syntax korrekt ist.

- **Semantic Analysis (Semantische Analyse):** Hier wird die Bedeutung der Anfrage überprüft. Dazu gehören:
 - Prüfung, ob alle angegebenen Tabellen und Spalten existieren.
 - Typenprüfung (z.B. ob Vergleiche zwischen kompatiblen Datentypen stattfinden).
 - Auflösung von Namen und Aliassen.
 - Berechtigungsprüfung (ob der Benutzer die nötigen Rechte hat).
- Das Ergebnis dieses Schrittes ist ein **Intermediate Query Plan** (ein Zwischen-Anfrageplan, z.B. in Form eines Operatorbaums der relationalen Algebra).

3. Query Optimizer (Anfrageoptimierer):

- Dies ist eine der wichtigsten Komponenten eines DBMS.
- Der Optimierer nimmt den Intermediate Query Plan und sucht nach dem **effizientesten Ausführungsplan**.
- Es gibt oft mehrere Wege, dieselbe Anfrage zu beantworten (z.B. verschiedene Reihenfolgen von Joins oder Filteroperationen).
- Der Optimierer bewertet die Kosten der verschiedenen Pläne (z.B. geschätzte Anzahl der zu lesenden Blöcke, CPU-Kosten) und wählt den Plan mit den niedrigsten geschätzten Kosten aus.
- Das Ergebnis ist ein **Execution Plan** (Ausführungsplan).

4. Code Generator:

- Der Code Generator nimmt den optimierten Execution Plan und generiert den eigentlichen Code, der vom Datenbankprozessor ausgeführt werden kann.
- Dies kann maschinennaher Code oder eine Sequenz von Aufrufen von Laufzeitroutinen sein.

5. Code for Query Execution:

- Der generierte Code, bereit zur Ausführung.

6. Runtime Database Processor (Laufzeit-Datenbankprozessor):

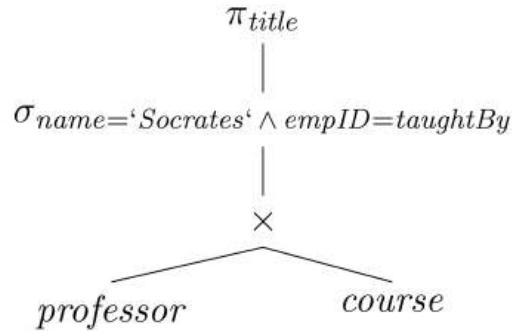
- Führt den generierten Code aus.
- Interagiert mit anderen Komponenten des DBMS (z.B. Puffermanagement, Speicherverwaltung, Transaktionsmanager), um die benötigten Daten zu lesen, zu verarbeiten und das Ergebnis zu liefern.

7. Anfrageergebnis:

- Das Endergebnis, das dem Benutzer präsentiert wird.

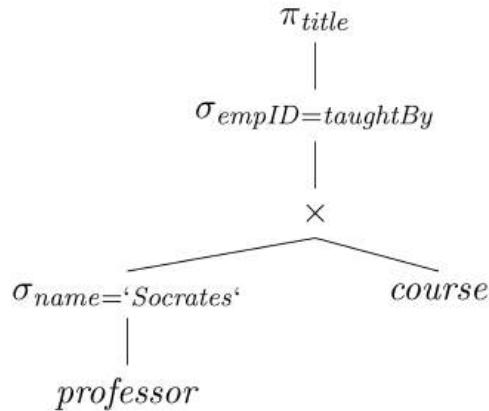
Alternativer Anfrageplan (Beispiel)

SELECT title
 FROM professor, course
 WHERE name='Socrates' AND
 empID = taughtBy;



$\pi_{title}(\sigma_{name='Socrates' \wedge empID=taughtBy}(professor \times course))$

SELECT title
 FROM professor, course
 WHERE name='Socrates' AND
 empID = taughtBy;



$\pi_{title}(\sigma_{empID=taughtBy}(\sigma_{name='Socrates'}(professor \times course)))$

Anfrageoptimierung

Alternativen bei der Anfrageoptimierung

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

- **Äquivalente Ausführungspläne:** Für dieselbe Anfrage gibt es oft mehrere Ausführungspläne, die dasselbe Ergebnis liefern. Der Optimierer wählt den kostengünstigsten.
- **Algorithmen zur Ausführung von Algebraoperatoren:** Für relationale Algebraoperatoren (z.B. Join, Selektion, Projektion) gibt es verschiedene Implementierungsalgorithmen (z.B. Nested-Loop Join, Hash Join, Sort-Merge Join). Der Optimierer wählt den passenden Algorithmus.

- **Methoden, um auf Relationen zuzugreifen (Indizes):** Es gibt verschiedene Möglichkeiten, auf Daten in Tabellen zuzugreifen. Ob ein Index verwendet wird oder ein vollständiger Tabellen-Scan, hängt von den Kosten ab.

Bei gleichem Ergebnis können Ausführungskosten sehr unterschiedlich sein.

Theorie vs. Realität

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

Anfrageausführungskosten

Kostenmodell

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

- **Festplattenzugriff (I/O-Kosten):**
 - Dies ist oft der **dominierende Faktor** bei den Kosten.
 - Daten müssen von der Festplatte in den Hauptspeicher geladen werden.
 - **Block Access Time:**
 - **Seek Time:** Die Zeit, die der Schreib-/Lesekopf benötigt, um zu der richtigen Spur auf der Festplatte zu gelangen.
 - **Rotation Time:** Die Zeit, die der benötigte Sektor benötigt, um unter dem Schreib-/Lesekopf vorbeizudrehen.
- **CPU-Kosten:**
 - Kosten für die Verarbeitung von Daten im Hauptspeicher (z.B. Sortieren, Hashen, Vergleichen).
- **Netzwerkkommunikation:**
 - Wenn Daten über ein Netzwerk abgerufen oder Ergebnisse an einen Client gesendet werden müssen.
- **Aktueller Query-Load:**
 - Die Anzahl und Art der gleichzeitig laufenden Anfragen kann die Performance beeinflussen (Konkurrenz um Ressourcen).
- **Parallelisierung:**
 - Wenn Operationen parallel ausgeführt werden können, kann dies die Antwortzeit verkürzen. Kosten für die Koordination.

