Verteilte Datenbanken

- **Motivation:**

geographisch verteilte Organisationsform einer Bank mit ihren Filialen

- **Filialen sollen Daten lokaler Kunden bearbeiten können**
- **Zentrale soll Zugriff auf alle Daten haben (z.B. für Kontogutschreibungen)**

Terminologie

Sammlung von Informationseinheiten, verteilt auf mehreren Rechnern, verbunden mittels Kommunikationsnetz → nach *Ceri & Pelagatti* (1984)

Kooperation zwischen autonom arbeitenden Stationen, zur Durchführung einer globalen Aufgabe

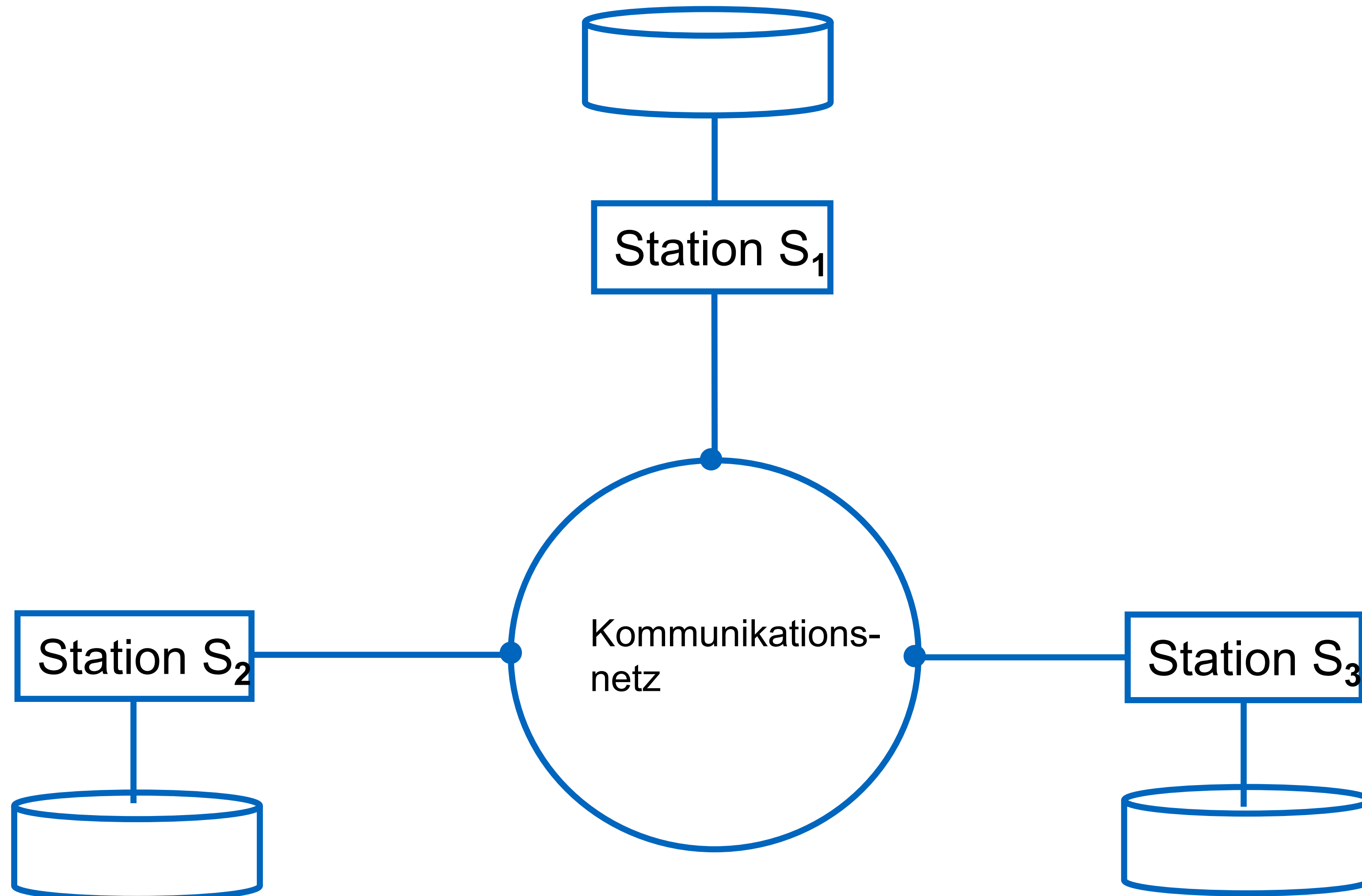
Kommunikationsmedien

LAN: local area network, z.B. Ethernet, Token-Ring oder FDDI-Netz

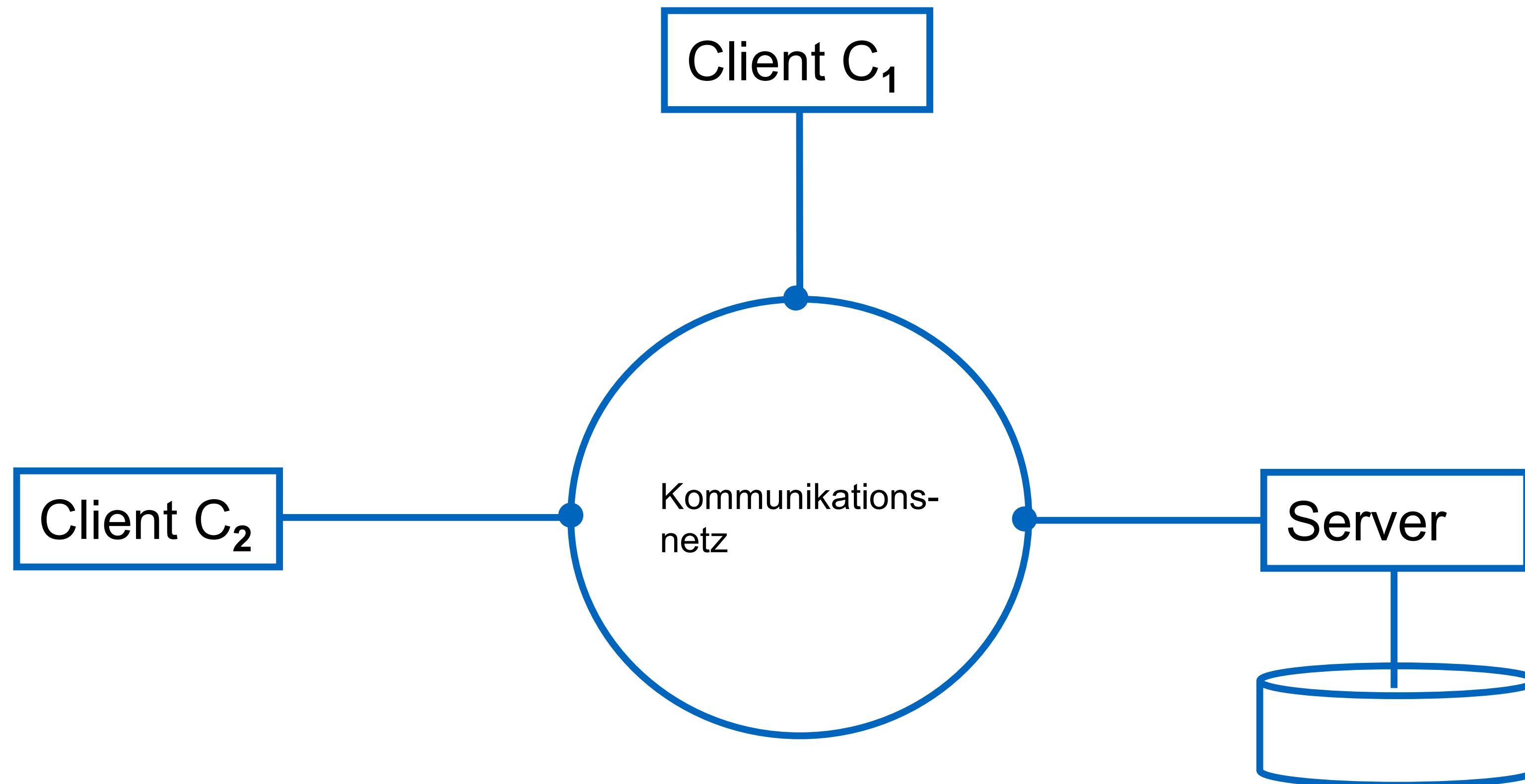
WAN: wide area network, z.B. das Internet

Telefonverbindungen, z.B. ISDN oder einfache Modem-Verbindungen

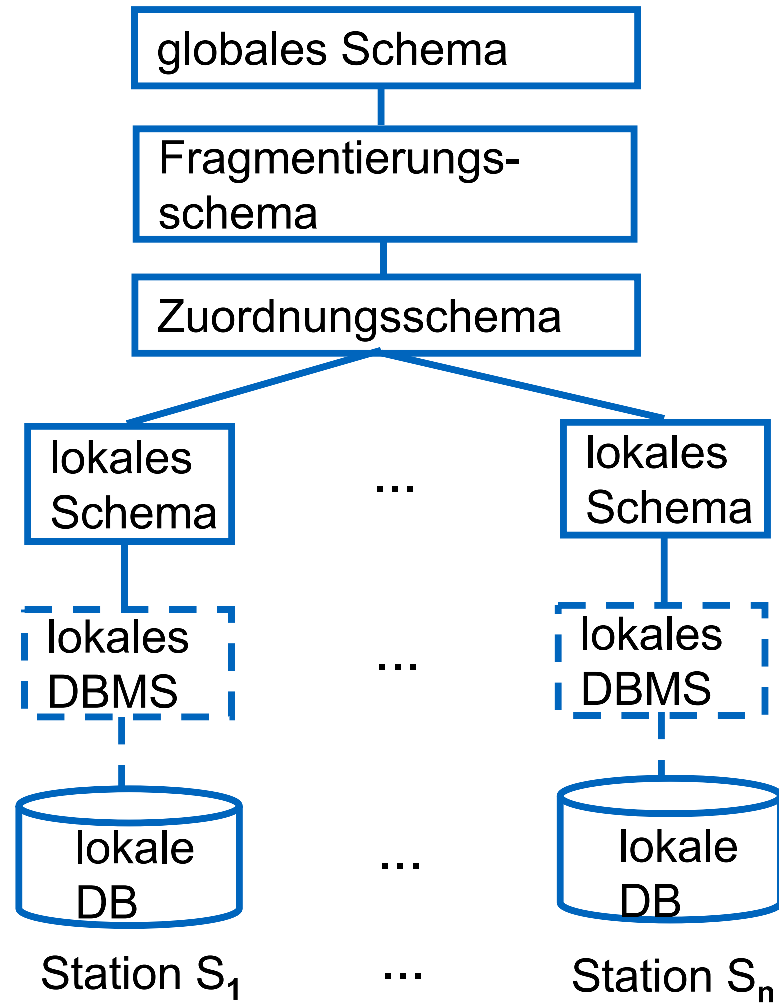
Verteiltes Datenbanksystem



Client-Server-Architektur



Aufbau und Entwurf eines verteilten Datenbanksystems

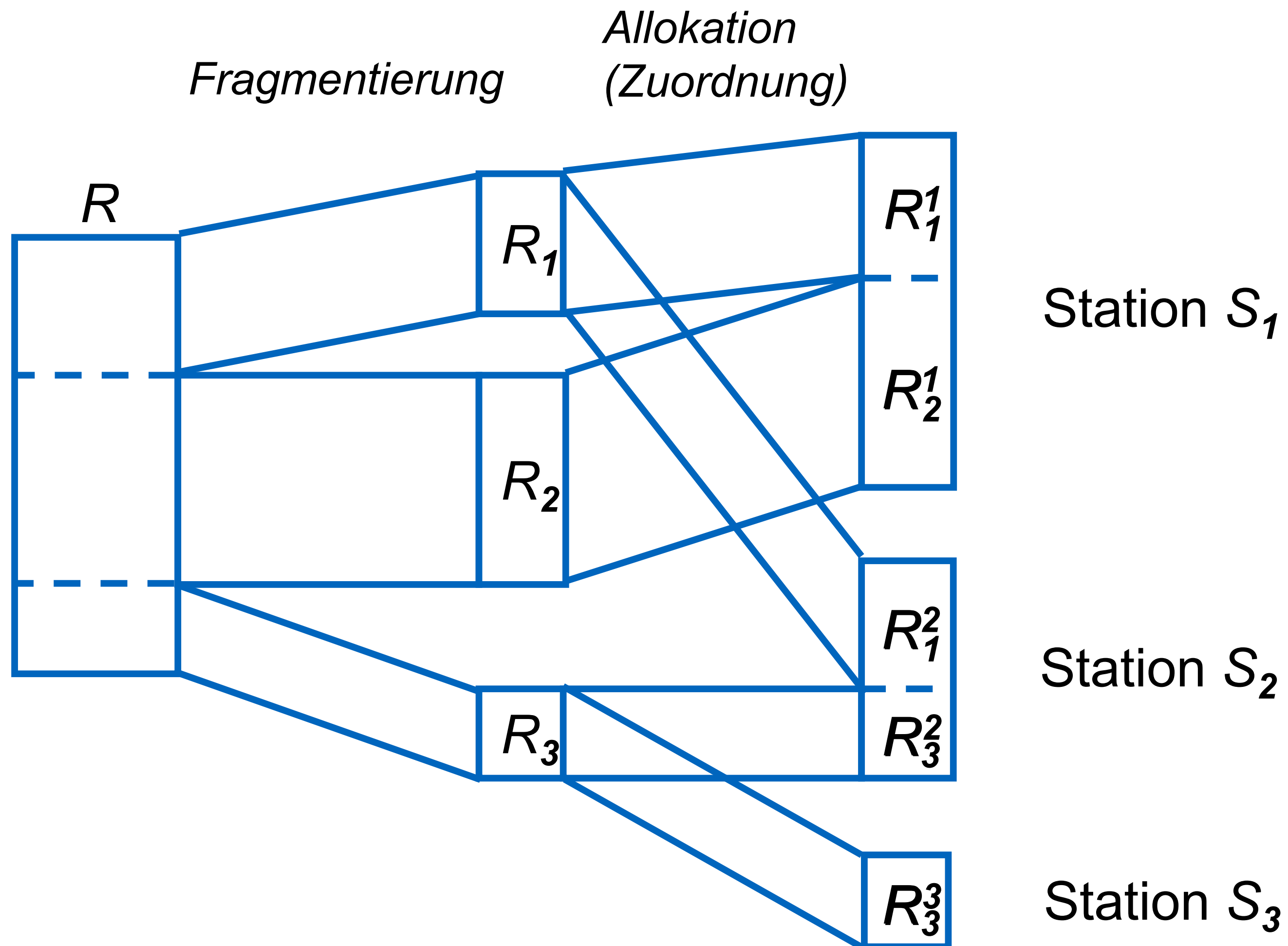


Fragmentierung und Allokation einer Relation

Fragmentierung: Fragmente enthalten Daten mit gleichem Zugriffsverhalten

Allokation: Fragmente werden den Stationen zugeordnet

- mit Replikation (redundanzfrei)
- ohne Replikation


Allokation
(Zuordnung)
Fragmentierung
R
R_1
R_2
R_3
R_1^1
R_2^1
R_1^2
R_3^2
R_3^3
Station S_1
Station S_2
Station S_3

Fragmentierung

horizontale Fragmentierung: Zerlegung der Relation in disjunkte Tupelmengen

vertikale Fragmentierung: Zusammenfassung von Attributen mit gleichem Zugriffsmuster

kombinierte Fragmentierung: Anwendung horizontaler und vertikaler Fragmentierung auf dieselbe Relation

