Verteilte Datenbanken

Verteilte Datenbanken

Terminologie

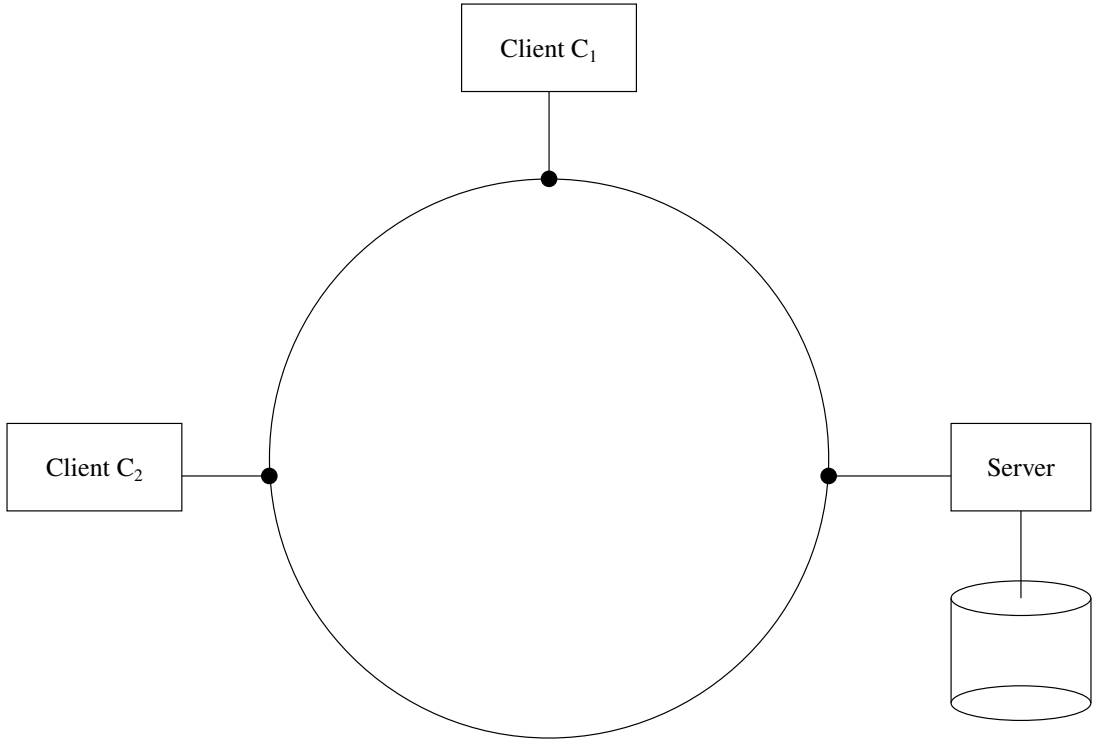
- Eine *verteilte Datenbank* (VDBMS) ist eine Sammlung von Informationseinheiten, die auf verschiedene Rechner verteilt ist, die durch Kommunikationsnetze verbunden sind
- Jede Station kann
  - autonom mit lokalen Daten arbeiten
  - global mit anderen Rechnern des Netzes zusammenarbeiten

Kommunikationsnetz

- Bei dem Kommunikationsnetz kann es sich handeln um
  - LAN: local area network (Ethernet, Token-Ring, FDDI-Netz)
  - WAN: wide area network (Internet)
  - Telefonverbindungen: ISDN, Modem
- Kommunikationsnetz ist transparent für Datenbankanwendung