Logische Anfrageoptimierung

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

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

Physische Anfrageoptimierung

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

- **Algorithmen und Operatorimplementationen:**
 - Hier werden die spezifischen Algorithmen für die relationalen Algebraoperatoren ausgewählt (z.B. welcher Join-Algorithmus verwendet wird: Nested-Loop Join, Hash Join, Sort-Merge Join).
 - Auch die Zugriffspfade auf die Daten werden festgelegt (z.B. Tabellen-Scan oder Index-Scan).
- **Kostenmodell:**
 - Ein Kostenmodell wird verwendet, um die geschätzten Ausführungskosten für verschiedene physische Pläne zu berechnen.
 - Dabei werden Faktoren wie CPU-Zeit, E/A-Operationen (Festplattenzugriffe) und Netzwerkkommunikation berücksichtigt.
 - Der Plan mit den geringsten geschätzten Kosten wird ausgewählt.

Logische (heuristische) Anfrageoptimierung

Logische Anfrageoptimierung

Grundlagen der Logischen Anfrageoptimierung

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

Was ist ein guter Plan?

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

Hauptziel der logischen Anfrageoptimierung

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

Äquivalenzerhaltende Transformationsregeln

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

Aufbrechen von Konjunktionen in Selektionsprädikaten

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

σ ist kommutativ

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

π -Kaskaden

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

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

Vertauschen der Reihenfolge von σ und π

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

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

\cup, \cap und \bowtie sind kommutativ

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

Vertauschen von σ und \bowtie

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

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

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

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

Vertauschen von π und \bowtie

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

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

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

Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

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

σ ist distributiv mit $\cap, \cup, -$

Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

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

π ist distributiv mit \cup

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

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

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

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

Kombination von Kartesischem Produkt und Selektion

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

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

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

Phasen der logischen Anfrageoptimierung

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

1. Aufbrechen von Selektionen:

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

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

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

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

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

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

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

5. ggf. Einführen von Projektionen:

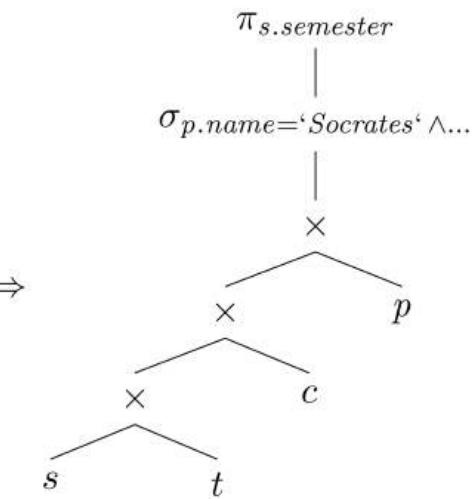
- Projektionen (SELECT-Klausel) wählen die benötigten Spalten aus. Manchmal müssen Projektionen eingefügt werden, um unnötige Spalten frühzeitig zu eliminieren.

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

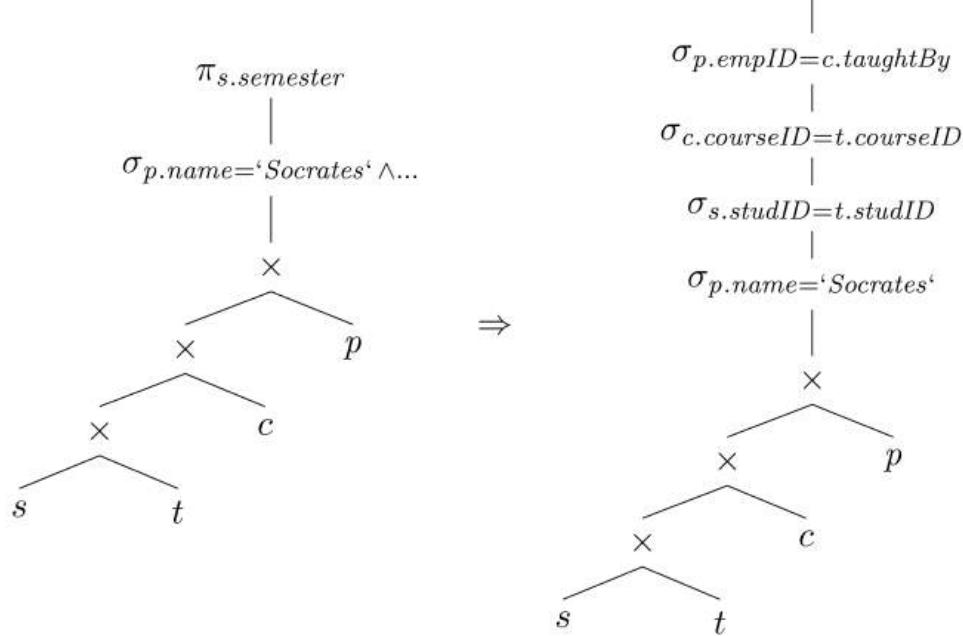
- Ähnlich wie bei Selektionen kann es vorteilhaft sein, Projektionen so früh wie möglich anzuwenden, um die Breite der Zwischenergebnisse zu reduzieren (d.h., weniger Spalten zu verarbeiten).
- **Nicht immer nötig:** Manchmal sind Projektionen nach Joins oder Aggregationen sinnvoller, insbesondere wenn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