Korrektheits-Anforderungen

Rekonstruierbarkeit

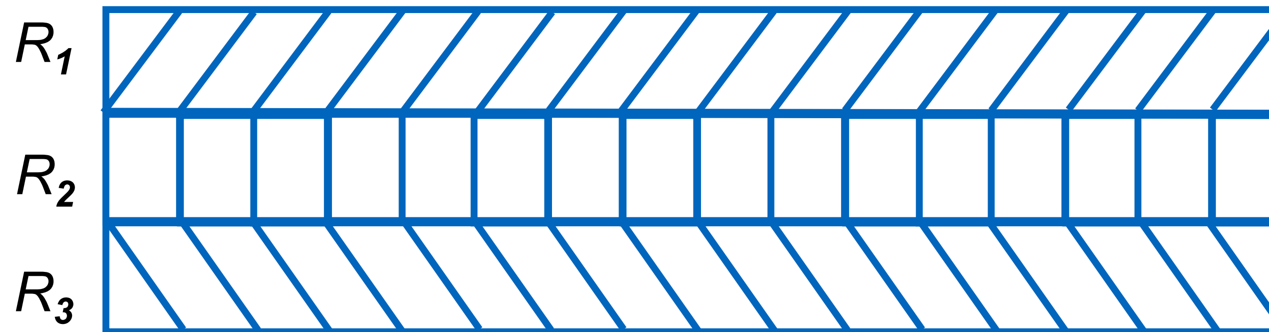
Vollständigkeit

Disjunktheit

Beispielrelation Professoren

Professoren						
PersNr	**Name**	**Rang**	**Raum**	**Fakultät**	**Gehalt**	**Steuerklasse**
2125	Sokrates	C4	226	Philosophie	85000	1
2126	Russel	C4	232	Philosophie	80000	3
2127	Kopernikus	C3	310	Physik	65000	5
2133	Popper	C3	52	Philosophie	68000	1
2134	Augustinus	C3	309	Theologie	55000	5
2136	Curie	C4	36	Physik	95000	3
2137	Kant	C4	7	Philosophie	98000	1

abstrakte Darstellung: *R*



Für 2 Prädikate p_1 und p_2 ergeben sich 4 Zerlegungen:

$$R1 := \sigma_{p_1 \wedge p_2}(R)$$
$$R2 := \sigma_{p_1 \wedge \neg p_2}(R)$$
$$R3 := \sigma_{\neg p_1 \wedge p_2}(R)$$
$$R4 := \sigma_{\neg p_1 \wedge \neg p_2}(R)$$

n Zerlegungsprädikate $p_1,\ldots,p_n$ ergeben 2^n Fragmente

sinnvolle Gruppierung der Professoren nach Fakultätszugehörigkeit:

⇨ 3 Zerlegungsprädikate:

p_1 ≡ Fakultät = ‚Theologie‛

p_2 ≡ Fakultät = ‚Physik‛

p_3 ≡ Fakultät = ‚Philosophie‛

TheolProfs´ $:= \sigma_{p1 \wedge \neg p2 \wedge \neg p3}$(Professoren) = σ_{p1}(Professoren)

PhysikProfs´ $:= \sigma_{\neg p1 \wedge p2 \wedge \neg p3}$(Professoren) = σ_{p2}(Professoren)

PhiloProfs´ $:= \sigma_{\neg p1 \wedge \neg p2 \wedge p3}$(Professoren) = σ_{p3}(Professoren)

AndereProfs´ $:= \sigma_{\neg p1 \wedge \neg p2 \wedge \neg p3}$(Professoren)

Abgeleitete horizontale Fragmentierung

Beispiel *Vorlesungen* aus dem Universitätsschema:
Zerlegung in Gruppen mit gleicher SWS-Zahl

2SWSVorls := $\sigma_{\mathbf{SWS=2}}$ (Vorlesungen)

3SWSVorls := $\sigma_{\mathbf{SWS=3}}$ (Vorlesungen)

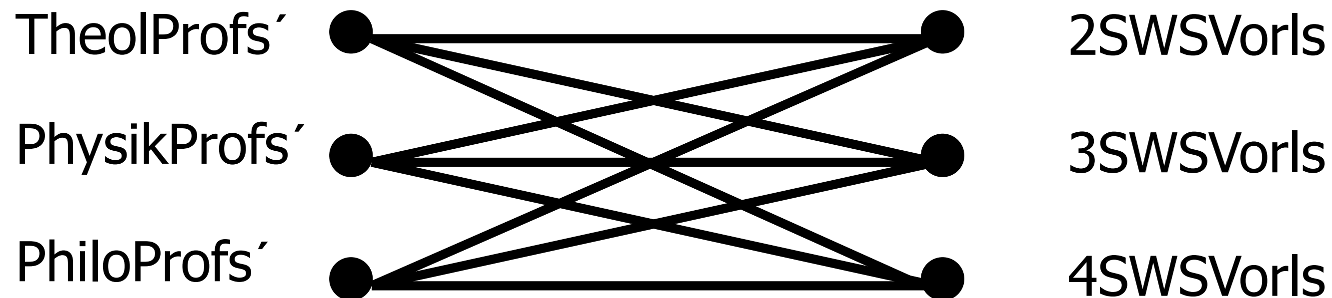
4SWSVorls := $\sigma_{\mathbf{SWS=4}}$ (Vorlesungen)

```
select Titel, Name
from Vorlesungen, Professoren
where gelesenVon = PersNr;
```

resultiert in:

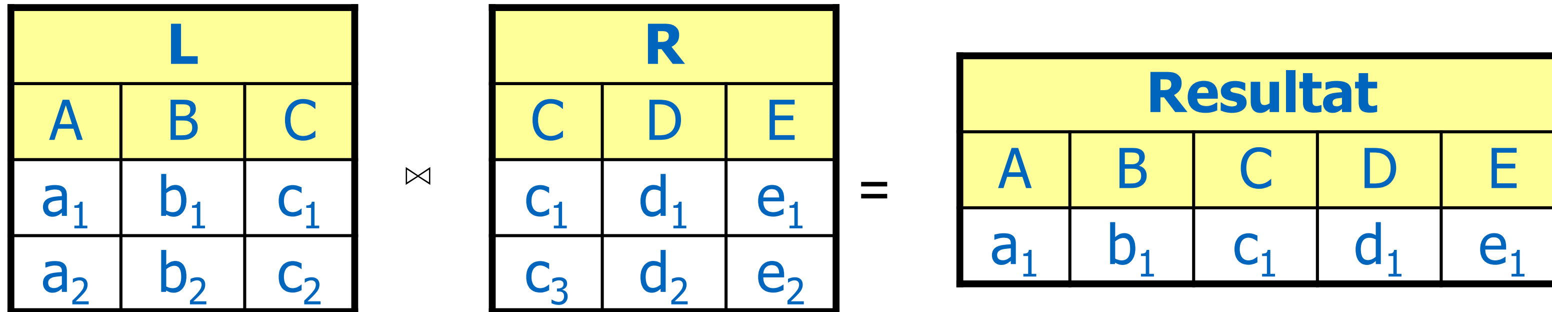
$$\Pi_{\text{Titel, Name}}((\text{TheolProfs}' \bowtie \text{2SWSVorls}) \cup (\text{TheolProfs}' \bowtie \text{3SWSVorls}) \cup \ldots \cup (\text{PhiloProfs}' \bowtie \text{4SWSVorls}))$$

Join-Graph zu diesem Problem:



Andere Join-Arten

- natürlicher Join



L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

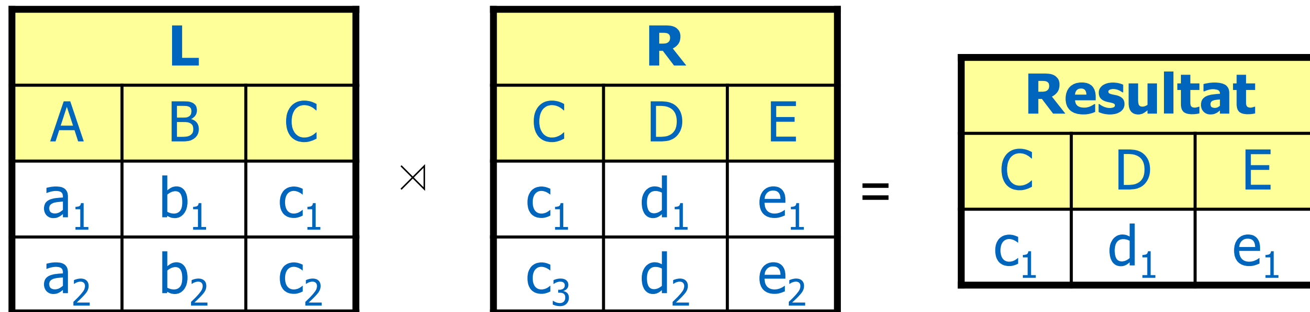
R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

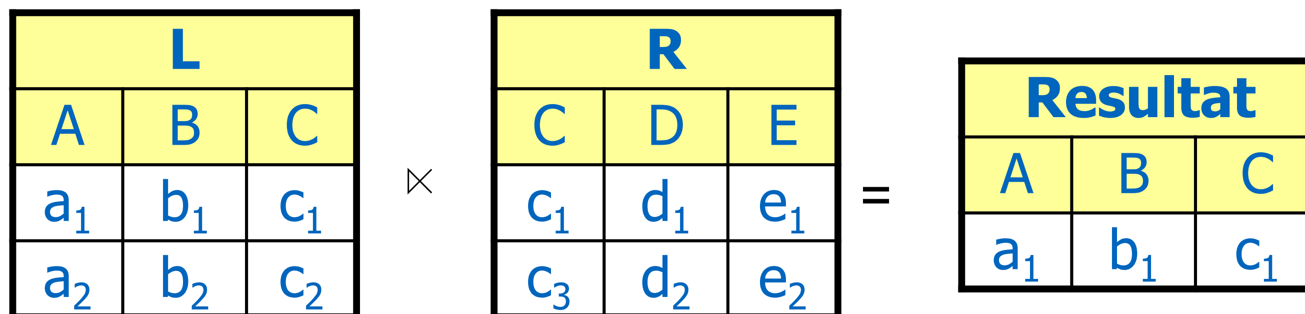
Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

Semi-Joins

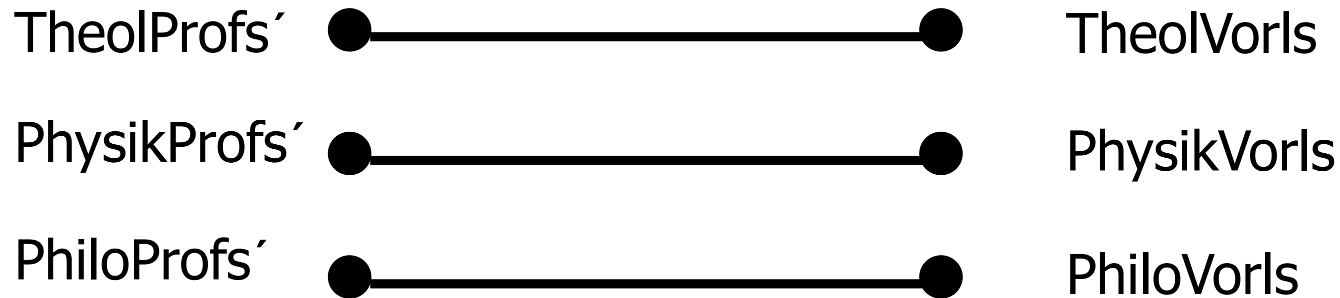
- Semi-Join von R mit L



- Semi-Join von L mit R



Lösung: abgeleitete Fragmentierung



TheolVorls := Vorlesungen $\ltimes_{\mathbf{gelesenVon=PersNr}}$ TheolProfs´

PhysikVorls := Vorlesungen $\ltimes_{\mathbf{gelesenVon=PersNr}}$ PhysikProfs´

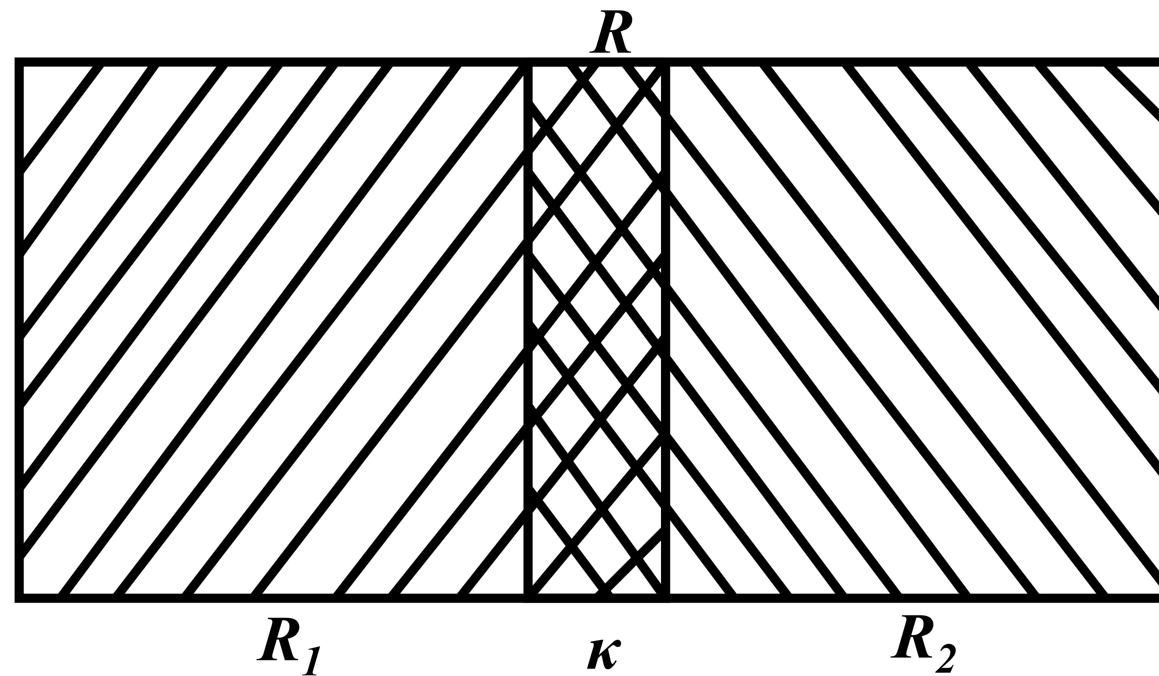
PhiloVorls := Vorlesungen $\ltimes_{\mathbf{gelesenVon=PersNr}}$ PhiloProfs´

$\Pi_{\mathbf{Titel,\ Name}}$((TheolProfs´ $\bowtie_p$ TheolVorls) $\cup$
(PhysikProfs´ $\bowtie_p$ PhysikVorls) $\cup$
(PhiloProfs´ $\bowtie_p$ PhiloVorls))

mit $p \equiv$ (PersNr = gelesenVon)

Vertikale Fragmentierung

abstrakte Darstellung:



Vertikale Fragmentierung

Beliebige vertikale Fragmentierung gewährleistet ***keine Rekonstruierbarkeit***

2 mögliche Ansätze, um Rekonstruierbarkeit zu garantieren:

- jedes Fragment enthält den Primärschlüssel der Originalrelation. Aber: Verletzung der *Disjunktheit*

- jedem Tupel der Originalrelation wird ein eindeutiges ***Surrogat*** (= künstlich erzeugter Objektindikator) zugeordnet, welches in jedes vertikale Fragment des Tupels mit aufgenommen wird