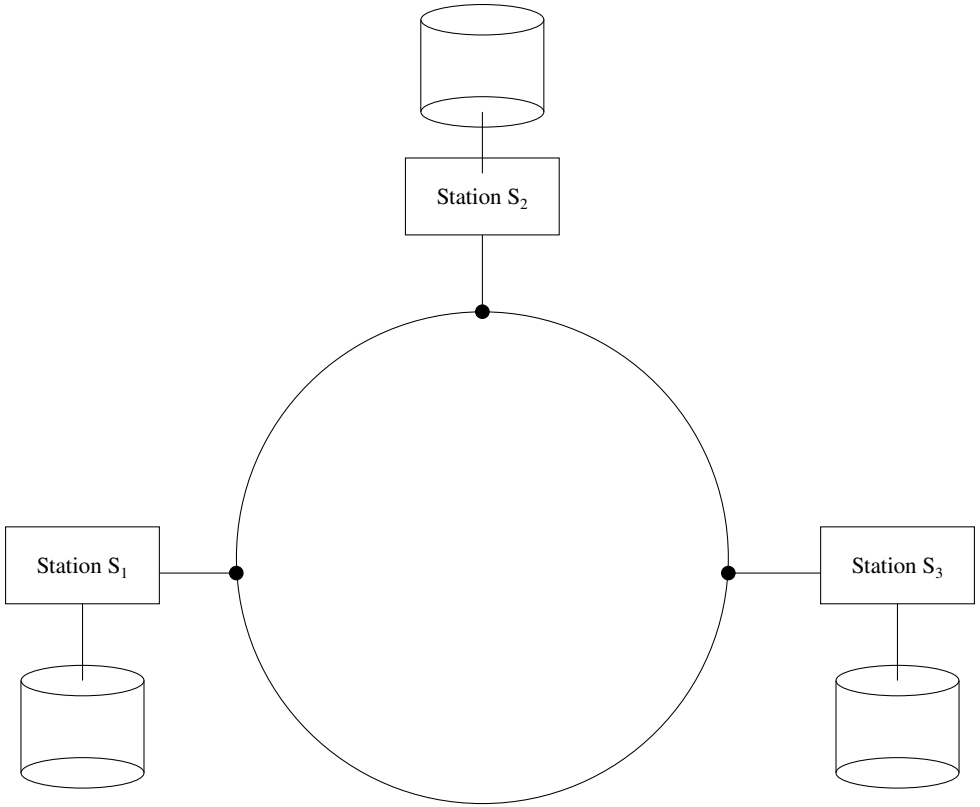
Abgrenzung

- VDBMS ist keine Client-Server-Architektur

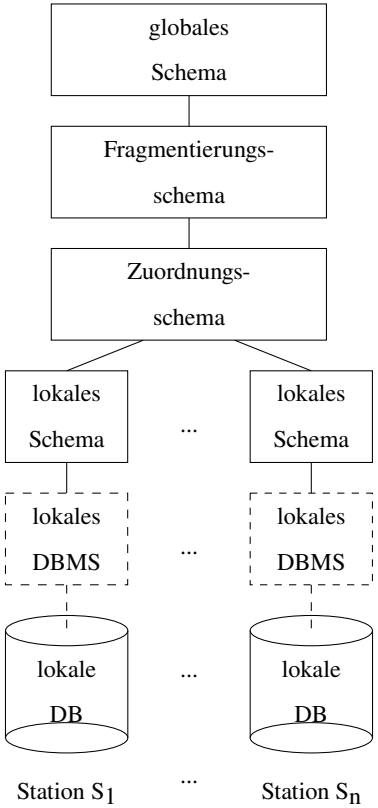


Abgrenzung(2)

- Jede Station hält eigene Daten



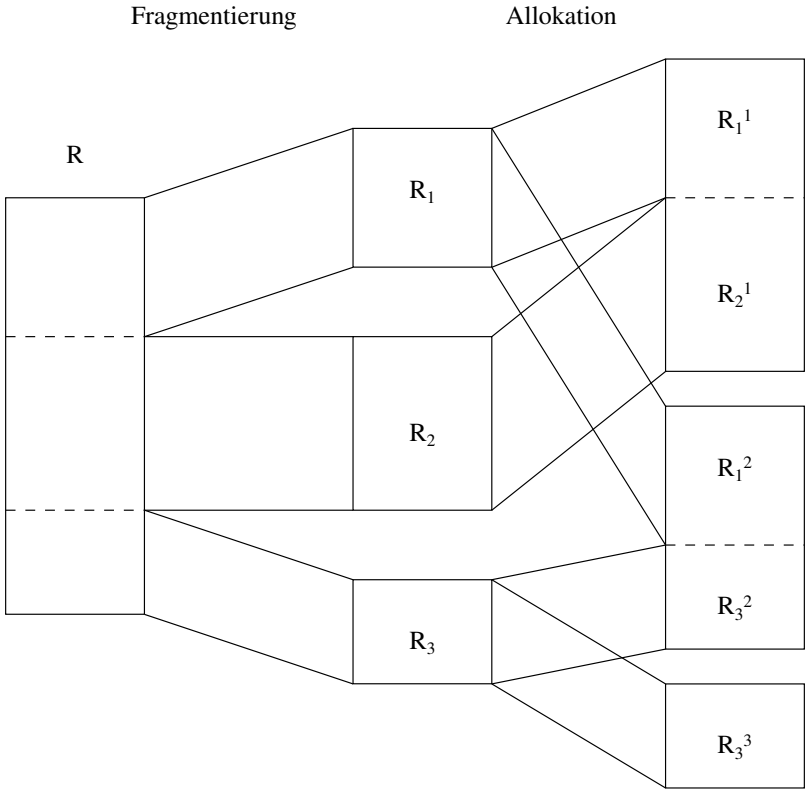
Aufbau eines VDBMS



Fragmentierung/Allokation

- Fragmentierung
  - Fragmente enthalten Daten mit gleichem Zugriffsverhalten
- Allokation
  - Fragmente werden den Stationen zugeordnet
    - Mit Replikation
    - Ohne Replikation

Fragmentierung/Allokation(2)