Beispiel

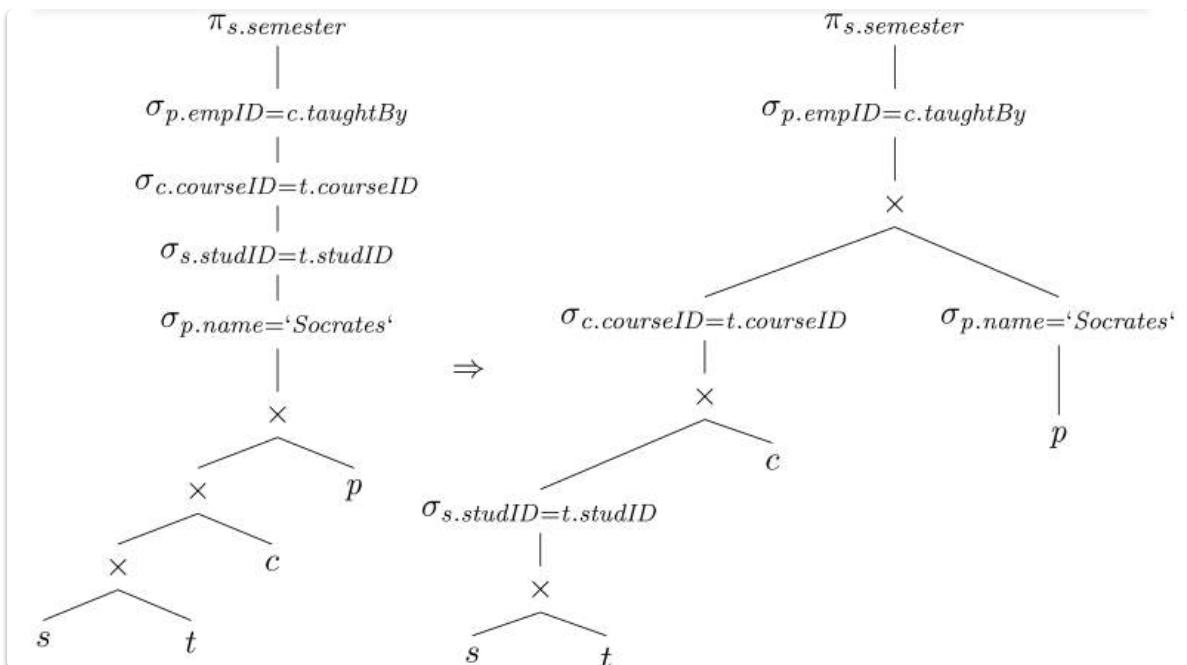
SELECT DISTINCT s.semester
 FROM student s, takes t,
 course c, professor p
 WHERE p.name='Socrates' AND
 c.taughtBy = p.empID AND
 c.courseID = t.courseID AND
 t.studID = s.studID;