Vertikale Fragmentierung (Beispiel)

für die Universitätsverwaltung sind PersNr, Name, Gehalt und Steuerklasse interessant:

ProfVerw := $\Pi_{\text{PersNr, Name, Gehalt, Steuerklasse}}$ (Professoren)

für Lehre und Forschung sind dagegen PersNr, Name, Rang, Raum und Fakultät von Bedeutung:

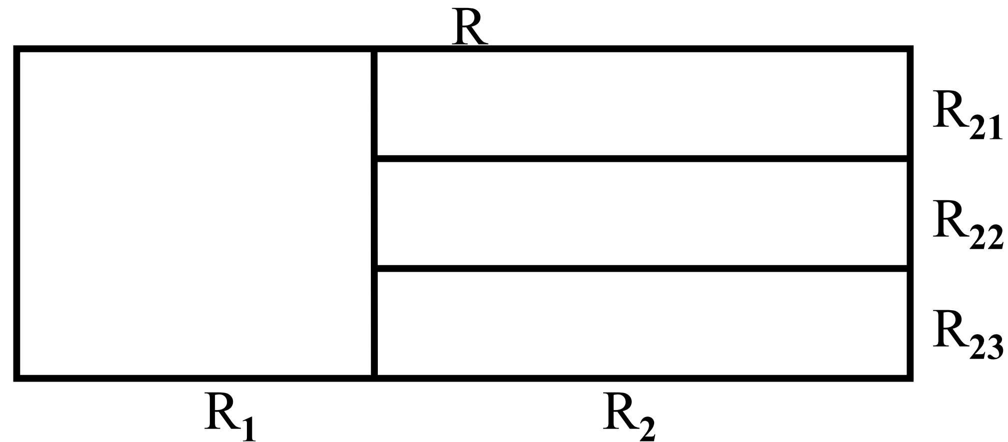
Profs := $\Pi_{\text{PersNr, Name, Rang, Raum, Fakultät}}$ (Professoren)

Rekonstruktion der Originalrelation *Professoren*:

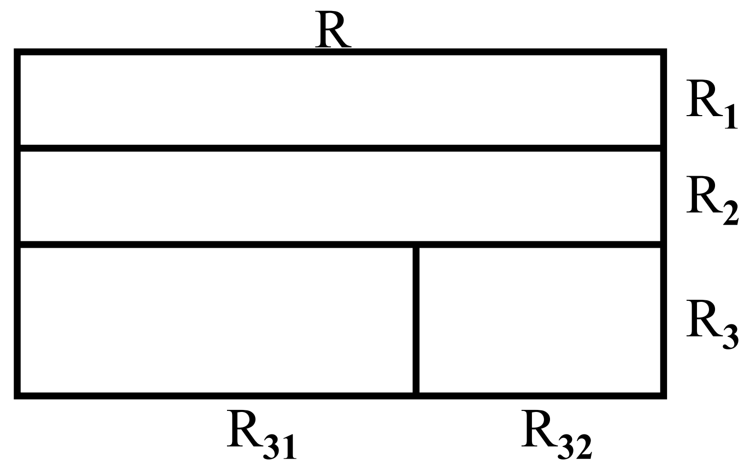
Professoren = ProfVerw $\bowtie_{\text{ProfVerw.PersNr = Profs.PersNr}}$ Profs

Kombinierte Fragmentierung

a) horizontale Fragmentierung nach vertikaler Fragmentierung



b) vertikale Fragmentierung nach horizontaler Fragmentierung



Rekonstruktion nach kombinierter Fragmentierung

TUM

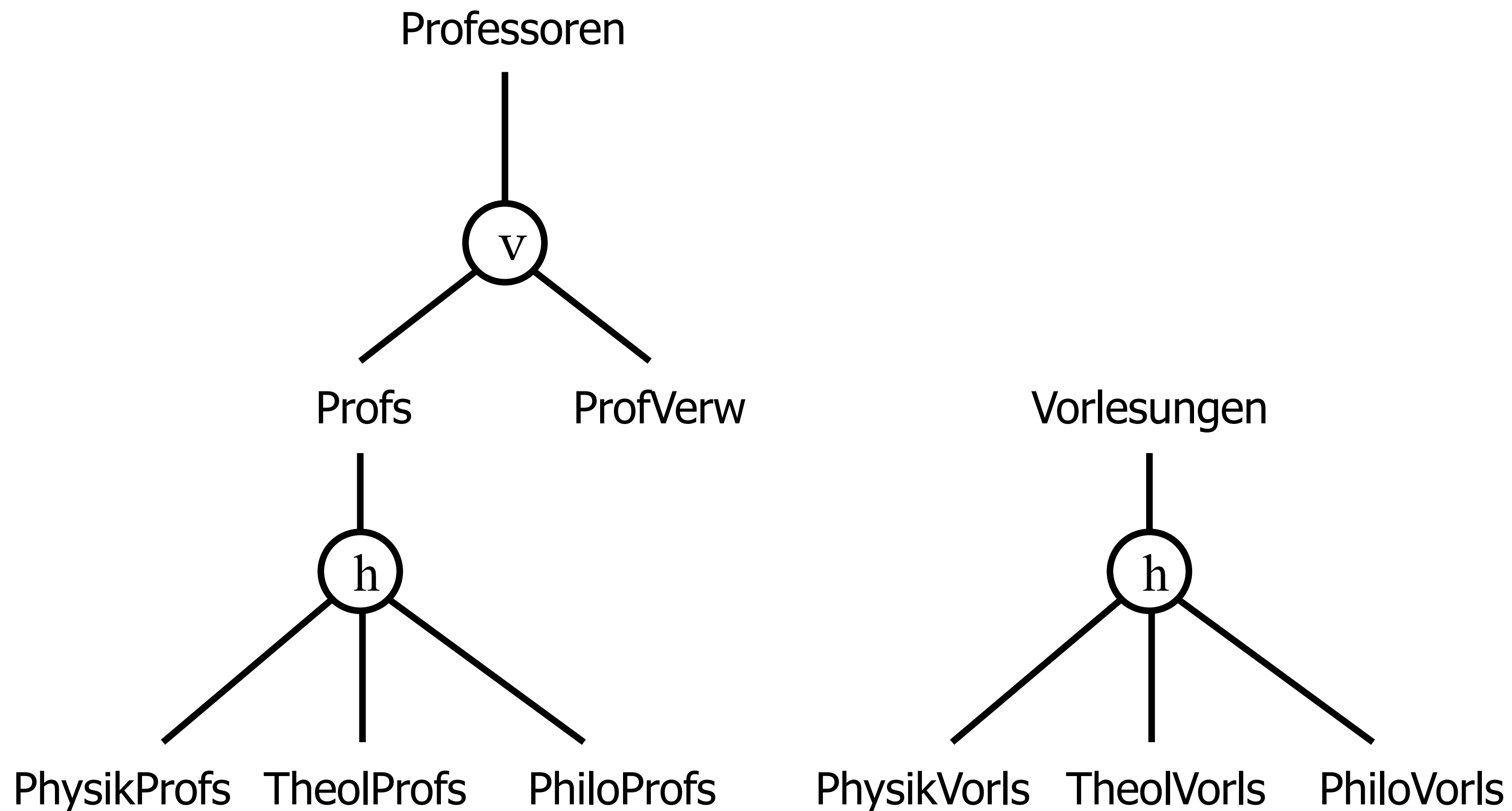
Fall a)

$$R = R_1 \ltimes_p (R_{21} \cup R_{22} \cup R_{23})$$

Fall b)

$$R = R_1 \cup R_2 \cup (R_{31} \bowtie_{R_{31}.K = R_{32}.K} R_{32})$$

Baumdarstellung der Fragmentierungen (Beispiel)



Allokation

Dasselbe Fragment kann mehreren Stationen zugeordnet werden

Allokation für unser Beispiel ohne Replikationen ⇒ **redundanzfreie** Zuordnung

Station	Bemerkung	zugeordnete Fragemente
$S_{\mathbf{Verw}}$	Verwaltungsrechner	*{ProfVerw}*
$S_{\mathbf{Physik}}$	Dekanat Physik	*{PhysikVorls, PhysikProfs}*
$S_{\mathbf{Philo}}$	Dekanat Philosophie	*{PhiloVorls, PhiloProfs}*
$S_{\mathbf{Theol}}$	Dekanat Theologie	*{TheolVorls, TheolProfs}*

Transparenz in verteilten Datenbanken

Grad der Unabhängigkeit den ein VDBMS dem Benutzer beim Zugriff auf verteilte Daten vermittelt

verschiedene Stufen der Transparenz:

- Fragmentierungstransparenz
- Allokationstransparenz
- Lokale Schema-Transparenz

Fragmentierungstranparenz

Beispielanfrage, die Fragmentierungstransparenz voraussetzt:

select Titel, Name
from Vorlesungen, Professoren
where gelesenVon = PersNr;

Beispiel für eine Änderungsoperation, die Fragmentierungs-transparenz voraussetzt:

update Professoren
set Fakultät = ‚Theologie'
where Name = ‚Sokrates';

Fortsetzung Beispiel

Ändern des Attributwertes von *Fakultät*

- Transferieren des Sokrates-Tupels aus Fragment *PhiloProfs* in das Fragment *TheolProfs* (= Löschen aus *PhiloProfs*, Einfügen in *TheolProfs*)

- Ändern der abgeleiteten Fragmentierung von *Vorlesungen* (= Einfügen der von Sokrates gehaltenen Vorlesungen in *TheolVorls*, Löschen der von ihm gehaltenen Vorlesungen aus *PhiloVorls*)

Allokationstransparenz

Benutzer müssen Fragmentierung kennen, aber nicht den „Aufenthaltsort" eines Fragments

Beispielanfrage:

select Gehalt
from ProfVerw
where Name = ‚Sokrates';

Allokationstransparenz (Forts.)

unter Umständen muss Originalrelation rekonstruiert werden

Beispiel:
Verwaltung möchte wissen, wieviel die C4-Professoren der Theologie insgesamt verdienen

da Fragmentierungstransparenz fehlt muss die Anfrage folgendermaßen formuliert werden:

select sum (Gehalt)
from ProfVerw, TheolProfs
where ProfVerw.PersNr = TheolProfs.PersNr **and**
Rang = ‚C4‘;

Lokale Schema-Transparenz

Der Benutzer muss auch noch den Rechner kennen, auf dem ein Fragment liegt.

Beispielanfrage:

select Name
from TheolProfs **at** S_{Theol}
where Rang = ‚C3‘;

Lokale Schema-Transparenz (Forts.)

Ist überhaupt Transparenz gegeben?

Lokale Schema-Transparenz setzt voraus, dass alle Rechner dasselbe Datenmodell und dieselbe Anfragesprache verwenden.
⇒ vorherige Anfrage kann somit analog auch an Station S_{Philo} ausgeführt werden
Dies ist nicht möglich bei Kopplung unterschiedlicher DBMS.

Verwendung grundsätzlich verschiedener Datenmodelle auf lokalen DBMS nennt man „Multi-Database-Systems“ (oft unumgänglich in „realer“ Welt).

Anfrageübersetzung und Anfrageoptimierung

Voraussetzung: Fragmentierungstransparenz

Aufgabe des Anfrageübersetzers: Generierung eines Anfrageauswertungsplans auf den Fragmenten

Aufgabe des Anfrageoptimierers: Generierung eines möglichst effizienten Auswertungsplanes

→ abhängig von der Allokation der Fragmente auf den verschiedenen Stationen des Rechnernetzes

Anfragebearbeitung bei horizontaler Fragmentierung

Übersetzung einer SQL-Anfrage auf dem globalen Schema in eine äquivalente Anfrage auf den Fragmenten benötigt 2 Schritte:

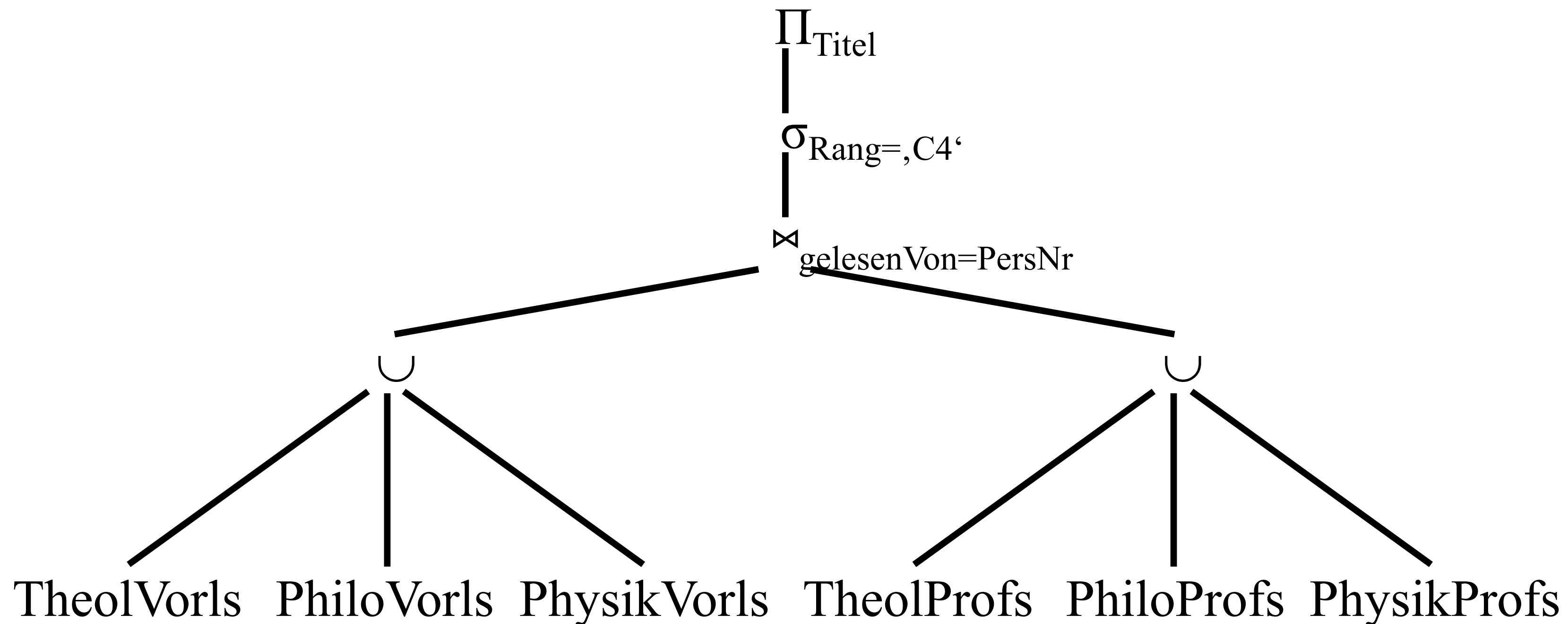
1. Rekonstruktion aller in der Anfrage vorkommenden globalen Relationen aus den Fragmenten, in die sie während der Fragmentierungsphase zerlegt wurden. Hierfür erhält man einen algebraischen Ausdruck.

2. Kombination des Rekonstruktionsausdrucks mit dem algebraischen Anfrageausdruck, der sich aus der Übersetzung der SQL-Anfrage ergibt.