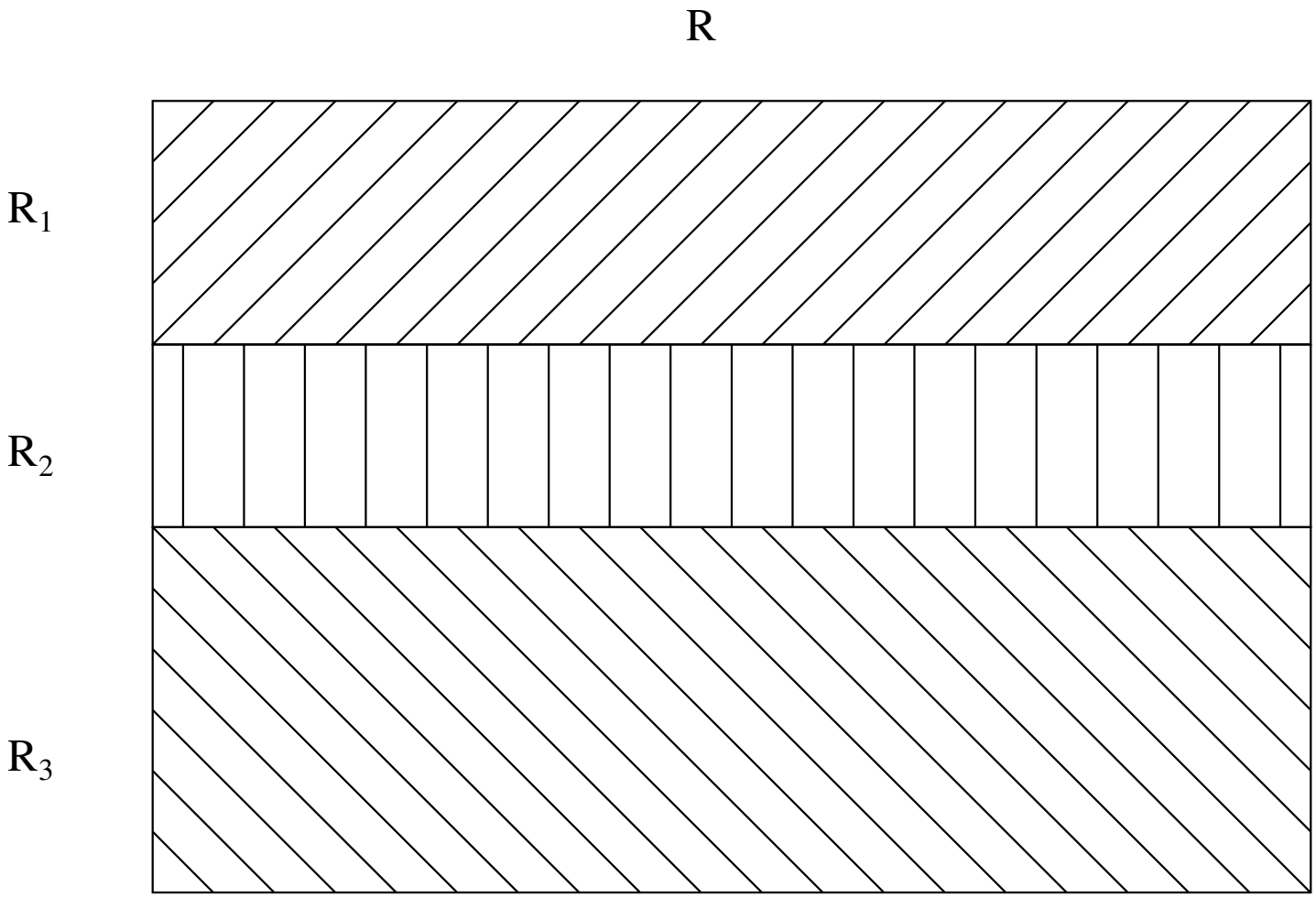
Fragmentierung

- Es existieren verschiedene Methoden der Fragmentierung:
  - Horizontal: Zerlegung einer Relation in disjunkte Tupelmengen, Zerlegung durch Selektionen
  - Vertikal: Zusammenfassen von Attributen mit gleichen Zugriffsmustern, Zerlegung durch Projektionen
  - Kombiniert: horizontale und vertikale Fragmentierung auf der gleichen Relation

Korrektheit

- Es gibt drei grundlegende Korrektheitsanforderungen an Fragmentierungen:
  - Rekonstruierbarkeit: die Ursprungsrelation lässt sich aus den Fragmenten wiederherstellen
  - Vollständigkeit: jedes Datum ist einem Fragment zugeordnet
  - Disjunktheit: Fragmente überlappen sich nicht, d.h. ein Datum ist nicht mehreren Fragmenten zugeordnet

Horizontale Fragmentierung



Horizontale Fragmentierung(2)

- Bei n Zerlegungsprädikaten gibt es insgesamt 2^n mögliche Fragmente
- Ein Prädikat p_1:

$$R_1 := \sigma_{p_1}(R)$$
$$R_2 := \sigma_{\neg p_1}(R)$$

- Zwei Prädikate p_1, p_2:

$$R_1 := \sigma_{p_1 \wedge p_2}(R)$$
$$R_2 := \sigma_{p_1 \wedge \neg p_2}(R)$$
$$R_3 := \sigma_{\neg p_1 \wedge p_2}(R)$$
$$R_4 := \sigma_{\neg p_1 \wedge \neg p_2}(R)$$

Beispiel

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

Beispiel(2)

$$
\begin{aligned}
p_1 &\equiv \text{Fakultät} = \text{'Theologie'} \\
p_2 &\equiv \text{Fakultät} = \text{'Physik'} \\
p_3 &\equiv \text{Fakultät} = \text{'Philosophie'}
\end{aligned}
$$

$$
\begin{aligned}
\text{TheolProfs}' &:= \sigma_{p_1 \wedge \neg p_2 \wedge \neg p_3}(\text{Professoren}) = \sigma_{p_1}(\text{Professoren}) \\
\text{PhysikProfs}' &:= \sigma_{\neg p_1 \wedge p_2 \wedge \neg p_3}(\text{Professoren}) = \sigma_{p_2}(\text{Professoren}) \\
\text{PhiloProfs}' &:= \sigma_{\neg p_1 \wedge \neg p_2 \wedge p_3}(\text{Professoren}) = \sigma_{p_3}(\text{Professoren}) \\
\text{AndereProfs}' &:= \sigma_{\neg p_1 \wedge \neg p_2 \wedge \neg p_3}(\text{Professoren})
\end{aligned}
$$