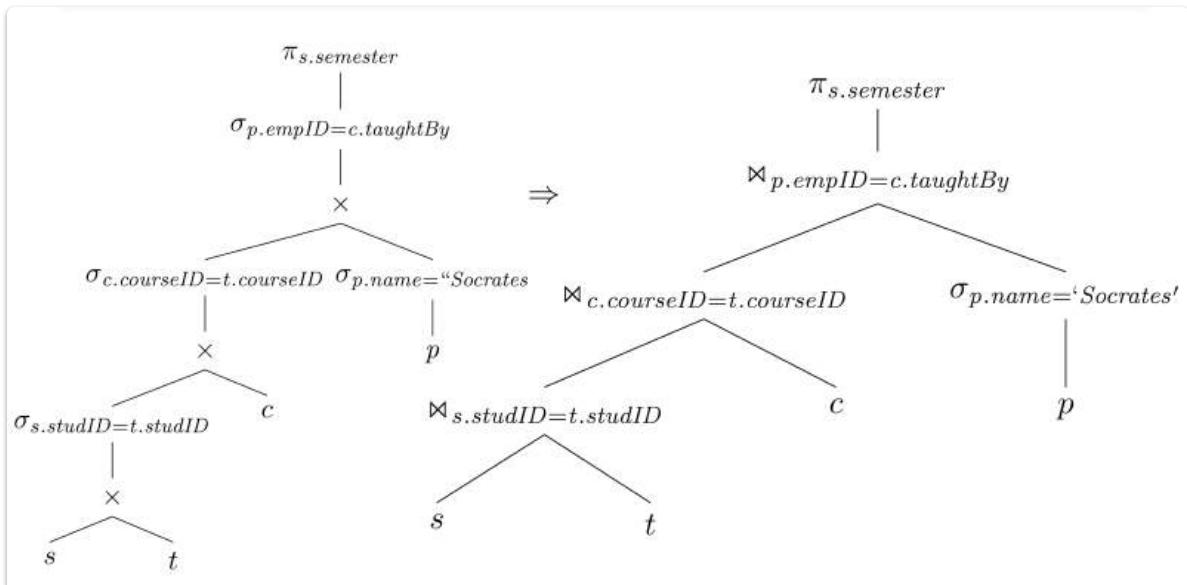
Aufbrechen von Selektionen



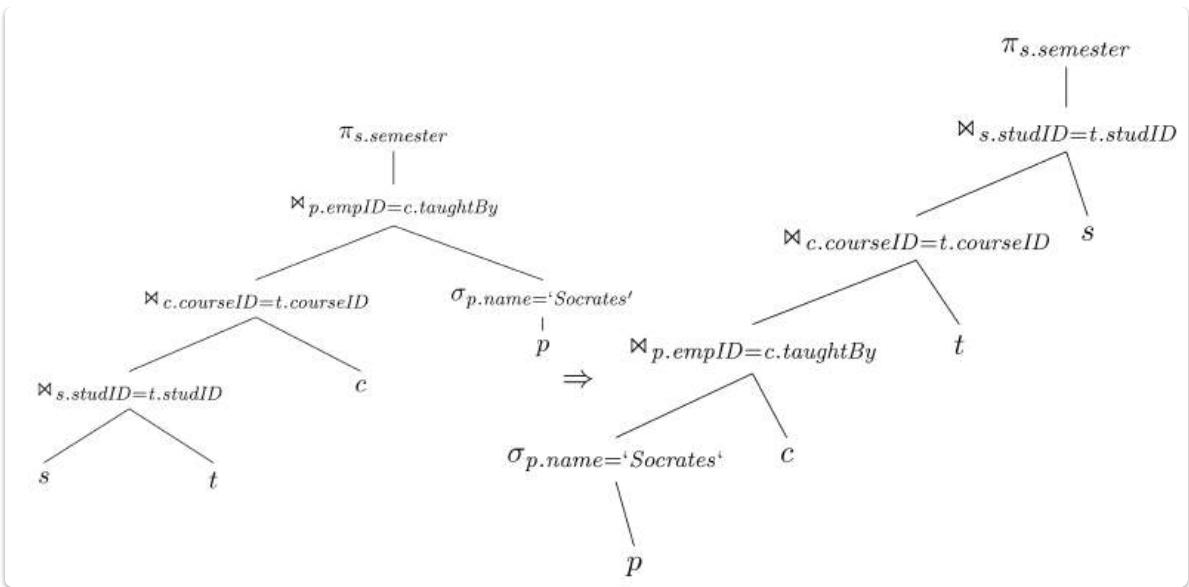
Verschieben von Selektionen



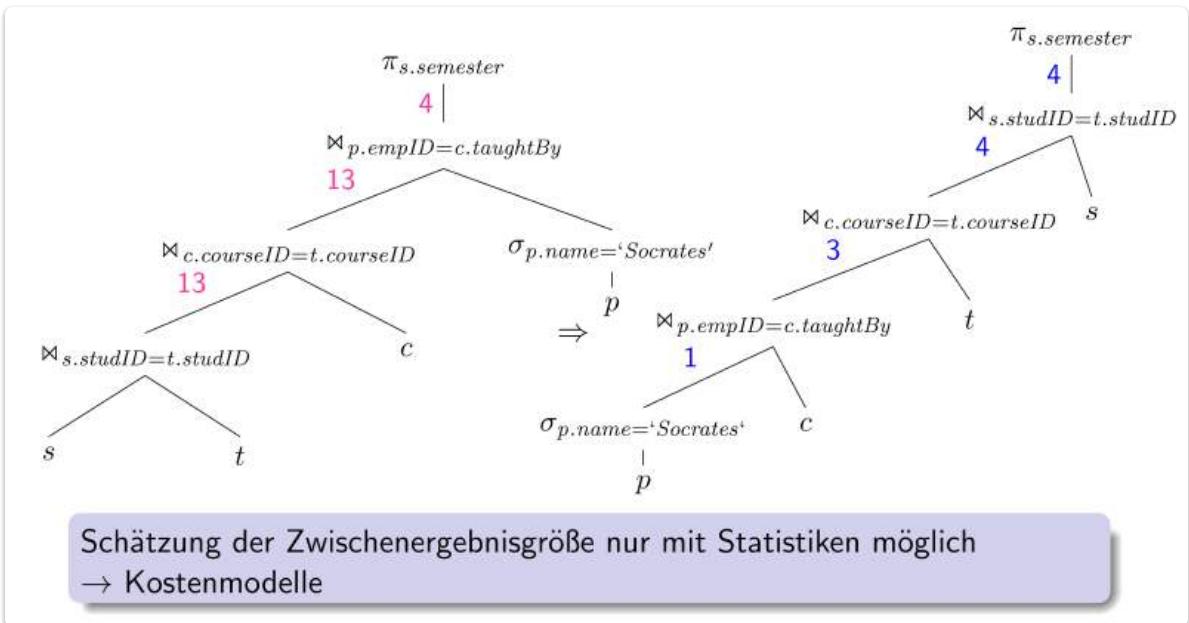
Joins einführen



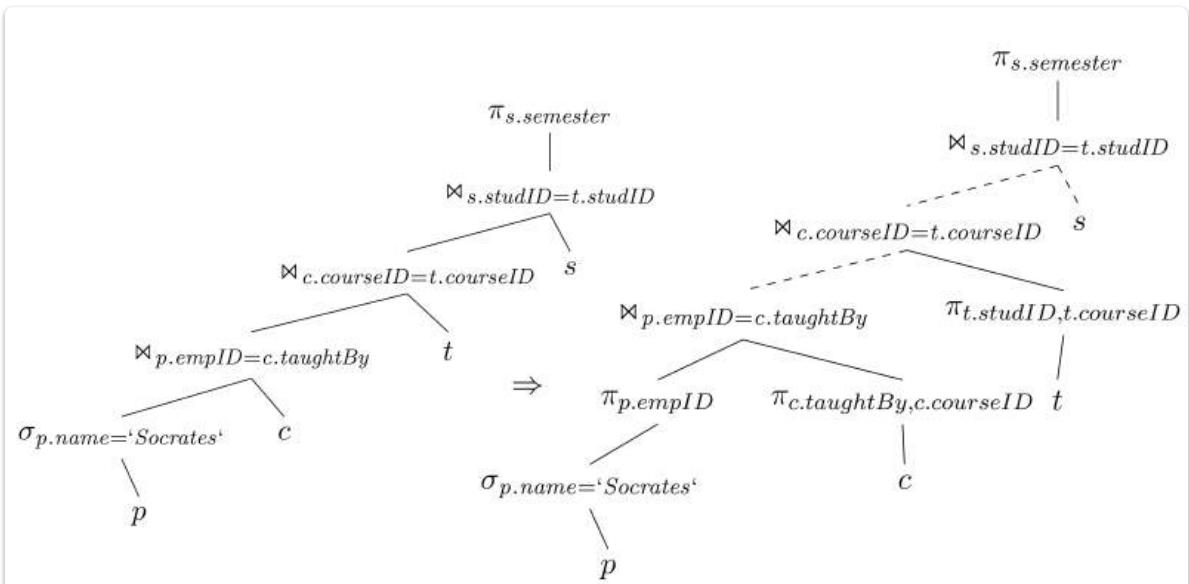
Joinreihenfolge bestimmen



Effekt: Verkleinerung der Zwischenergebnisse



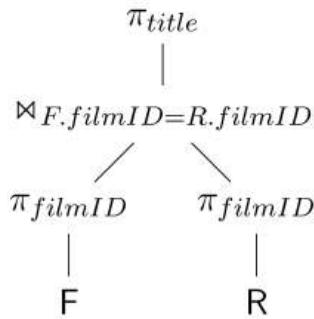
Einführen und verschieben von Projektionen



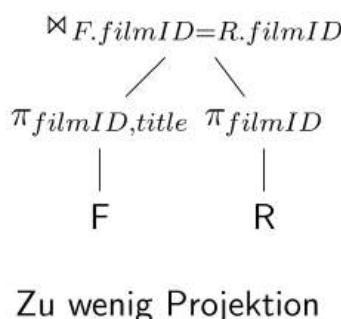
Vorsicht - Beispiele

Finde die Titel von reservierten Filmen

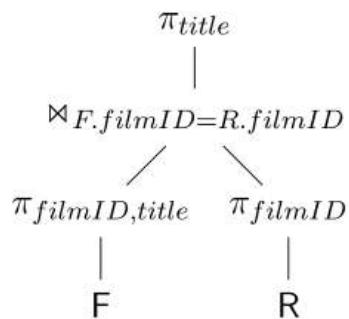
```
SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID
```



Zu viel Projektion



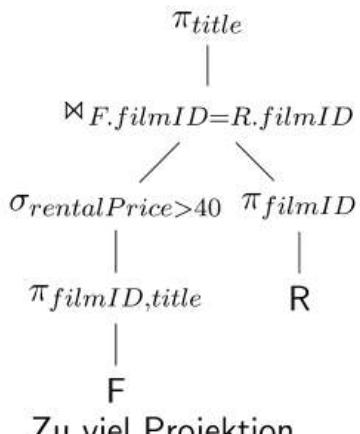
Zu wenig Projektion



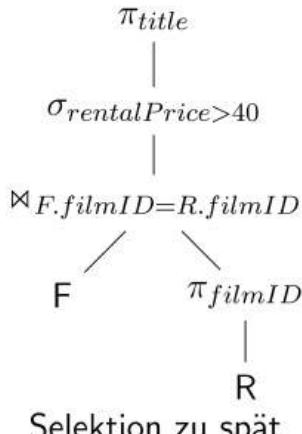
Korrekt

Finde die Titel von teuren reservierten Filmen

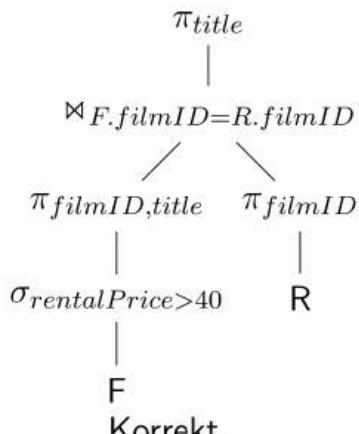
```
SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID AND F.rentalPrice > 40
```



Zu viel Projektion



Selektion zu spät



Korrekt

Zusammenfassung: heuristische Anfrageoptimierung

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

Erfahrungsregeln

Die wichtigsten Heuristiken (Erfahrungsregeln) der logischen Anfrageoptimierung sind:

- Selektionen so früh wie möglich ausführen:

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

Optimierungsprozess

Der allgemeine Prozess der heuristischen Anfrageoptimierung umfasst zwei Schritte:

1. **Erstelle initialen Plan aus der SQL-Anfrage:**
 - Die ursprüngliche SQL-Anfrage wird in einen ersten, meist naiven Operatorbaum der relationalen Algebra übersetzt (z.B. ein Parsen in einen Baum).
2. **Modifiziere den Plan, um ihn in einen effizienteren zu überführen:**
 - Äquivalenzerhaltende Transformationsregeln und die oben genannten Heuristiken werden angewendet, um den initialen Plan schrittweise in einen effizienteren Ausführungsplan umzuwandeln.