Beispiel

select Titel
from Vorlesungen, Profs
where gelesenVon = PersNr **and**
Rang = ‚C4‘;

Der entstandene algebraische Ausdruck heißt kanonische Form der Anfrage:



Algebraische Äquivalenzen

Für eine effizientere Abarbeitung der Anfrage benutzt der Anfrageoptimierer die folgende Eigenschaft:

$$(R_1 \cup R_2) \bowtie_p (S_1 \cup S_2) =$$
$$(R_1 \bowtie_p S_1) \cup (R_1 \bowtie_p S_2) \cup (R_2 \bowtie_p S_1) \cup (R_2 \bowtie_p S_2)$$

Die Verallgemeinerung auf *n* horizontale Fragmente $R_1,\ldots,R_n$ von *R* und *m* Fragmente $S_1,\ldots,S_m$ von *S* ergibt:

$$(R_1 \cup \ldots \cup R_n) \bowtie_p (S_1 \cup \ldots \cup S_m) = \bigcup_{1 \le i \le n} \bigcup_{1 \le j \le m} (R_i \bowtie_p S_j)$$

Falls gilt: $S_i = S \ltimes_p R_i$ mit $S = S_1 \cup \ldots \cup S_n$

Dann gilt immer:

$$R \bowtie_p S = \bigcup_{1 \le i \le m} (R_i \bowtie_p S_i)$$

Algebraische Äquivalenzen (Forts.)

Für eine derartig abgeleitete horizontale Fragmentierung von S gilt somit:

$$(R_1 \cup \ldots \cup R_n) \bowtie_p (S_1 \cup \ldots \cup S_m) =$$

$$(R_1 \bowtie_p S_1) \cup (R_2 \bowtie_p S_2) \cup \ldots \cup (R_n \bowtie_p S_n)$$

Algebraische Äquivalenzen (Forts.)

Für eine derartig abgeleitete horizontale Fragmentierung von S gilt somit:

$$(R_1 \cup \ldots \cup R_n) \bowtie_p (S_1 \cup \ldots \cup S_m) =$$

$$(R_1 \bowtie_p S_1) \cup (R_2 \bowtie_p S_2) \cup \ldots \cup (R_n \bowtie_p S_n)$$

Für unser Beispiel gilt nun folgendes:

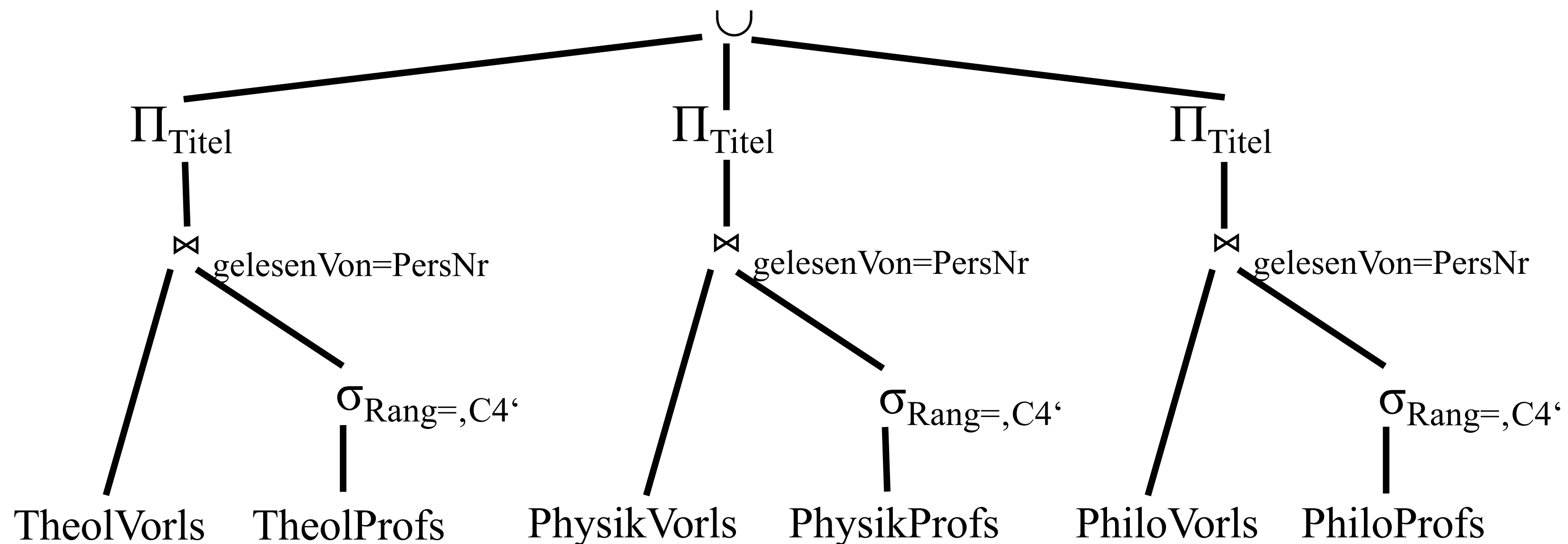
(TheolVorls ∪ PhysikVorls ∪ PhiloVorls) $\bowtie_{\ldots}$
(TheolProfs ∪ PhysikProfs ∪ PhiloProfs)

Um Selektionen und Projektionen über den Vereinigungs-operator hinweg „nach unten zu drücken" benötigt man folgende Regeln:

- $\sigma_p(R_1 \cup R_2) = \sigma_p(R_1) \cup \sigma_p(R_2)$
- $\Pi_L(R_1 \cup R_2) = \Pi_L(R_1) \cup \Pi_L(R_2)$

Optimale Form der Anfrage

Die Anwendung dieser algebraischen Regeln generiert den folgenden Auswertungsplan:



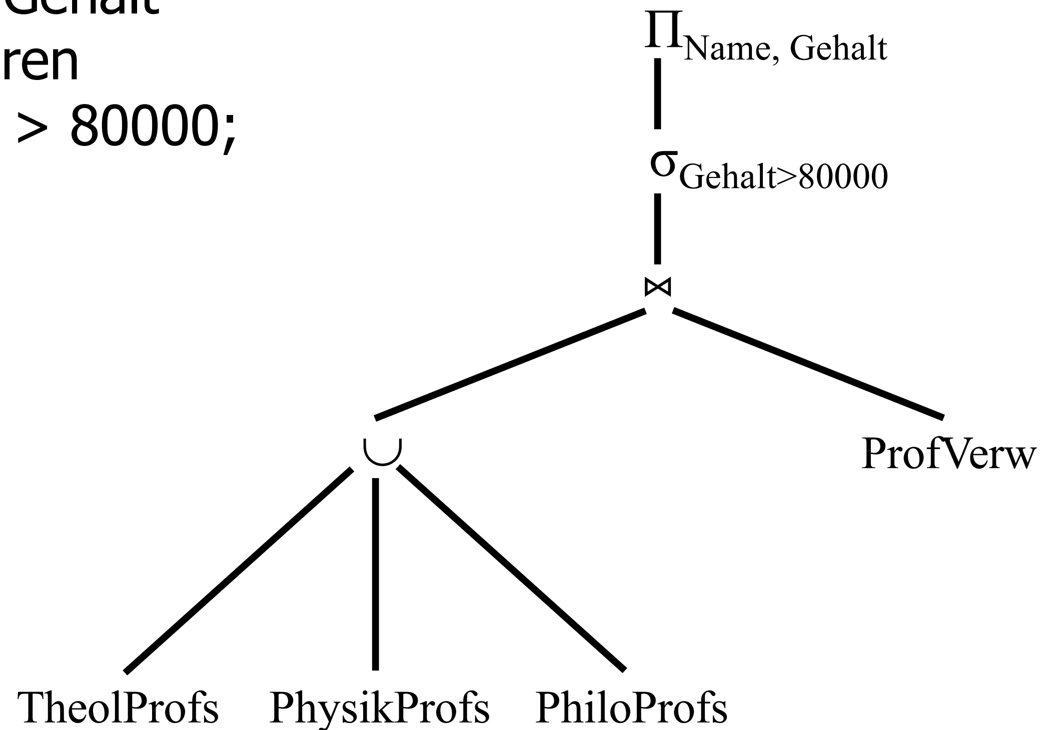
Auswertungen können lokal auf den Stationen S_{Theol}, S_{Physik} und S_{Philo} ausgeführt werden ⇒ Stationen können parallel abarbeiten und lokales Ergebnis voneinander unabhängig an die Station, die die abschliessende Vereinigung durchführt, übermitteln.

Anfragebearbeitung bei vertikaler Fragmentierung

Beispiel:

select Name, Gehalt
from Professoren
where Gehalt > 80000;

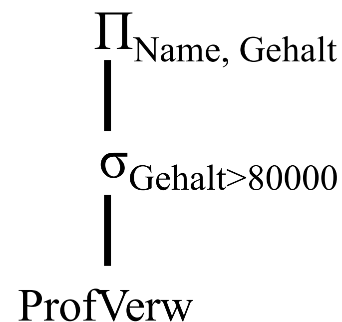
kanonischer Auswertungsplan:



Optimierung bei vertikaler Fragmentierung

Für unser Beispiel gilt:

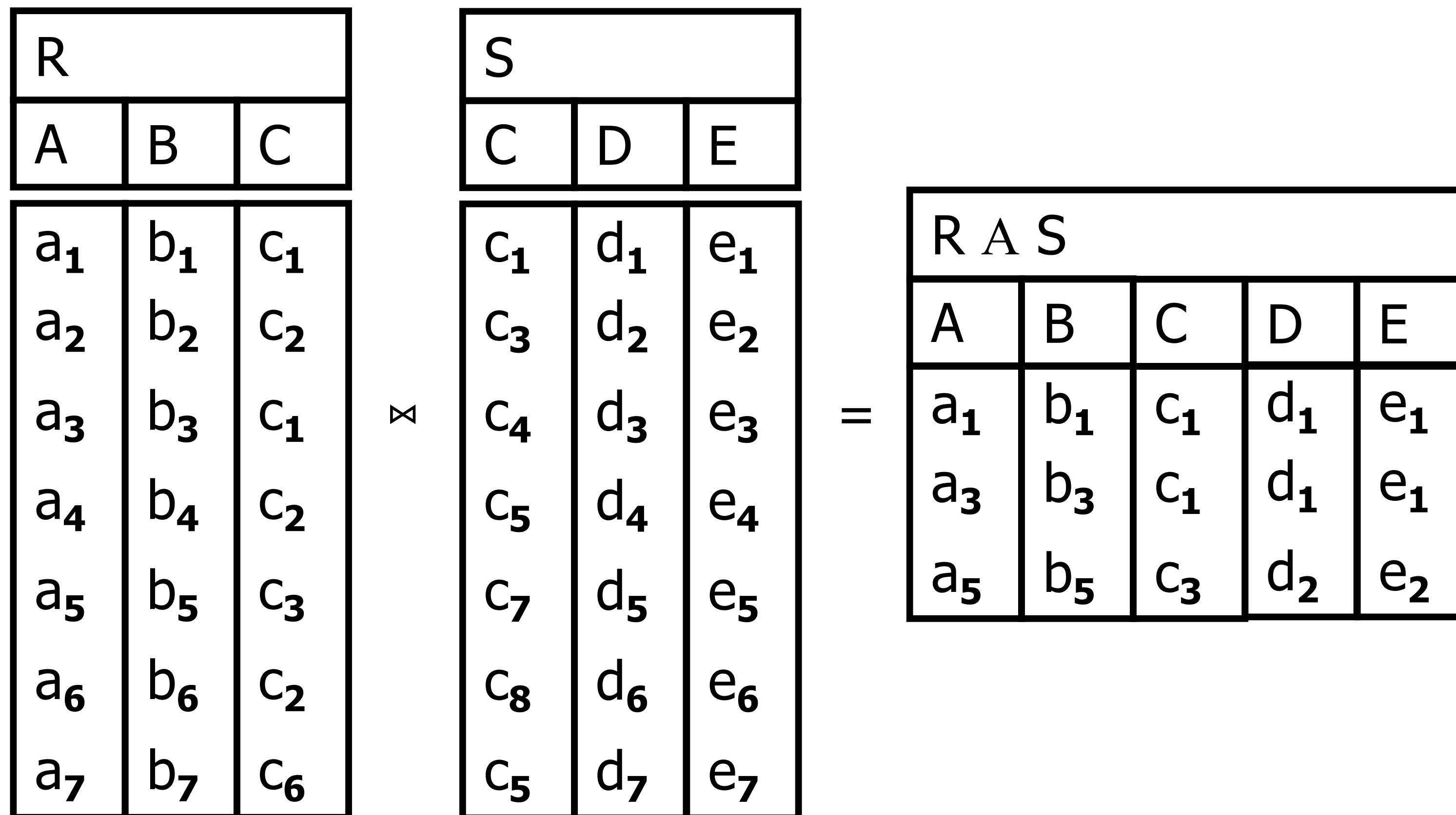
Alle notwendigen Informationen sind in *ProfVerw* enthalten ⇒ der Teil mit Vereinigung und Join kann „abgeschnitten" werden. Das ergibt den folgenden optimierten Auswertungsplan:



Beispiel für eine schlecht zu optimierende Anfrage:
(Attribut *Rang* fehlt in *ProfVerw*)

select Name, Gehalt, Rang
from Professoren
where Gehalt > 80000;

Der natürliche Verbund zweier Relationen R und S



R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

$\bowtie$

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

=

R A S

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

Join-Auswertung in VDBMS

spielt kritischere Rolle als in zentralisierten Datenbanken

Problem: Argumente eines Joins zweier Relationen können auf unterschiedlichen Stationen des VDBMS liegen

2 Möglichkeiten: Join-Auswertung mit und ohne Filterung

Join-Auswertung

Betrachtung des allgemeinsten Falles:

äußere Argumentrelation *R* ist auf Station St_R gespeichert

innere Argumentrelation *S* ist dem Knoten St_S zugeordnet

Ergebnis der Joinberechnung wird auf einem dritten Knoten St_{Result} benötigt

Join-Auswertung ohne Filterung

Nested-Loops

Transfer einer Argumentrelation

Transfer beider Argumentrelationen

Nested-Loops

Iteration durch die äußere Relation *R* mittels Laufvariable *r* und Anforderung des/der zu jedem Tupel *r* passenden Tupel *s* $\in S$ mit $r.C = s.C$ (über Kommunikationsnetz bei St_S)

Diese Vorgehensweise benötigt pro Tupel aus *R* eine Anforderung und eine passende Tupelmenge aus *S* (welche bei vielen Anforderungen leer sein könnte)
$\Rightarrow$ es werden **$2 * |R|$** Nachrichten benötigt

Der natürliche Verbund zweier Relationen R und S

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

TUM

Transfer einer Argumentrelation

1. vollständiger Transfer einer Argumentrelation (z.B. R) zum Knoten der anderen Argumentrelation

2. Ausnutzung eines möglicherweise auf *S.C* existierenden Indexes

Der natürliche Verbund zweier Relationen R und S

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Transfer beider Argumentrelationen

1. Transfer beider Argumentrelationen zum Rechner St_{Result}

2. Berechnung des Ergebnisses auf dem Knoten St_{Result} mittels
   a) Merge-Join (bei vorliegender Sortierung)
   oder
   b) Hash-Join (bei fehlender Sortierung)

   → evtl. Verlust der vorliegenden Indexe für die Join-Berechnung
   → kein Verlust der Sortierung der Argumentrelation(en)

$\rtimes$

Join-Auswertung mit Filterung

Verwendung des Semi-Join-Operators zur Filterung

Schlüsselidee: nur Transfer von Tupeln, die passenden Join-Partner haben

Benutzung der folgenden algebraischen Eigenschaften:

- $R \bowtie S = R \bowtie (R \rtimes S)$
- $R \rtimes S = \Pi_{c}(R) \rtimes S$

Join-Auswertung mit Filterung (Beispiel, Filterung der Relation *S*)