Abgeleitete h. Fragmentierung

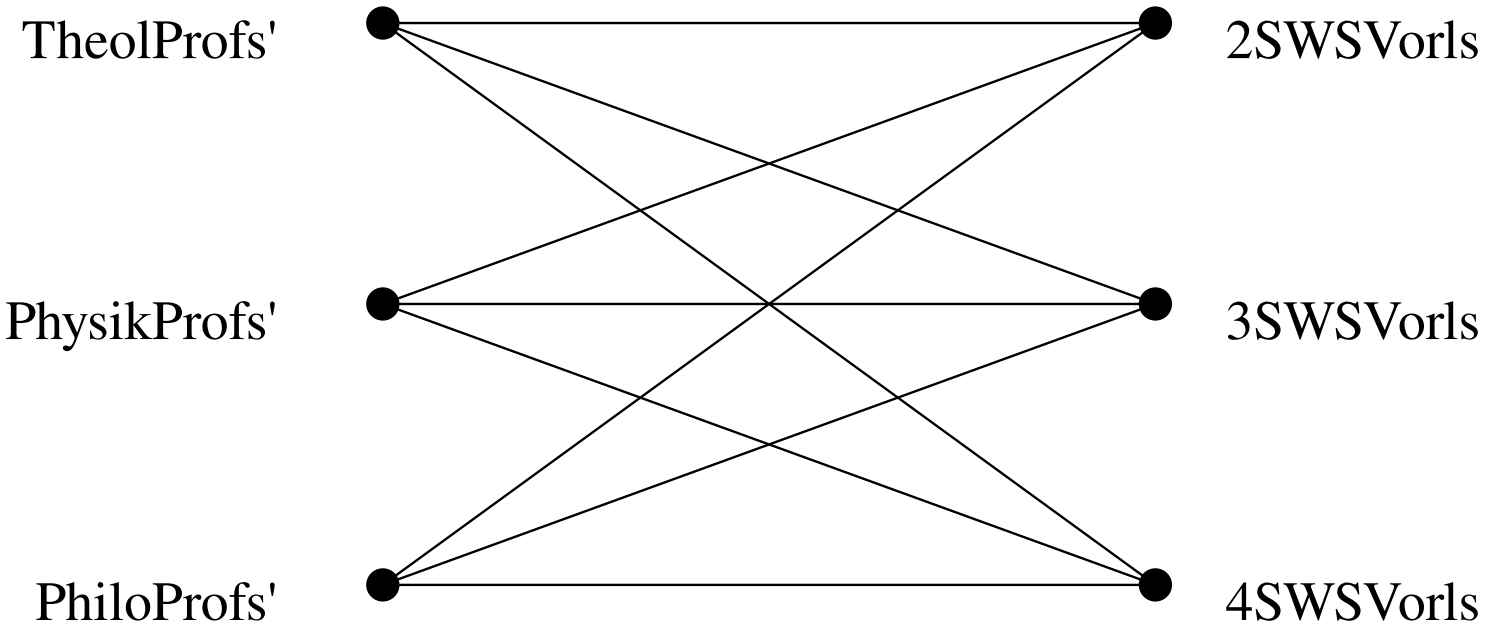
- Manchmal ist es sinnvoll, eine Relation abhängig von einer anderen horizontalen Fragmentierung zu zerlegen
- Beispiel: völlig unabhängige Zerlegung von Vorlesungen nach SWS:

$$\begin{aligned} \text{2SWSVorls} &:= \sigma_{\text{SWS}=2}(\text{Vorlesungen}) \\ \text{3SWSVorls} &:= \sigma_{\text{SWS}=3}(\text{Vorlesungen}) \\ \text{4SWSVorls} &:= \sigma_{\text{SWS}=4}(\text{Vorlesungen}) \end{aligned}$$

Abgeleitete Fragmentierung(2)

- Bei Beantwortung folgender Anfrage müssen 9 Joins von Fragmenten durchgeführt werden:

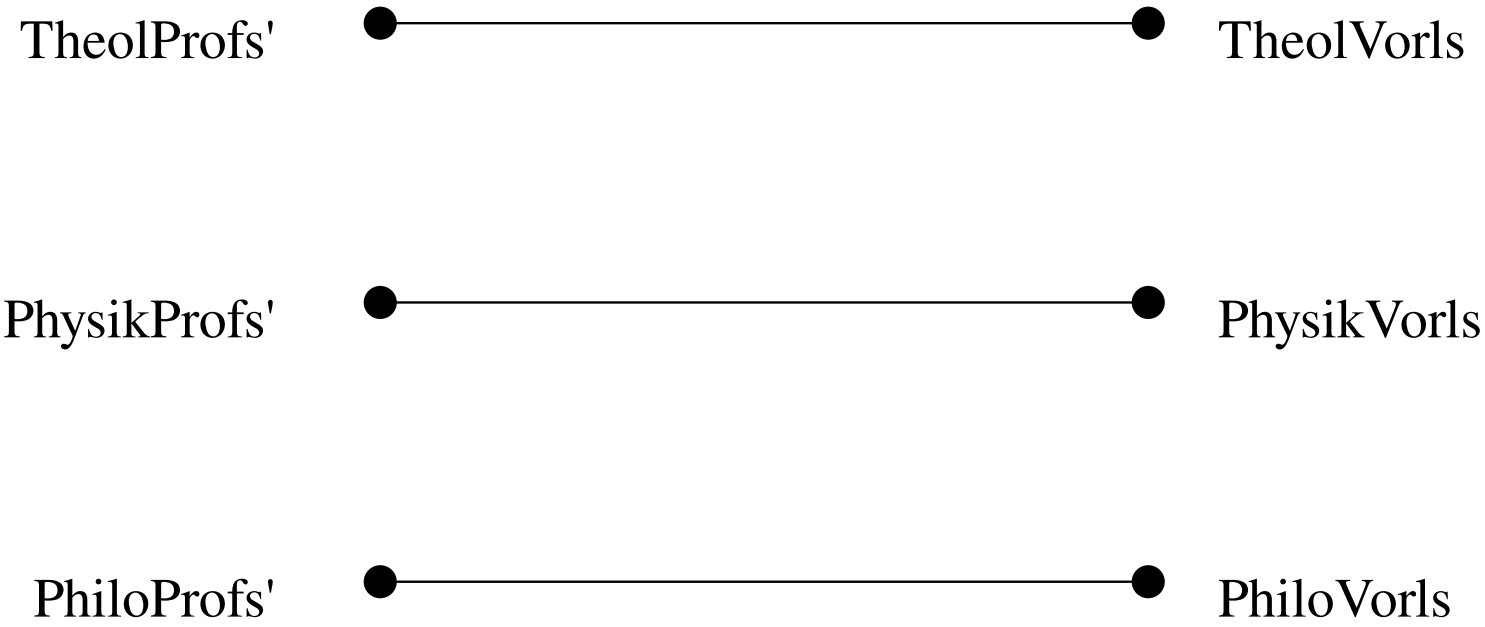
```
select Titel, Name
from   Vorlesungen, Professoren
where  gelesenVon = PersNr
```