Hinweis

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

Implementierung von Operatoren

Selektion (Access Paths)

Verschiedene Verwendungen von Select

- Primary Key, Punkt

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

- Punkt

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

- Bereich

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

- Konjunktion (logisches und)

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

- Disjunktion (logisches oder)

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

Selektion - Punkt-/Bereichsanfragen

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

- **Linear Search (Sequenzielle Suche):**
 - **Aufwändig:** Jedes Tupel (jede Zeile) der Relation muss gelesen und geprüft werden.
 - **Funktioniert aber immer:** Unabhängig davon, ob die Daten sortiert sind oder Indizes existieren.
- **Binary Search (Binäre Suche):**
 - **Nur wenn die Datei entsprechend sortiert ist:** Setzt voraus, dass die Daten physisch nach dem Suchkriterium sortiert sind.
- **Primary Hash Index:**
 - **Single Record Retrieval** – Funktioniert sehr effizient für den direkten Zugriff auf einzelne Datensätze über einen exakten Schlüsselwert (Gleichheitsanfragen).
 - **Funktioniert nicht für Bereichsanfragen:** Da Hash-Funktionen keine Ordnung herstellen.
- **Primary/Clustering Index:**
 - Mehrere Records für jeden Wert.

- Pointer zum Block mit dem ersten Record (Daten sind physisch nach Index sortiert).
- **Vorteil:** Ideal für Bereichsanfragen, da zugehörige Datenblöcke sequenziell gelesen werden können.
- **Secondary Index:**
 - Jeder Record hat einen eigenen Pointer.
 - **Kann teuer werden:** Insbesondere bei Bereichsanfragen, die viele Tupel zurückgeben, da dies zu vielen *zufälligen* E/A-Zugriffen auf der Festplatte führen kann.

Strategien für konjunktive Anfragen

Selektion (Access Paths)

- Beispiel einer SQL-Anfrage mit konjunktiven Bedingungen:

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

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

Optimierung von konjunktiven Anfragen

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

Strategien für konjunktive Anfragen (Fortsetzung)

- Existierende Indizes verwenden:
 - **Idealfall:** Es existiert ein Index, der alle Attribute abdeckt, die in der Anfrage benötigt werden.
 - **Falls es mehrere Indizes gibt:**

- Der Datenbankoptimierer wählt den Index, der am selektivsten ist (d.h. der die Anzahl der potenziellen Ergebnisse am stärksten reduziert).
- Die verbleibenden Bedingungen werden anschließend auf die durch den Index gefilterten Ergebnisse angewendet (ausgewertet).
- **Überschneidung von Pointern ausnutzen (Index Merge / Intersection):**
 - Diese Strategie wird angewendet, wenn mehrere Indizes auf verschiedene Attribute in einer konjunktiven Anfrage anwendbar sind.
 - **Schritte:**
 1. **Index Lookups:** Für jede Bedingung, für die ein Index existiert, werden die relevanten Pointer (Verweise auf die Datenblöcke/Tupel) ermittelt.
 2. **Überschneidung der Pointer:** Die Schnittmenge (Konjunktion) dieser Pointer wird gebildet. Das bedeutet, es werden nur die Pointer behalten, die in *allen* relevanten Index-Lookups vorkommen. (Also nur die Tupel, die *alle* Bedingungen erfüllen.)
 3. **Records/Tupel lesen:** Basierend auf den ermittelten, überschneidenden Pointern werden die zugehörigen Records/Tupel aus der Tabelle gelesen.

Disjunktive Anfragen (mit OR-Verknüpfungen)

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

Wichtigkeit von Tuning

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

Join Algorithmen

Algorithmen

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

Strategien basieren auf Blöcken (nicht Tupeln) als Basis

- Bei der Kostenabschätzung von Join-Algorithmen geht man davon aus, dass Daten in Blöcken (Seiten) und nicht in einzelnen Tupeln gelesen werden.
- Schätze I/Os (Block Retrievals):** Die Kosten eines Join-Algorithmus werden primär durch die Anzahl der benötigten Ein-/Ausgabeoperationen (I/Os), d.h. das Lesen von Blöcken vom Speicher (z.B. Festplatte) in den Hauptspeicher, bestimmt.
- Benutze Puffer (Buffer) im Hauptspeicher:** Ein Puffer im Hauptspeicher wird genutzt, um die gelesenen Blöcke temporär zu speichern und die Anzahl der teuren Plattenzugriffe zu minimieren.

Tabellengröße und Join-Selektivität bestimmen die Kosten

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

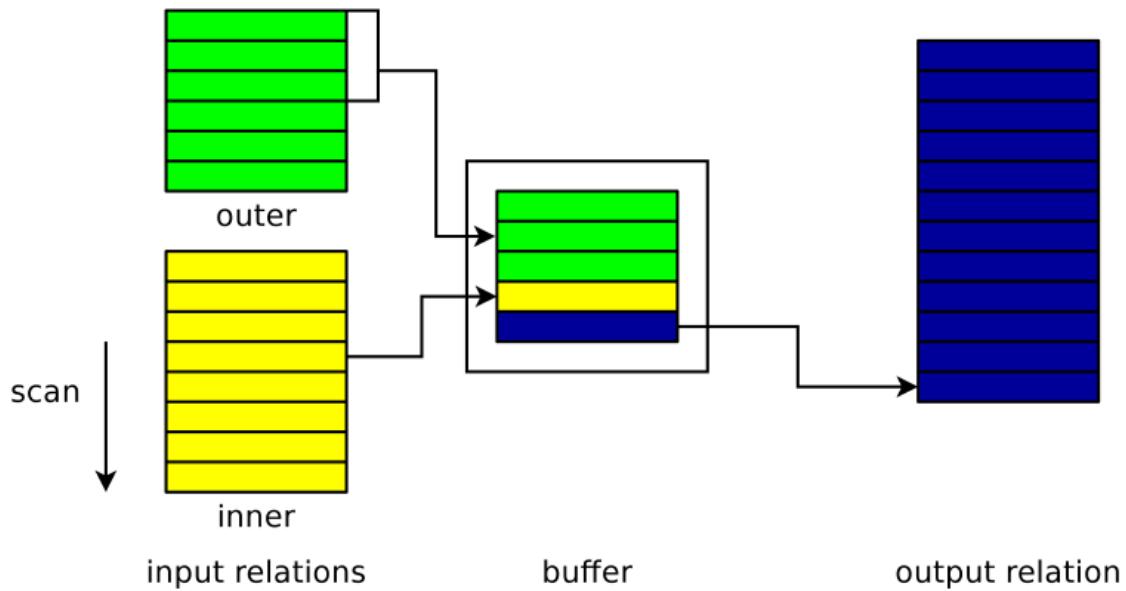
$$sel = \frac{\#\text{tuples in result}}{\#\text{candidates}}$$