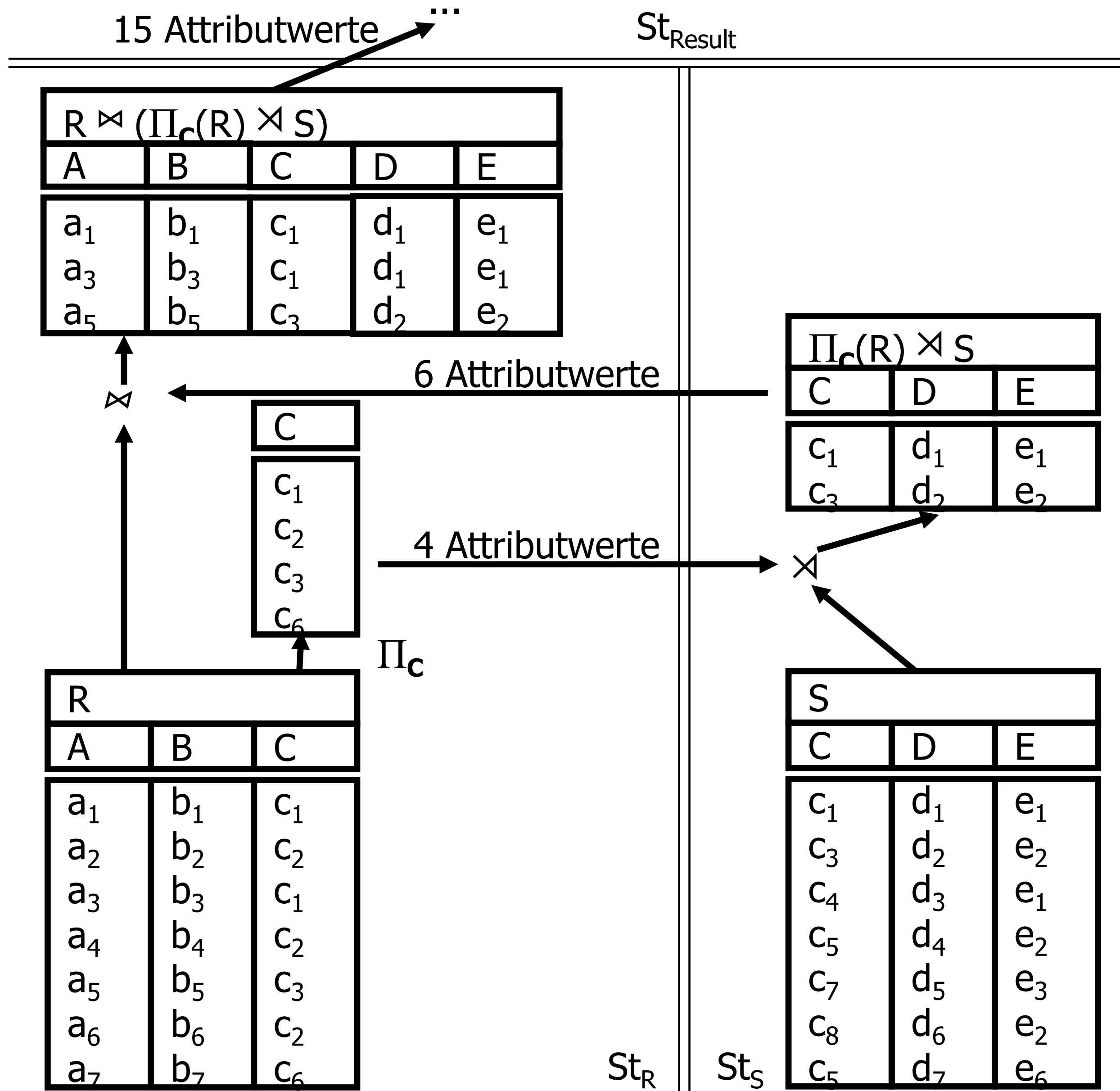
1. Transfer der unterschiedlichen *C*-Werte von *R* (= $\Pi_C(R)$) nach St_S
2. Auswertung des Semi-Joins $R \ltimes S = \Pi_C(R) \ltimes S$ auf St_S und Transfer nach St_R
3. Auswertung des Joins auf St_R, der nur diese transferierten Ergebnistupel des Semi-Joins braucht

Transferkosten werden nur reduziert, wenn gilt:

$\| \Pi_C(R) \| + \| R \ltimes S \| < \| S \|$

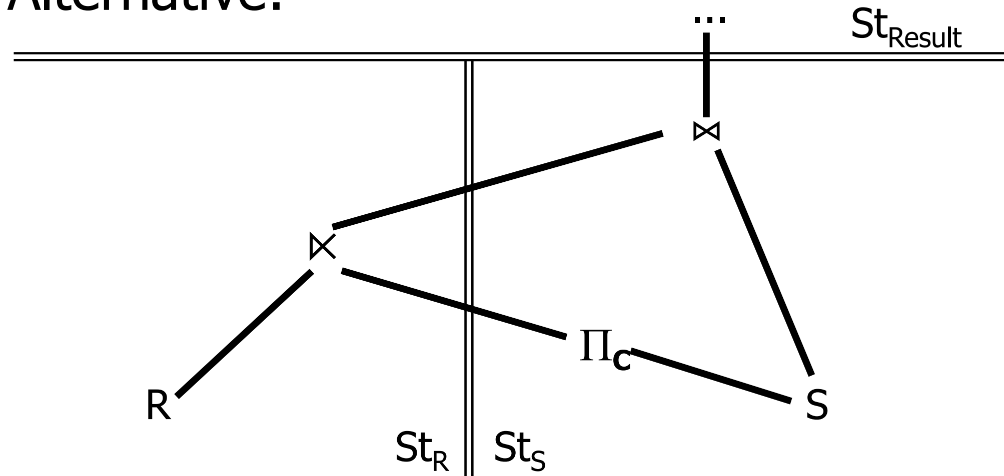
mit $\| R \|$ = Größe (in Byte) einer Relation

Auswertung des Joins $R \bowtie S$ mit Semi-Join-Filterung von S



Alternative Auswertungungspläne

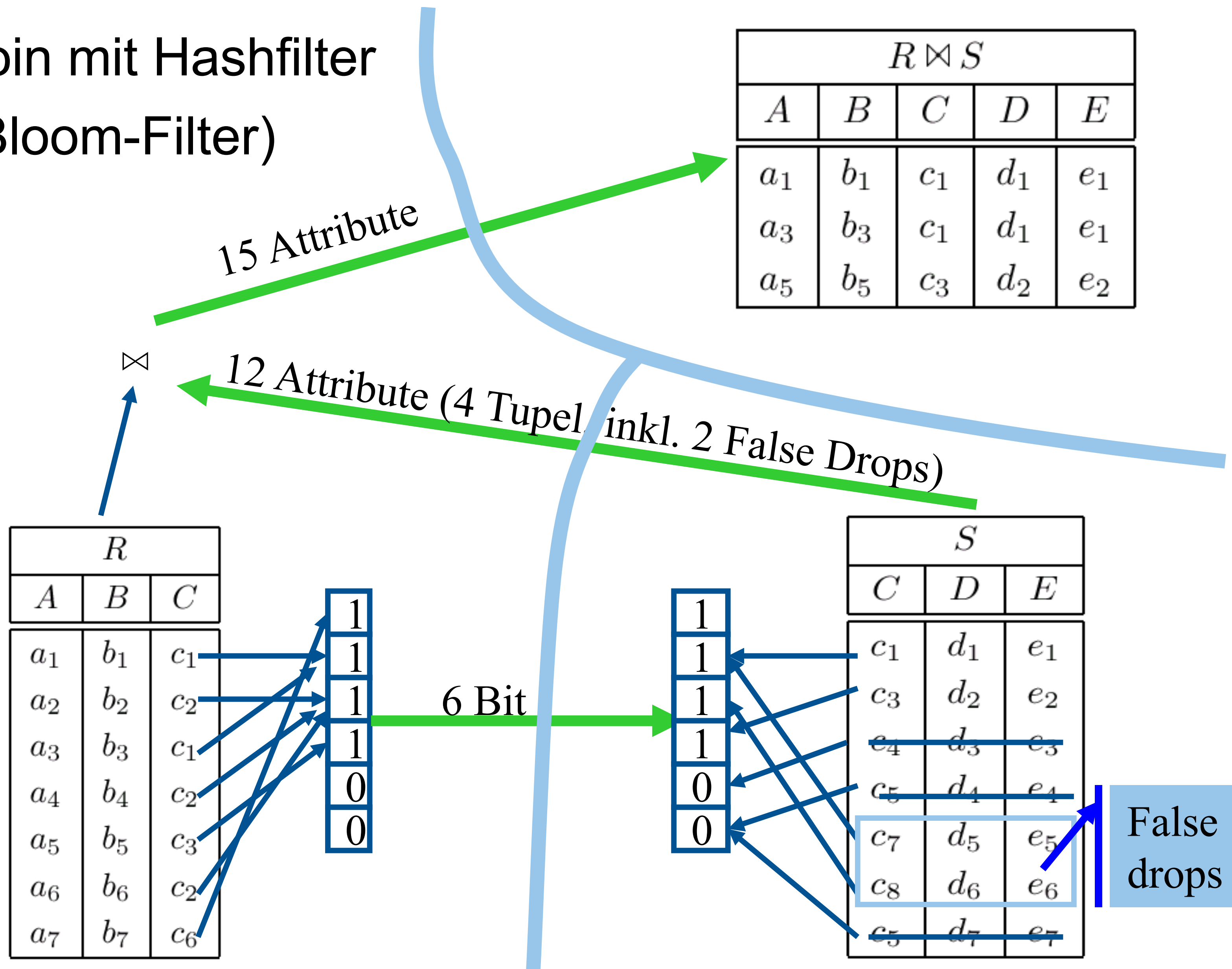
1. Alternative:



2. Alternative:

$(R \ltimes \Pi_c(S)) \bowtie (\Pi_c(R) \rtimes S)$

Join mit Hashfilter (Bloom-Filter)



$R \bowtie S$				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

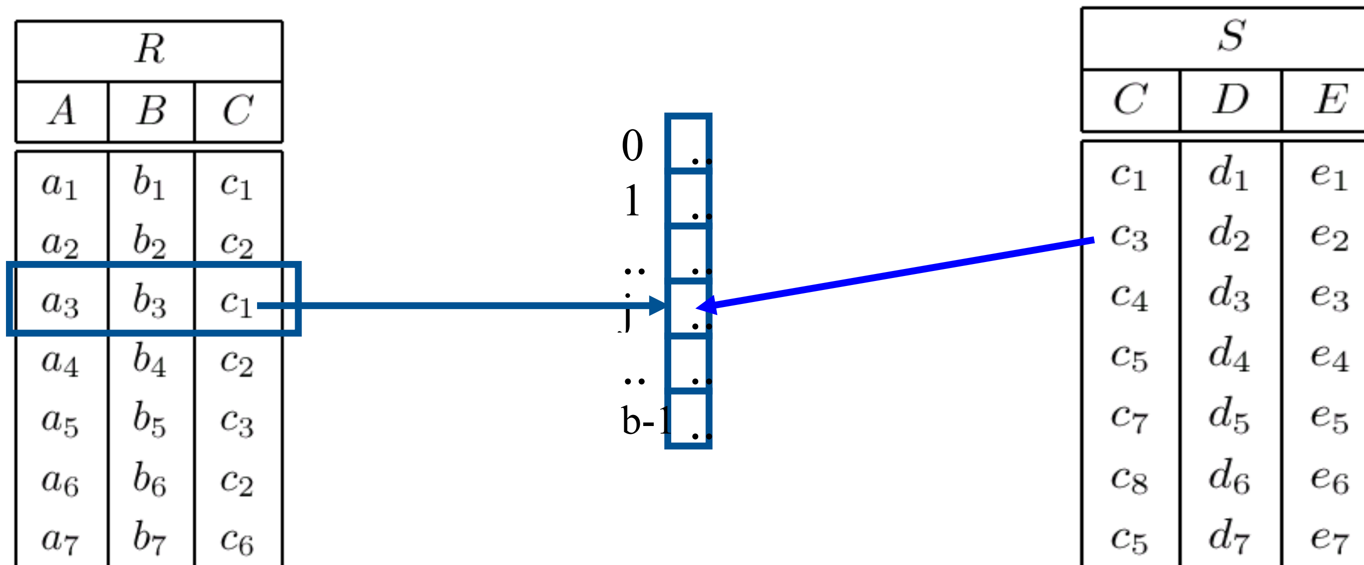
S		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Join mit Hashfilter

(False Drop Abschätzung)

Wahrscheinlichkeit, dass ein bestimmtes Bit j gesetzt ist

- W. dass ein bestimmtes $r \in R$ das Bit setzt: 1/b
- W. dasss kein $r \in R$ das Bit setzt: $(1-1/b)^{|R|}$
- W. dass ein $r \in R$ das Bit gesetzt hat: $1- (1-1/b)^{|R|}$



Join mit Hashfilter

(False Drop Abschätzung)

W. dass irgendein $r \in R$ ein bestimmtes Bit gesetzt hat: $1-(1-1/b)^{|R|}$

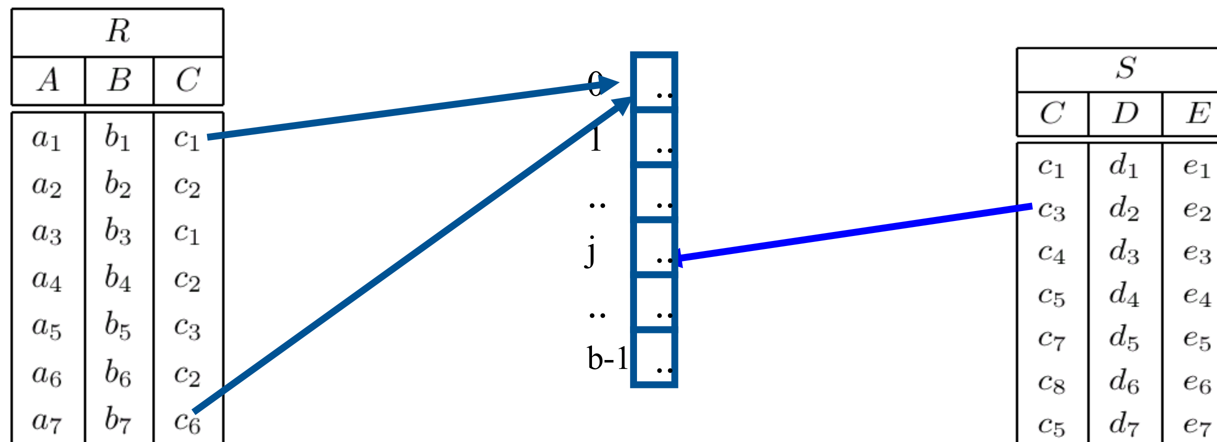
Wieviele Bits sind gesetzt?