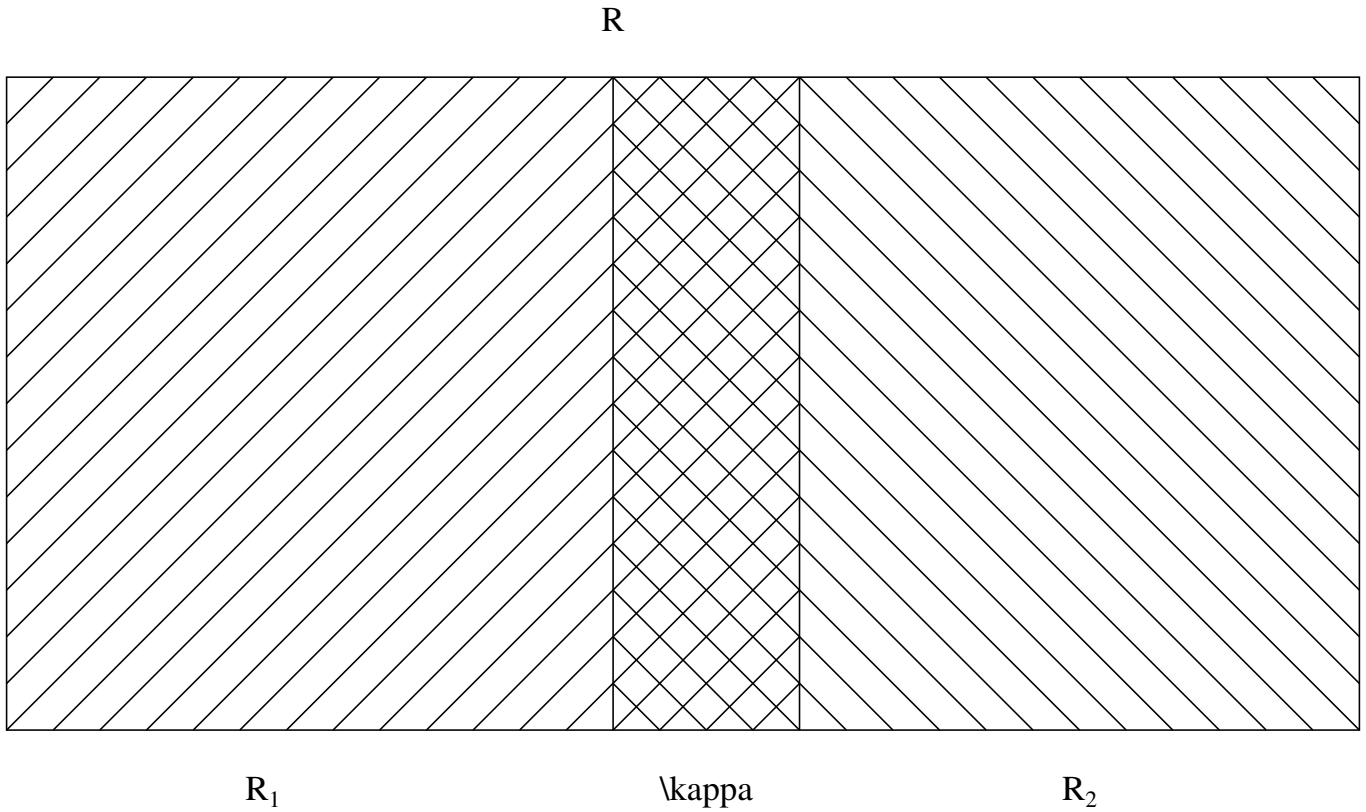
Abgeleitete Fragmentierung(3)

- Sinnvoller ist folgende (abgeleitete) Fragmentierung:

$$\begin{aligned} \text{TheolVorls} &:= \text{Vorlesungen} \ltimes_{\text{gelesenVon=PersNr}} \text{TheolProfs}' \\ \text{PhysikVorls} &:= \text{Vorlesungen} \ltimes_{\text{gelesenVon=PersNr}} \text{PhysikProfs}' \\ \text{PhiloVorls} &:= \text{Vorlesungen} \ltimes_{\text{gelesenVon=PersNr}} \text{PhiloProfs}' \end{aligned}$$



Vertikale Fragmentierung



Vertikale Fragmentierung(2)

- Bei Zerlegung ohne Überlappung gibt es bei vertikaler Fragmentierung ein Problem: Verstoß gegen die Rekonstruierbarkeit
- Man lässt “leichten” Verstoß gegen Disjunktheit zu:
  - Jedes Fragment enthält Primärschlüssel
  - Jedem Tupel der Originalrelation wird künstlicher Surrogatschlüssel zugewiesen, der in Fragment übernommen wird