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

Nested Loop Join

Sehr umfangreiches Beispiel von [DBS-9, p.37](#) bis [DBS-9, p.37](#)

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



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

```

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

```

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

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

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

- **Berechnung der Berechnungszeit:**

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

Beispiel

- **Beispiel:** Join von `reserved` \bowtie `customer`

- **Anzahl der Blöcke:**

- $b_{\text{reserved}} = 2.000$ (Anzahl der Blöcke der Relation `reserved`)
- $b_{\text{customer}} = 10$ (Anzahl der Blöcke der Relation `customer`)

- **Größe des Buffers im Hauptspeicher:**

- $n_B = 6$ (Anzahl der Blöcke, die gleichzeitig im Hauptspeicher gehalten werden können)

- **Kostenschätzung (Block Transfers) - Wiederholung der Formel:**

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

- **Kostenberechnung für das Beispiel:**

- **1. Fall:** `reserved` als äußere Relation ($b_{\text{outer}} = b_{\text{reserved}}$, $b_{\text{inner}} = b_{\text{customer}}$)

- $n_B - 2 = 6 - 2 = 4$
- Kosten = $2.000 + (\lceil 2.000/4 \rceil) \cdot 10$
- Kosten = $2.000 + (500) \cdot 10$
- Kosten = $2.000 + 5.000 = 7.000$ Block-Transfers

- **2. Fall:** `customer` als äußere Relation ($b_{\text{outer}} = b_{\text{customer}}$, $b_{\text{inner}} = b_{\text{reserved}}$)

- $n_B - 2 = 6 - 2 = 4$
- Kosten = $10 + (\lceil 10/4 \rceil) \cdot 2.000$
- Kosten = $10 + (2.5) \cdot 2.000$
- Kosten = $10 + (3) \cdot 2.000$
- Kosten = $10 + 6.000 = 6.010$ Block-Transfers

- **Ergebnis des Beispiels:** Es ist effizienter, die kleinere Relation (`customer`) als äußere Relation zu wählen, um die Anzahl der Block-Transfers zu minimieren.

Index-based Nested Loop Join

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

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

Merge Join

Ausnutzen der sortierten Reihenfolge

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

Annahme:

Beide Input-Relationen sind sortiert

Umfangreiches Beispiel

In den Slides von [DBS-9, p.43|Seite 126](#) bis [DBS-9, p.43|Seite 136](#)

Kosten von Merge Join

Parameter

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

Kostenschätzung (Block Transfers)

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

$$b_1 + b_2$$

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

Erweiterungen (Zusätzliche Überlegungen)

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

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

Hash Join

ID	name		number	ID
10	Jim	×	100	23
13	Joe		110	10
14	Sue		120	15
15	Pete		130	23
21	Dave		140	23
23	Anne		150	13
			160	15
			170	21

emp phone

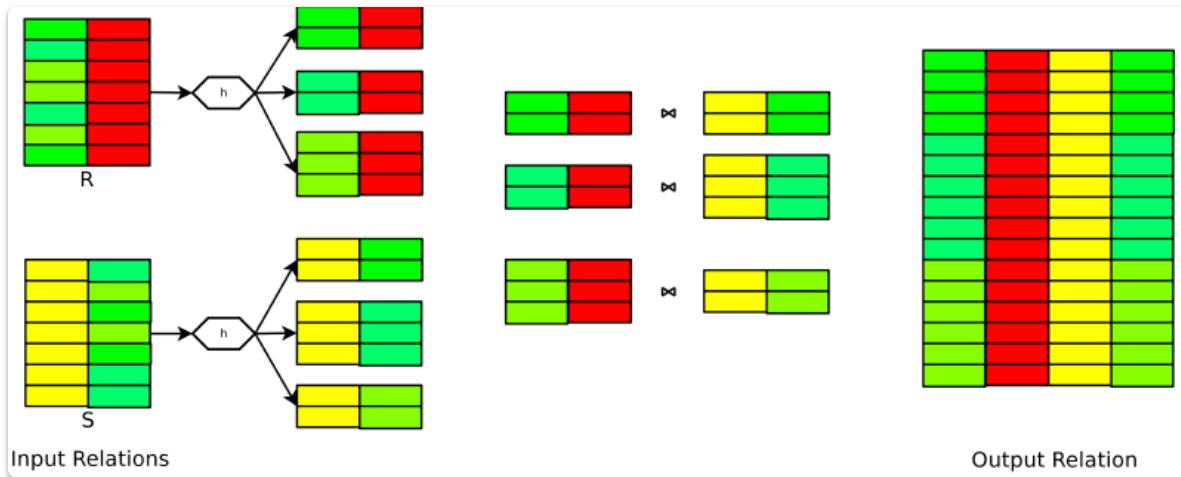
Wende Hashfunktion auf die Join-Attribute an
→ Partitioniere Tupel in Buckets