- $b * [1-(1-1/b)^{|R|}]$

Mehrere $r \in R$ können dasselbe Bit setzen

Approximation: alle $r \in R$ setzen unterschiedliche Bits

- W. dass ein bestimmtes Bit j gesetzt ist: $|R| / b$
- $b >> |R|$



Join mit Hashfilter

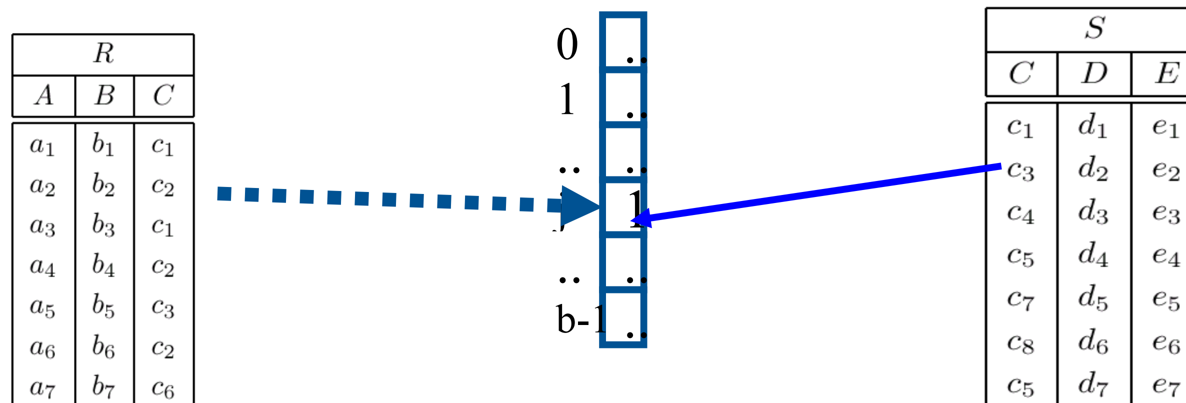
(False Drop Abschätzung)

W. dass irgendein r∈R ein bestimmtes Bit gesetzt hat:
- $1-(1-1/b)^{|R|}$

W. dass ein bestimmtes s∈S ausgewählt wird:
- $1-(1-1/b)^{|R|}$

Wieviele s∈S werden ausgewählt?
- $|S| * [1-(1-1/b)^{|R|}]$

Approximation: alle r setzen unterschiedliche Bits
- W. dass ein bestimmtes Bit j gesetzt ist: |R| / b
- |S|*(|R|/b) Elemente aus S werden ausgewählt



Parameter für die Kosten eines Auswertungsplan

Kardinalitäten von Argumentrelationen

Selektivitäten von Joins und Selektionen

Transferkosten für Datenkommunikation (Verbindungsaufbau + von Datenvolumen abhängiger Anteil für Transfer)

Auslastung der einzelnen VDBMS-Stationen

Effektive Anfrageoptimierung muss auf Basis eines Kosten-modells durchgeführt werden und soll mehrere Alternativen für unterschiedliche Auslastungen des VDBMS erzeugen.

Transaktionskontrolle in VDBMS

Transaktionen können sich bei VDBMS über mehrere Rechnerknoten erstrecken

⇒ Recovery:

- Redo: Wenn eine Station nach einem Fehler wieder anläuft, müssen alle Änderungen einmal abgeschlossener Transaktionen - seien sie lokal auf dieser Station oder global über mehrere Stationen ausgeführt worden - auf den an dieser Station abgelegten Daten wiederhergestellt werden.
- Undo: Die Änderungen noch nicht abgeschlossener lokaler und globaler Transaktionen müssen auf den an der abgestürzten Station vorliegenden Daten rückgängig gemacht werden.

EOT-Behandlung

Die EOT (End-of-Transaction)-Behandlung von globalen Transaktionen stellt in VDBMS ein Problem dar.

Eine globale Transaktion muss atomar beendet werden, d.h. entweder

- **commit**: globale Transaktion wird an allen (relevanten) lokalen Stationen festgeschrieben

oder

- **abort**: globale Transaktion wird gar nicht festgeschrieben

⇒ Problem in verteilter Umgebung, da die Stationen eines VDBMS unabhängig voneinander „abstürzen" können

TUM

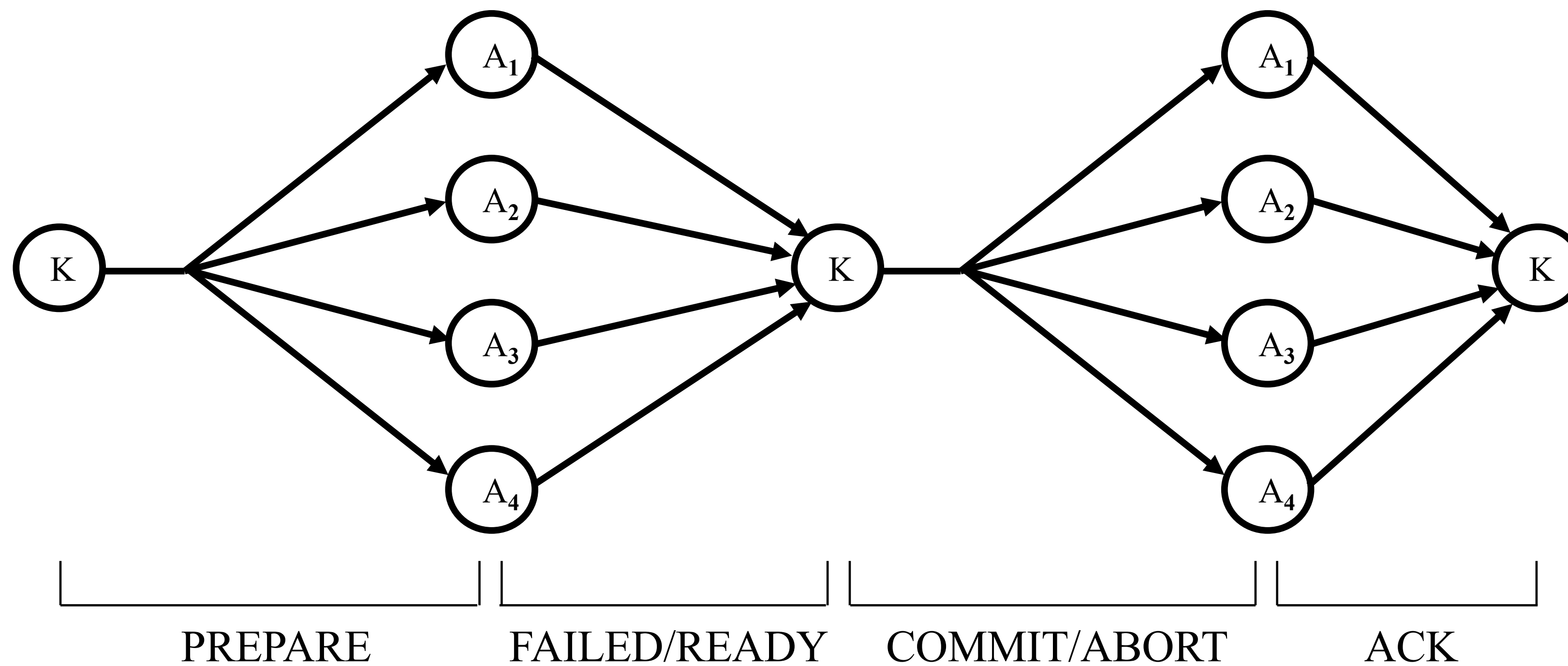
Problemlösung:

Zweiphasen-Commit-Protokoll

→ gewährleistet die Atomarität der EOT-Behandlung

das 2PC-Verfahren wird von sog. Koordinator *K* überwacht und gewährleistet, dass die *n* Agenten (=Stationen im VDBMS) $A_1, \ldots A_n$, die an einer Transaktion beteiligt waren, entweder alle von Transaktion *T* geänderten Daten festschreiben oder alle Änderungen von *T* rückgängig machen

Nachrichtenaustausch beim 2PC-Protokoll (für 4 Agenten)



Ablauf der EOT-Behandlung beim 2PC-Protokoll

K schickt allen Agenten eine PREPARE-Nachricht, um herauszufinden, ob sie Transaktionen festschreiben können

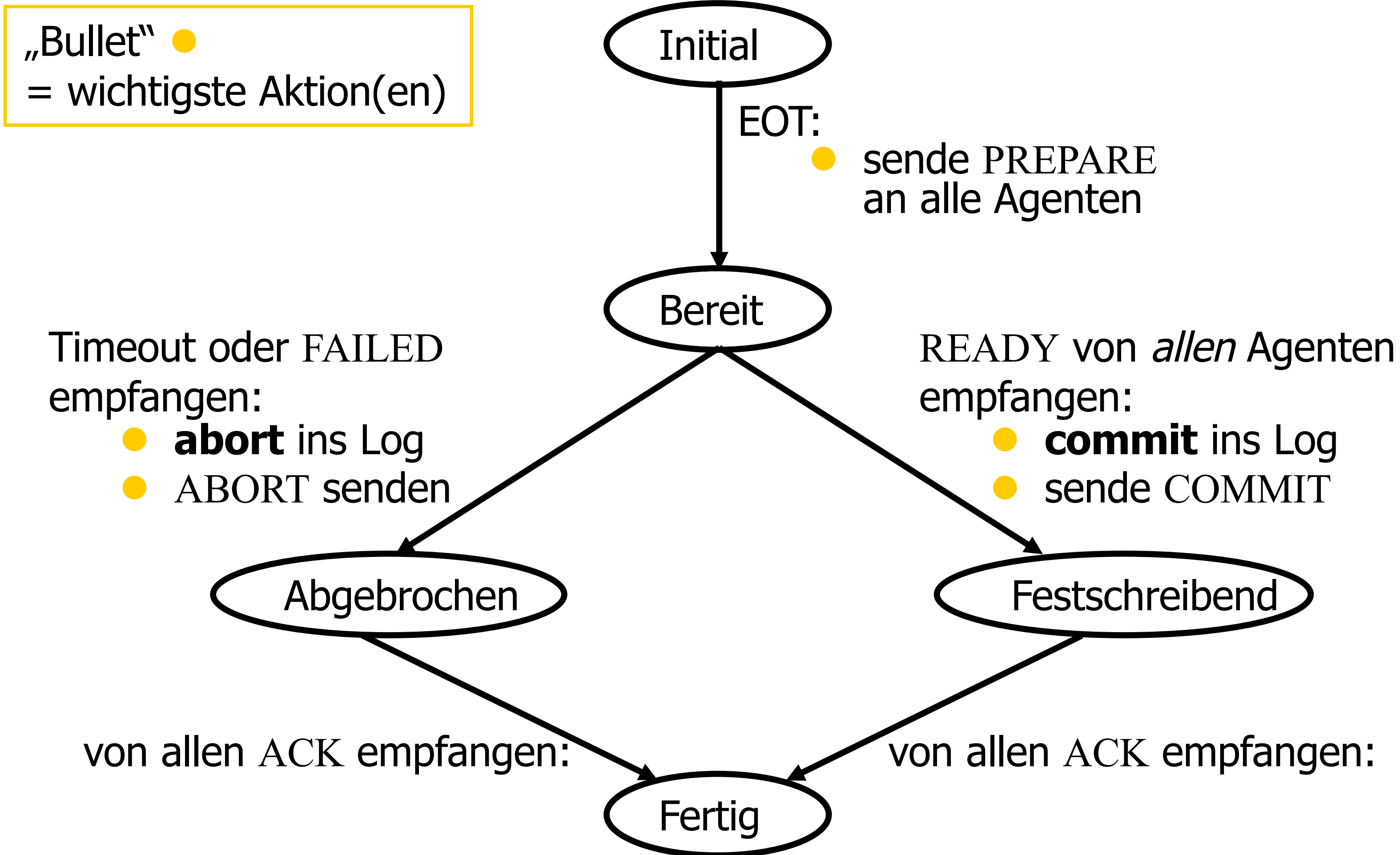
jeder Agent A_i empfängt PREPARE-Nachricht und schickt eine von zwei möglichen Nachrichten an *K*:

- READY, falls A_i in der Lage ist, die Transaktion *T* lokal festzuschreiben
- FAILED, falls A_i kein **commit** durchführen kann (wegen Fehler, Inkonsistenz etc.)

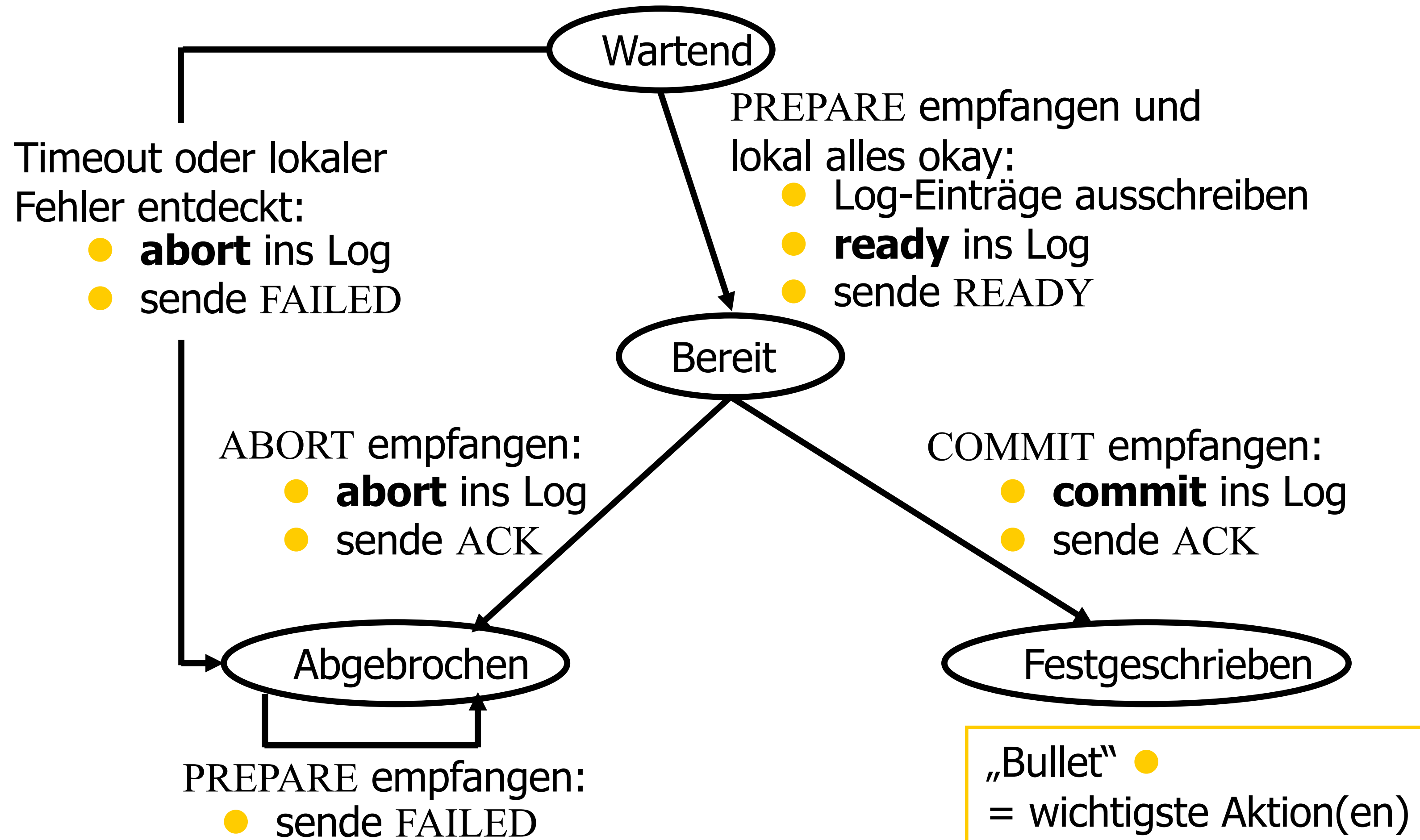
- hat *K* von **allen** *n* Agenten $A_1,\ldots,A_n$ ein READY erhalten, kann *K* ein COMMIT an alle Agenten schicken mit der Aufforderung, die Änderungen von *T* lokal festzuschreiben; antwortet einer der Agenten mit FAILED od. gar nicht innerhalb einer bestimmten Zeit (*timeout*), schickt *K* ein ABORT an alle Agenten und diese machen die Änderungen der Transaktion rückgängig
- haben die Agenten ihre lokale EOT-Behandlung abgeschlossen, schicken sie eine ACK-Nachricht (=acknowledgement, dt. Bestätigung) an den Koordinator

Lineare Organisationsform beim 2PC-Protokoll

Zustandsübergang beim 2PC-Protokoll: Koordinator



Zustandsübergang beim 2PC-Protokoll: Agent



Fehlersituationen des 2PC-Protokolls

Absturz eines Koordinators

Absturz eines Agenten

verlorengegangene Nachrichten

Absturz eines Koordinators

- Absturz vor dem Senden einer COMMIT-Nachricht → Rückgängigmachen der Transaktion durch Versenden einer ABORT-Nachricht

- Absturz nachdem Agenten ein READY mitgeteilt haben → Blockierung der Agenten

  ⇒ Hauptproblem des 2PC-Protokolls beim Absturz des Koordinators, da dadurch die Verfügbarkeit des Agenten bezüglich andere globaler und lokaler Transaktionen drastisch eingeschränkt ist

Um Blockierung von Agenten zu verhindern, wurde ein **Dreiphasen-Commit-Protokoll** konzipiert, das aber in der Praxis zu aufwendig ist (VDBMS benutzen das 2PC-Protokoll).