Beispiel

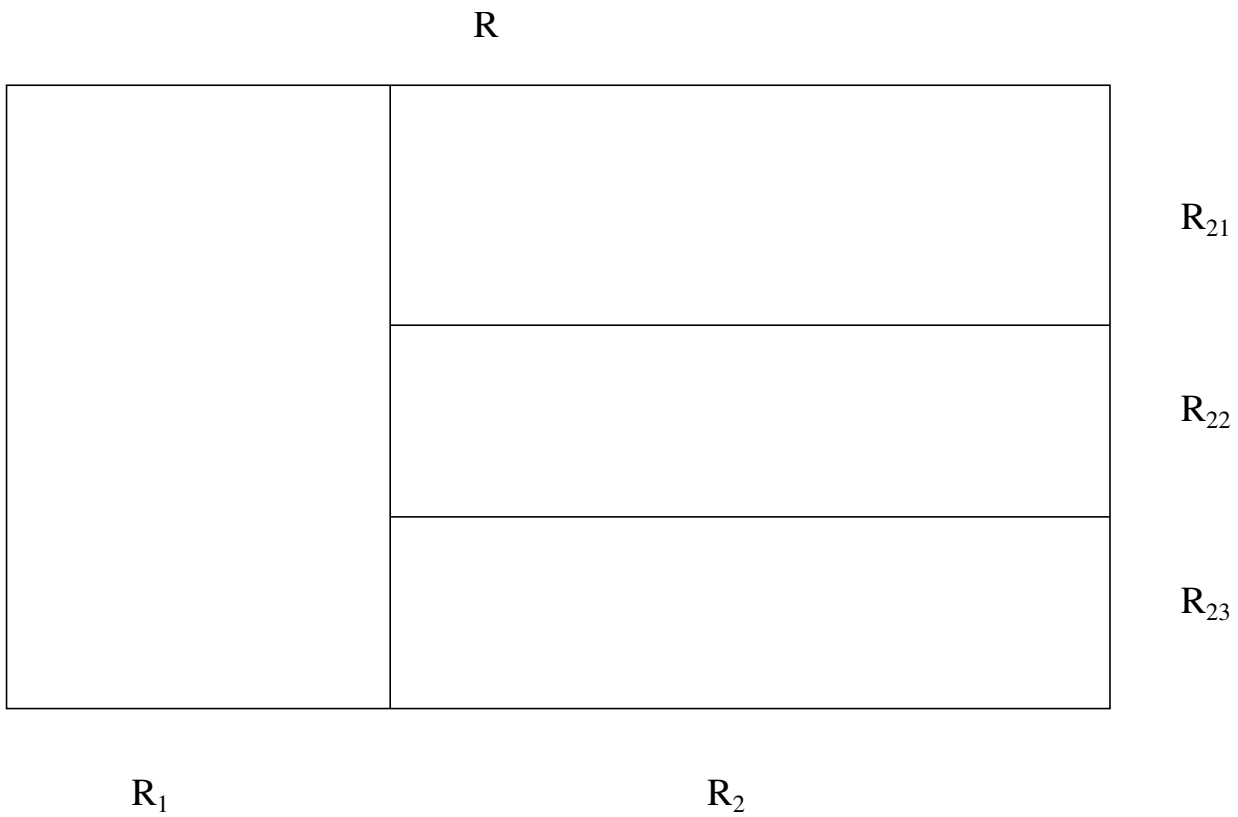
- Ein Fragment für die Univerwaltung: ProfVerw
- Ein Fragment für Lehre und Forschung: Profs