<table border="1"> <thead> <tr> <th>ID</th><th>name</th></tr> </thead> <tbody> <tr> <td>15</td><td>Pete</td></tr> <tr> <td>21</td><td>Dave</td></tr> </tbody> </table> <p style="text-align: center;">emp₀</p>	ID	name	15	Pete	21	Dave	<table border="1"> <thead> <tr> <th>number</th><th>ID</th></tr> </thead> <tbody> <tr> <td>120</td><td>15</td></tr> <tr> <td>160</td><td>15</td></tr> <tr> <td>170</td><td>21</td></tr> </tbody> </table> <p style="text-align: center;">phone₀</p>	number	ID	120	15	160	15	170	21	\times	$=$	<table border="1"> <thead> <tr> <th>ID</th><th>name</th><th>number</th></tr> </thead> <tbody> <tr> <td>15</td><td>Pete</td><td>120</td></tr> <tr> <td>15</td><td>Pete</td><td>160</td></tr> <tr> <td>21</td><td>Dave</td><td>170</td></tr> </tbody> </table> <p style="text-align: center;">result₀</p>	ID	name	number	15	Pete	120	15	Pete	160	21	Dave	170
ID	name																													
15	Pete																													
21	Dave																													
number	ID																													
120	15																													
160	15																													
170	21																													
ID	name	number																												
15	Pete	120																												
15	Pete	160																												
21	Dave	170																												
<table border="1"> <thead> <tr> <th>ID</th> <th>name</th> </tr> </thead> <tbody> <tr> <td>10</td> <td>Jim</td> </tr> <tr> <td>13</td> <td>Joe</td> </tr> </tbody> </table> <p style="text-align: center;">emp₁</p>	ID	name	10	Jim	13	Joe	<table border="1"> <thead> <tr> <th>number</th> <th>ID</th> </tr> </thead> <tbody> <tr> <td>110</td> <td>10</td> </tr> <tr> <td>150</td> <td>13</td> </tr> </tbody> </table> <p style="text-align: center;">phone₁</p>	number	ID	110	10	150	13	\times	$=$	<table border="1"> <thead> <tr> <th>ID</th> <th>name</th> <th>number</th> </tr> </thead> <tbody> <tr> <td>10</td> <td>Jim</td> <td>110</td> </tr> <tr> <td>13</td> <td>Joe</td> <td>150</td> </tr> </tbody> </table> <p style="text-align: center;">result₁</p>	ID	name	number	10	Jim	110	13	Joe	150					
ID	name																													
10	Jim																													
13	Joe																													
number	ID																													
110	10																													
150	13																													
ID	name	number																												
10	Jim	110																												
13	Joe	150																												
<table border="1"> <thead> <tr> <th>ID</th> <th>name</th> </tr> </thead> <tbody> <tr> <td>14</td> <td>Sue</td> </tr> <tr> <td>23</td> <td>Anne</td> </tr> </tbody> </table> <p style="text-align: center;">emp₂</p>	ID	name	14	Sue	23	Anne	<table border="1"> <thead> <tr> <th>number</th> <th>ID</th> </tr> </thead> <tbody> <tr> <td>100</td> <td>23</td> </tr> <tr> <td>130</td> <td>23</td> </tr> <tr> <td>140</td> <td>23</td> </tr> </tbody> </table> <p style="text-align: center;">phone₂</p>	number	ID	100	23	130	23	140	23	\times	$=$	<table border="1"> <thead> <tr> <th>ID</th> <th>name</th> <th>number</th> </tr> </thead> <tbody> <tr> <td>23</td> <td>Anne</td> <td>100</td> </tr> <tr> <td>23</td> <td>Anne</td> <td>130</td> </tr> <tr> <td>23</td> <td>Anne</td> <td>140</td> </tr> </tbody> </table> <p style="text-align: center;">result₂</p>	ID	name	number	23	Anne	100	23	Anne	130	23	Anne	140
ID	name																													
14	Sue																													
23	Anne																													
number	ID																													
100	23																													
130	23																													
140	23																													
ID	name	number																												
23	Anne	100																												
23	Anne	130																												
23	Anne	140																												

$$\text{result} = \text{result}_0 \cup \text{result}_1 \cup \text{result}_2$$

- **Jede Relation mit Hashfunktion partitionieren:**

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



Parameter

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

Schritte

1. Partitioniere Relation R_1 mit h_1 in Buckets r_1 :

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

2. Partitioniere Relation R_2 mit h_1 in Buckets r_2 :

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

3. Build-Phase:

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

4. Probe-Phase:

- Für jedes Bucket von r_2 :

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

Kostenschätzung (Block Transfers)

- Die Gesamtkosten für den Hash Join sind:

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

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

Kosten und Anwendung von Join-Algorithmen

Nested Loop Join

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

Merge Join

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

Hash Join

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

Zusammenfassung

- **Anfrageoptimierung ist eine Kernkomponente von relationalen DBMS.** (Entscheidend für die Leistungsfähigkeit einer Datenbank).
- **Heuristische Optimierung kann immer verwendet werden, kann aber zu suboptimalen Plänen führen.** (Schnelle, aber nicht immer die beste Lösung).
- **Kostenbasierte Optimierung ist auf Statistiken angewiesen.** (Genauere Optimierung durch Berücksichtigung von Datenverteilung und -größe).
- **Datenbanksysteme bieten Informationen zu Anfrageausführung an (EXPLAIN).** (Werkzeuge wie EXPLAIN ermöglichen es, den Ausführungsplan einer Abfrage zu analysieren und Engpässe zu identifizieren).
- **Datenbankadministratoren müssen stetig über Verbesserungen nachdenken (z.B. Indizes).** (Kontinuierliches Monitoring und Tuning sind notwendig, um die Performance aufrechtzuerhalten).