Absturz eines Agenten

- antwortet ein Agent innerhalb eines Timeout-Intervalls nicht auf die *PREPARE*-Nachricht, gilt der Agent als abgestürzt; der Koordinator bricht die Transaktion ab und schickt eine *ABORT*-Nachricht an alle Agenten
- „abgestürzter“ Agent schaut beim Wiederanlauf in seine Log-Datei:
  - kein **ready**-Eintrag bzgl. Transaktion *T* → Agent führt ein abort durch und teilt dies dem Koordinator mit (*FAILED*-Nachricht)
  - **ready**-Eintrag aber kein **commit**-Eintrag → Agent fragt Koordinator, was aus Transaktion *T* geworden ist; Koordinator teilt *COMMIT* oder *ABORT* mit, was beim Agenten zu einem Redo oder Undo der Transaktion führt
  - **commit**-Eintrag vorhanden → Agent weiß ohne Nachfragen, dass ein (lokales) Redo der Transaktion nötig ist

Verlorengegangene Nachrichten

- *PREPARE*-Nachricht des Koordinators an einen Agenten geht verloren oder
- *READY*-(oder *FAILED*-)Nachricht eines Agenten geht verloren

→ nach Timeout-Intervall geht Koordinator davon aus, dass betreffender Agent nicht funktionsfähig ist und sendet *ABORT*-Nachricht an alle Agenten (Transaktion gescheitert)

- Agent erhält im Zustand **Bereit** keine Nachricht vom Koordinator

→ Agent ist blockiert, bis *COMMIT*- oder *ABORT*-Nachricht vom Koordinator kommt, da Agent nicht selbst entscheiden kann (deshalb schickt Agent eine „Erinnerung" an den Koordinator)

Mehrbenutzersynchronisation in VDBMS

Serialisierbarkeit

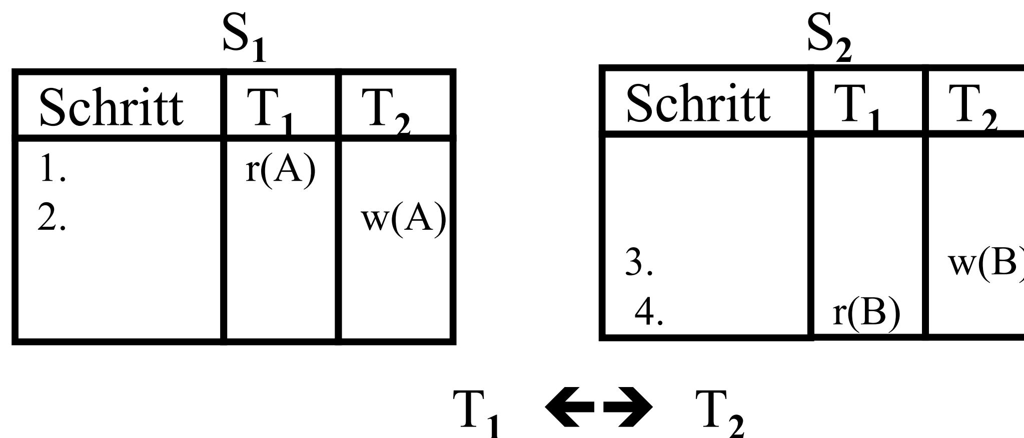
Zwei-Phasen-Sperrprotokoll in VDBMS

- lokale Sperrverwaltung an jeder Station
- globale Sperrverwaltung

Serialisierbarkeit

Lokale Serialisierbarkeit an jeder der an den Transaktionen beteiligten Stationen reicht nicht aus. Deshalb muß man bei der Mehrbenutzersynchronisation auf globaler Serialisierbarkeit bestehen.

Beispiel (lokal serialisierbare Historien):



S_1

Schritt	T_1	T_2
1.	r(A)	
2.		w(A)

S_2

Schritt	T_1	T_2
3.		w(B)
4.	r(B)	

$T_1 \leftrightarrow T_2$

Lokale Sperrverwaltung

globale Transaktion muss vor Zugriff/Modifikation eines Datums A, das auf Station S liegt, eine Sperre vom Sperrverwalter der Station S erwerben

Verträglichkeit der angeforderten Sperre mit bereits existierenden Sperren kann lokal entschieden werden → favorisiert lokale Transaktionen, da diese nur mit ihrem lokalen Sperrverwalter kommunizieren müssen

Globale Sperrverwaltung

= alle Transaktionen fordern alle Sperren an einer einzigen, ausgezeichneten Station an.

Nachteile:

zentraler Sperrverwalter kann zum Engpass des VDBMS werden, besonders bei einem Absturz der Sperrverwalter-Station („rien ne vas plus“)

Verletzung der lokalen Autonomie der Stationen, da auch lokale Transaktionen ihre Sperren bei der zentralisierten Sperrverwaltung anfordern müssen

⟹ zentrale Sperrverwaltung i.a. nicht akzeptabel

Deadlocks in VDBMS

Erkennung von Deadlocks (Verklemmungen)

- zentralisierte Deadlock-Erkennung
- dezentrale (verteilte) Deadlock-Erkennung

- Vermeidung von Deadlocks

„Verteilter“ Deadlock

S_1

Schritt	T_1	T_2
0.	**BOT**	
1.	**lockS**(A)	
2.	r(A)	
6.		**lockX**(A)
		~~

S_2

Schritt	T_1	T_2
3.		**BOT**
4.		**lockX**(B)
5.		w(B)
7.	**lockS**(B)	
	~~	

Timeout

betreffende Transaktion wird zurückgesetzt und erneut gestartet
→ einfach zu realisieren

Problem: richtige Wahl des Timeout-Intervalls:

- zu lang → schlechte Ausnutzung der Systemressourcen
- zu kurz → Deadlock-Erkennung, wo gar keine Verklemmung vorliegt

Zentralisierte Deadlock-Erkennung

Stationen melden lokal vorliegende Wartebeziehungen an neutralen Knoten, der daraus globalen Wartegraphen aufbaut (Zyklus im Graphen → Deadlock)

→ sichere Lösung

Nachteile:

- hoher Aufwand (viele Nachrichten)
- Entstehung von Phantom-Deadlocks (=nicht-existierende Deadlocks) durch „Überholen" von Nachrichten im Kommunikationssystem

Dezentrale Deadlock-Erkennung

lokale Wartegraphen an den einzelnen Stationen
Erkennen von lokalen Deadlocks

Erkennung globaler Deadlocks:

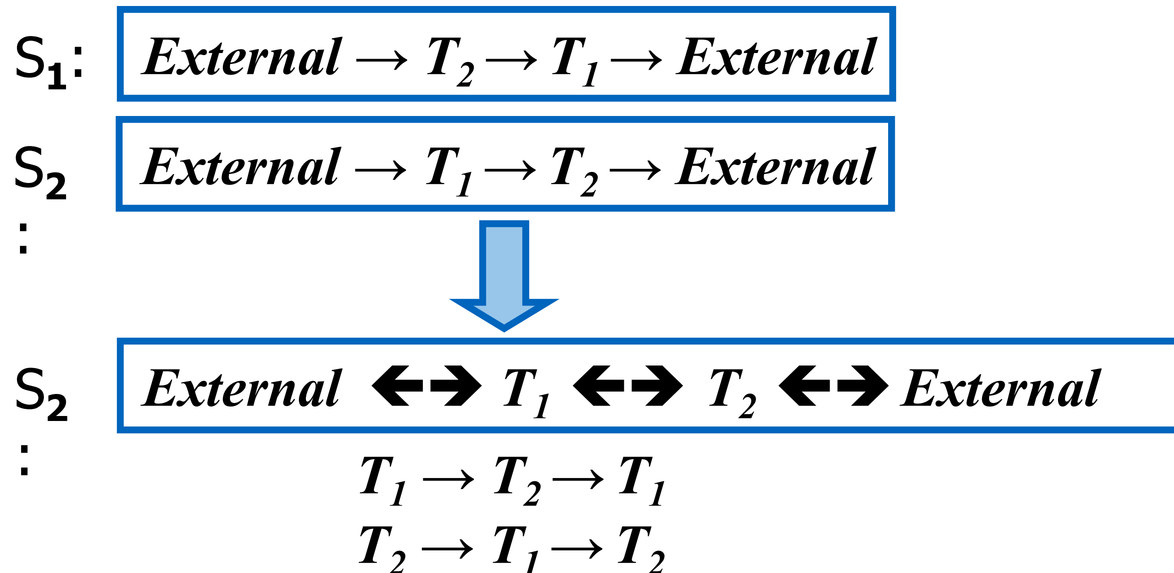
- jeder lokale Wartegraph hat einen Knoten *External*, der stationenübergreifenden Wartebeziehungen zu externen Subtransaktionen modelliert
- Zuordnung jeder Transition zu einem Heimatknoten, von wo aus *externe Subtransaktionen* auf anderen Stationen initiiert werden

Die Kante $\boldsymbol{External \rightarrow T_i}$
wird für jede „von außen" kommende Transaktion T_i eingeführt.

Die Kante $\boldsymbol{T_j \rightarrow External}$
wird für jede von außen kommende Transaktion T_j dieser Station eingeführt, falls T_j „nach außen" geht.

Beispiel:

S_1 Heimatknoten von T_1, S_2 Heimatknoten von T_2



Zur Reduzierung des Nachrichtenaufkommens wird der Pfad

$External \rightarrow T_1' \rightarrow T_2' \rightarrow \ldots \rightarrow T_n' \rightarrow External$

nur weitergereicht, wenn T_1' einen kleineren Identifikator als T_n' hat (= path pushing).

Deadlock-Vermeidung

optimistische Mehrbenutzersynchronisation:

nach Abschluss der Transaktionsbearbeitung wird Validierung durchgeführt

Zeitstempel-basierende Synchronisation:

Zuordnung eines Lese-/Schreib-Stempels zu jedem Datum

entscheidet, ob beabsichtigte Operation durchgeführt werden kann ohne Serialisierbarkeit zu verletzen oder ob Transaktion abgebrochen wird (**abort**)

Deadlockvermeidung bei Sperrbasierte Synchronisation

wound/wait:
- nur jüngere Transaktionen warten auf ältere;
- fordert ältere Transaktion Sperre an, die mit der von der jüngeren Transaktion gehaltenen nicht verträglich ist, wird jüngere Transaktion abgebrochen

wait/die:
- nur ältere Transaktionen warten auf jüngere;
- fordert jüngere Transaktion Sperre an, die mit der von der älteren Transaktion gehaltenen nicht kompatibel ist, wird jüngere Transaktion abgebrochen

Voraussetzungen für Deadlockvermeidungsverfahren

Vergabe global eindeutiger Zeitstempel als Transaktionsidentifikatoren

lokale Zeit	Stations-ID

- lokale Uhren müssen hinreichend genau aufeinander abgestimmt sein

Synchronisation bei replizierten Daten

ROWA: Read One, Write All

Problem:

Zu einem Datum A gibt es mehrere Kopien A1, A2, ..., An, die auf unterschiedlichen Stationen liegen.

Eine Lesetransaktion erfordert nur eine Kopie, bei Änderungstransaktionen müssen aber alle bestehenden Kopien geändert werden.

⇒ hohe Laufzeit und Verfügbarkeitsprobleme

Quorum-Consensus Verfahren

Ausgleich der Leistungsfähigkeit zwischen Lese- und Änderungstransaktionen
$\rightarrow$ teilweise Verlagerung des Overheads von den Änderungs- zu den Lesetransaktionen indem den Kopien A_i eines replizierten Datums A individuelle Gewichte zugeordnet werden

- ***Lesequorum*** $Q_r(A)$
- ***Schreibquorum*** $Q_w(A)$

Folgende Bedingungen müssen gelten:

1. $Q_w(A) + Q_w(A) > W(A)$
2. $Q_r(A) + Q_w(A) > W(A)$

Beispiel

Station (S_i)	Kopie (A_i)	Gewicht (w_i)
S_1	A_1	3
S_2	A_2	1
S_3	A_3	2
S_4	A_4	2

$$W(A) = \sum_{i=1}^{4} w_i(A) = 8$$

$$Q_r(A) = 4$$

$$Q_w(A) = 5$$

Zustände

a) vor dem Schreiben eines Schreibquorums

Station	Kopie	Gewicht	Wert	Versions#
S_1	A_1	3	1000	1
S_2	A_2	1	1000	1
S_3	A_3	2	1000	1
S_4	A_4	2	1000	1

b) nach dem Schreiben eines Schreibquorums

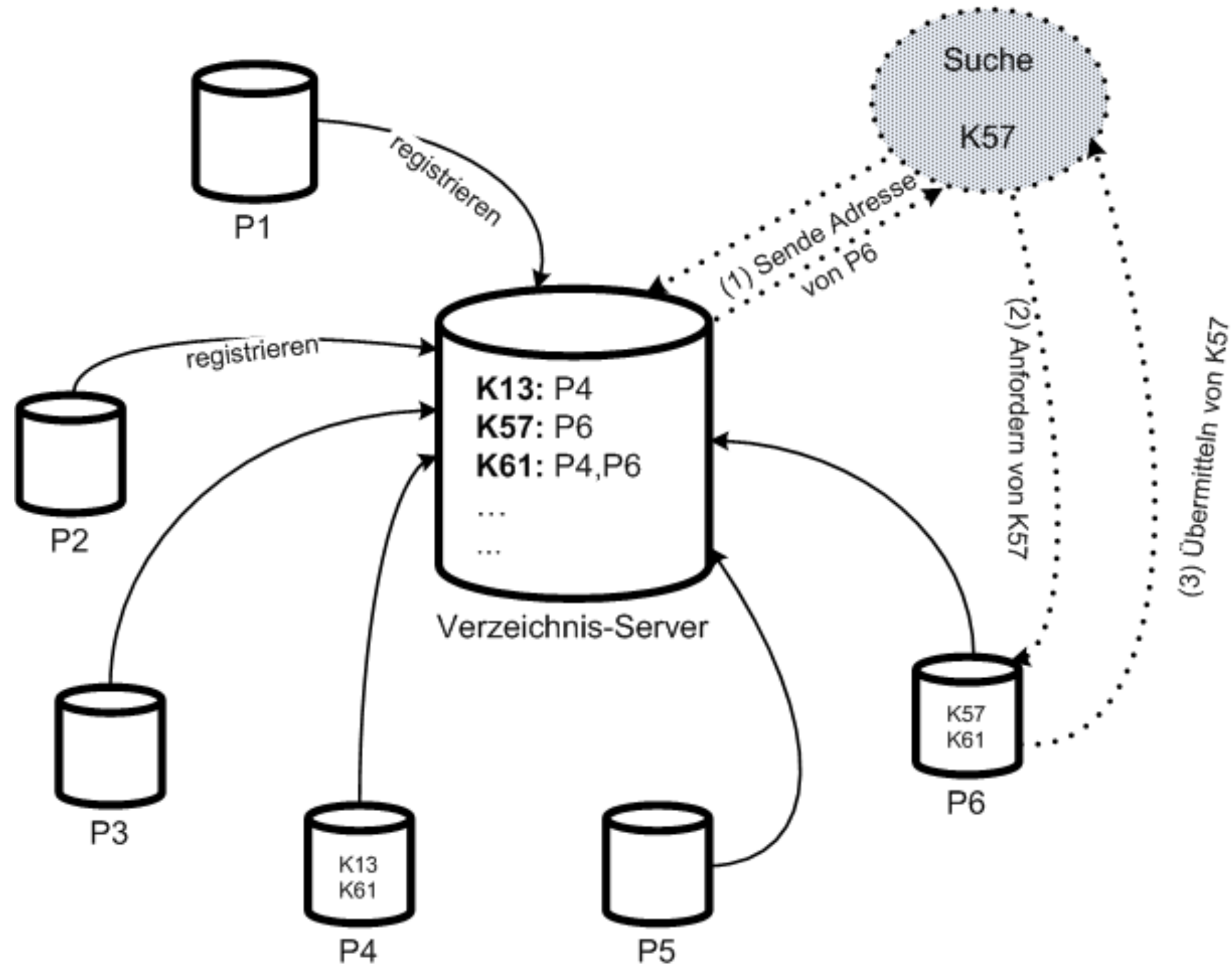
Station	Kopie	Gewicht	Wert	Versions#
S_1	A_1	3	1100	2
S_2	A_2	1	1000	1
S_3	A_3	2	1100	2
S_4	A_4	2	1000	1