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

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

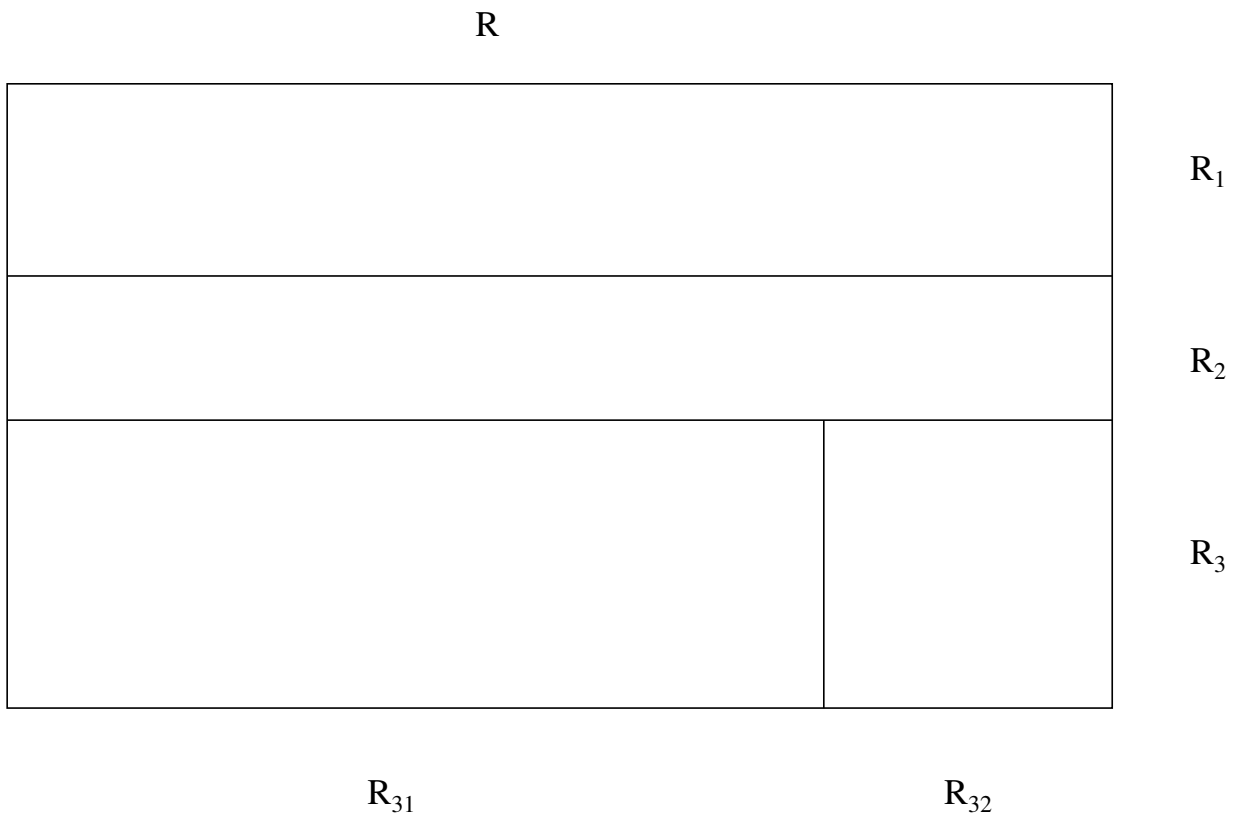
Kombinierte Fragmentierung

- Erst vertikal, dann horizontal:

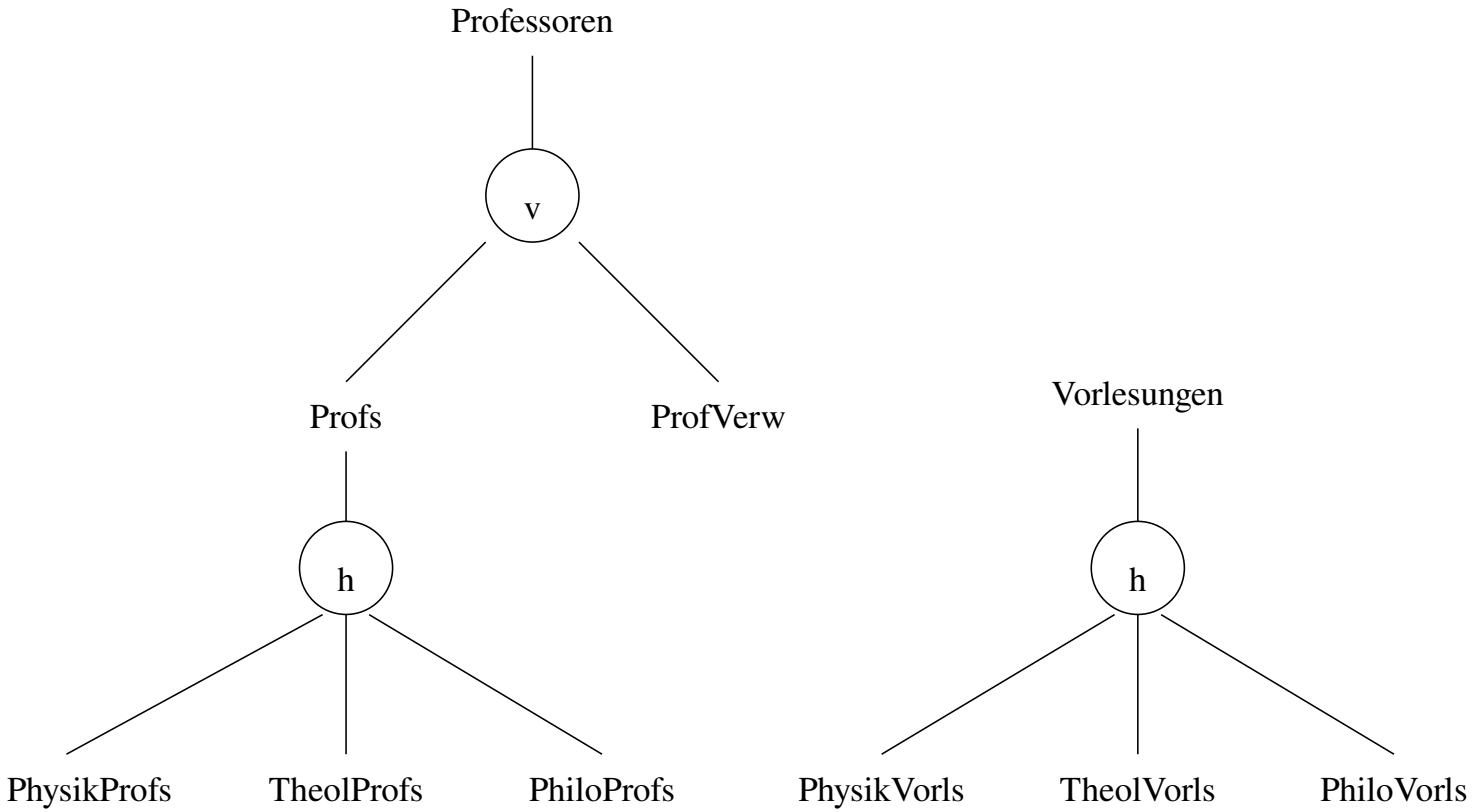


Kombin. Fragmentierung(2)

- Erst horizontal, dann vertikal:



Beispielanwendung



Beispielanwendung(2)

- Bei der Allokation werden nun die Fragmente Stationen zugeteilt (hier ohne Replikation)