Peer to Peer-Informationssysteme

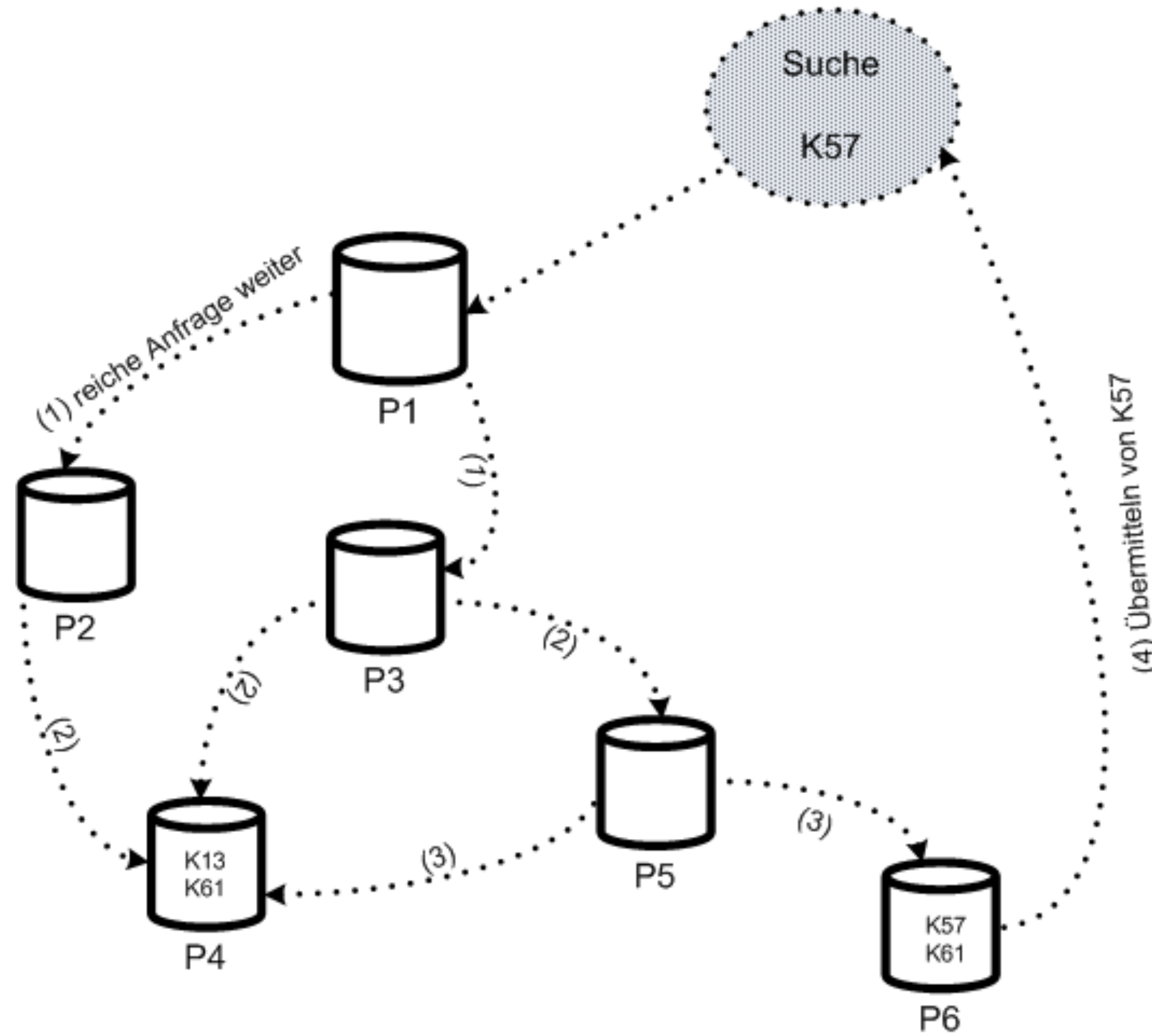
Seti@Home
- P2P number crunching

Napster
- P2P file sharing / Informationsmanagement

Napster-Architektur



Gnutella-Architektur



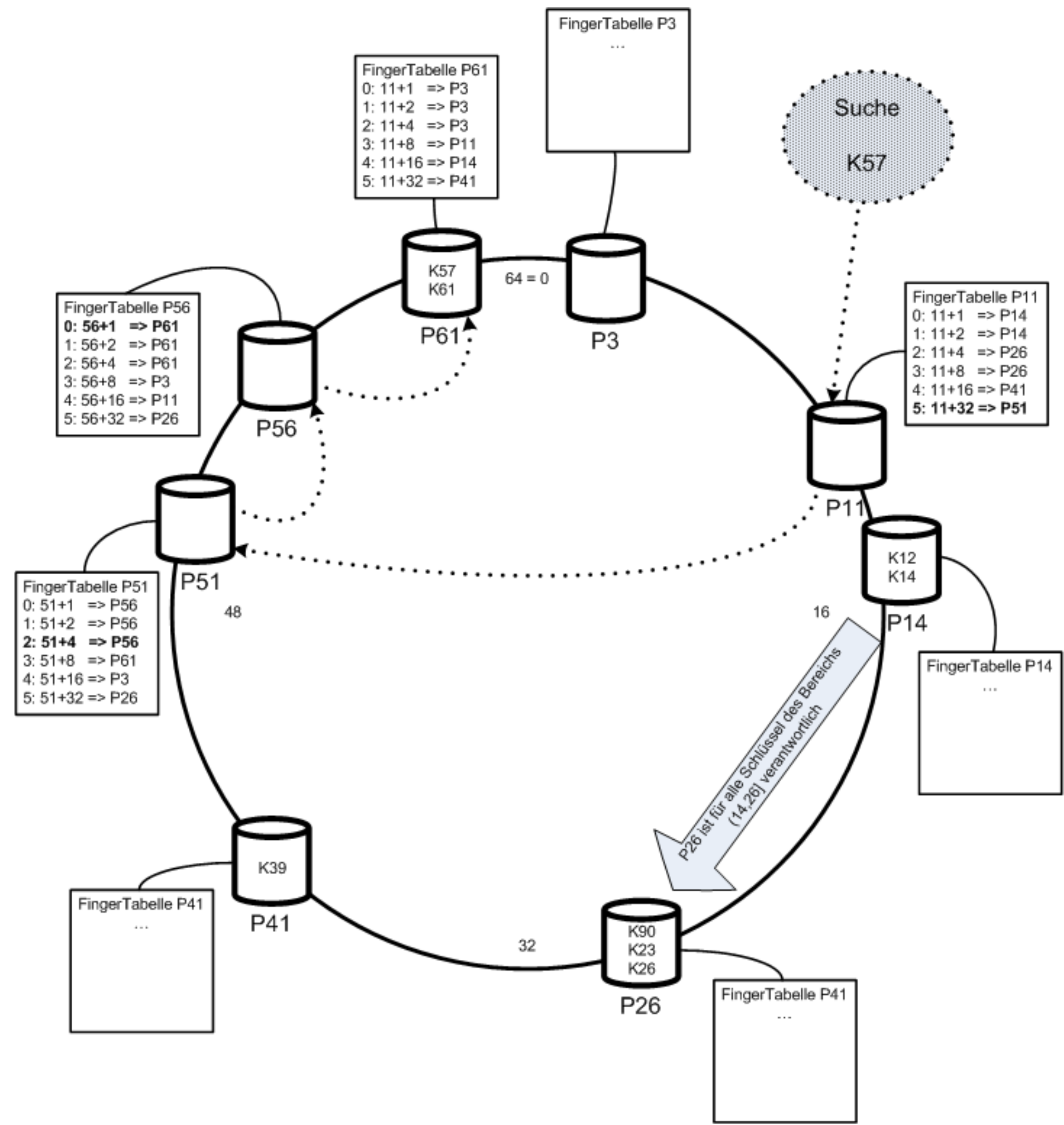
DHT: Distributed Hash Table

Basieren auf „consistent hashing“

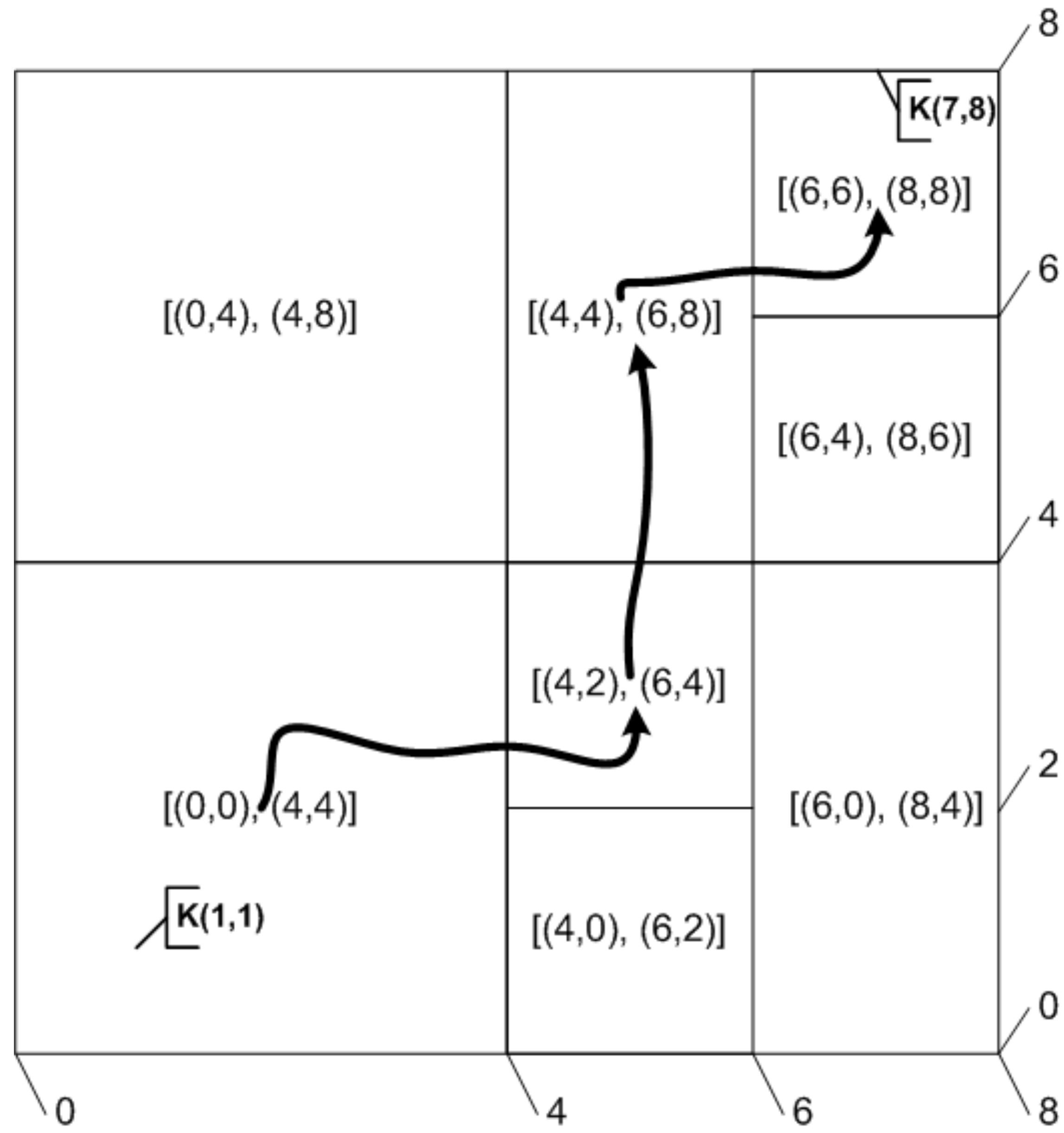
Vollständige Dezentralisierung der Kontrolle

Dennoch zielgerichtete Suche

CHORD



CAN



No-SQL Datenbanken

Internet-scale Skalierbarkeit

CAP-Theorem: nur 2 von 3 Wünschen erfüllbar

- Konsistenz (Consistency)
- Zuverläassigkeit/Verfügbarkeit (Availability)
- Partitionierungs-Toleranz

No-SQL Datenbanksysteme verteilen die Last innerhalb eines Clusters/Netzwerks

- Dabei kommen oft DHT-Techniken zum Einsatz

Schnittstelle der No-SQL Datenbanken

Insert(k,v)
Lookup(k)
Delete(k)

Extrem einfach ➔ effizient
Aber: wer macht denn die Joins/Selektionen/…

- ➔ das Anwendungsprogramm

Konsistenzmodell: ~~C~~AP

Relaxiertes Konsistenzmodell

- Replizierte Daten haben nicht alle den neuesten Zustand
  - Vermeidung des (teuren) Zwei-Phasen-Commit-Protokolls
- Transaktionen könnten veraltete Daten zu lesen bekommen
- Eventual Consistency
  - Würde man das System anhalten, würden alle Kopien irgendwann (also eventually) in denselben Zustand übergehen
- Read your Writes-Garantie
  - Tx leist auf jeden Fall ihre eigenen Änderungen
- Monotonic Read-Garantie
  - Tx würde beim wiederholten Lesen keinen älteren Zustand als den vorher mal sichtbaren lesen

Systeme

MongoDB
Cassandra
Dynamo
BigTable
Hstore
SimpleDB
S3
CockroachDB