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

Transparenz

- Unter *Transparenz* versteht man den Grad an Unabhängigkeit, den ein VDBMS dem Benutzer vermittelt
- Es werden verschiedene Stufen unterschieden:
  - Fragmentierungstransparenz
  - Allokationstransparenz
  - Lokale Schema-Transparenz

Fragmentierungstransparenz

- Höchste Stufe der Transparenz (Idealzustand)
- Benutzer arbeitet auf globalem Schema und VDBMS übersetzt Anfragen in Operationen auf Fragmenten
- Beispiel:

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

Allokationstransparenz

- Nächst niedrigere Stufe
- Benutzer muss zwar Fragmente kennen, aber nicht deren Aufenthaltsort
- Beispiel:

```
select Gehalt
from   ProfVerw
where  Name = 'Sokrates'
```

Lokale Schema-Transparenz

- Bei dieser Stufe muss Benutzer sowohl Fragment als auch Aufenthaltsort kennen
- Es stellt sich die Frage, inwieweit überhaupt noch Transparenz vorliegt (alle Rechner benutzen dasselbe Datenmodell)
- Beispiel:

```
select Name
from   TheolProfs at S_Theol
where  Rang = 'C3'
```

Aspekte verteilter DBS

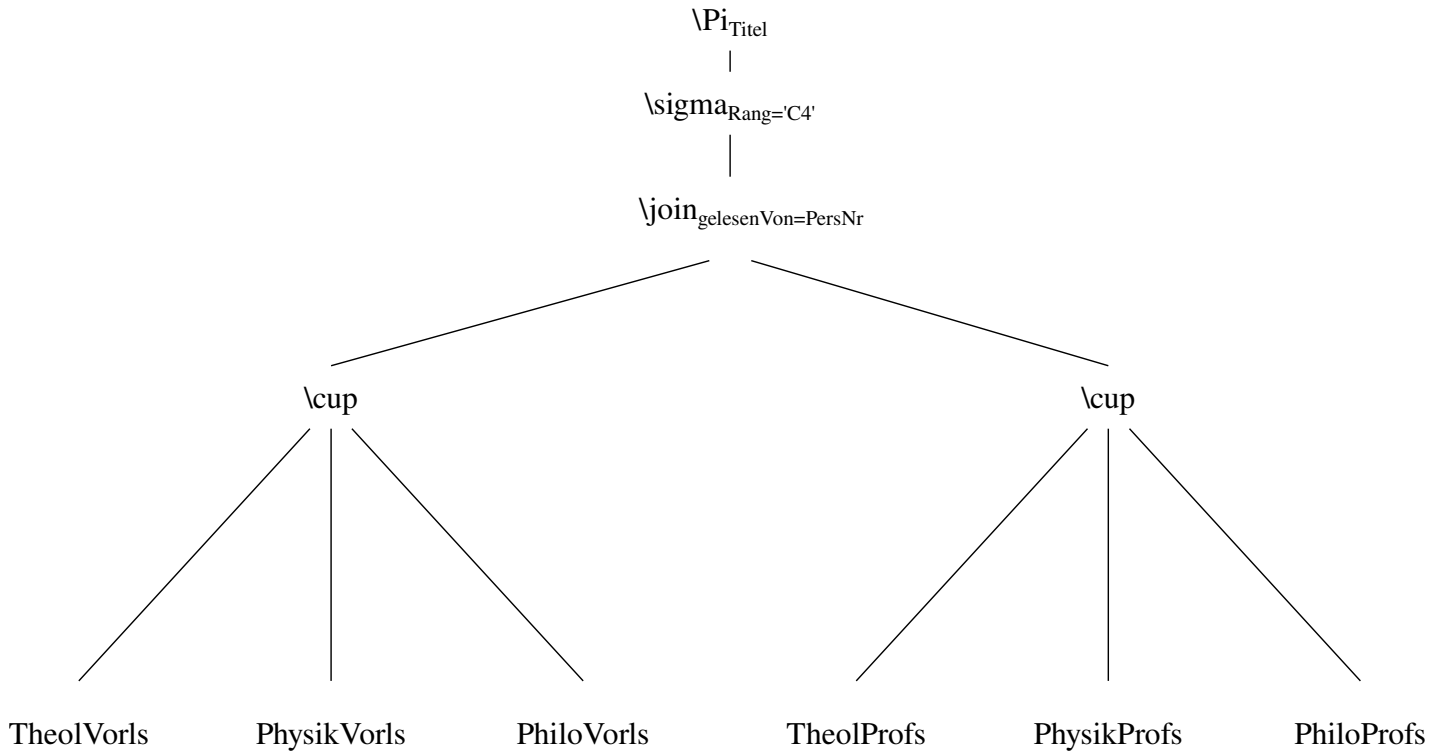
- Durch die Verteilung der Daten müssen folgende Bereiche angepasst werden:
  - Anfragebearbeitung/-optimierung
  - Transaktionskontrolle
  - Mehrbenutzersynchronisation

- Wir unterscheiden zwischen
  - Horizontaler Fragmentierung
  - Vertikaler Fragmentierung

```
select Titel
from   Vorlesungen, Profs
where  gelesenVon = PersNr and
       Rang = 'C3'
```

- Rekonstruiere alle in der Anfrage vorkommenden globalen Relationen aus den Fragmenten
- Kombiniere den Rekonstruktionsausdruck mit dem Ausdruck aus der Übersetzung der SQL-Anfrage

Kanonische Form



- Kanonische Form ist zwar korrekt, aber ineffizient
- Eine zentrale Eigenschaft der relationalen Algebra ist:

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

- Damit ist aber nicht viel erreicht (für das Zusammensetzen von $R_1, ..., R_n$ und $S_1, ..., S_m$ sind $n \cdot m$ Joinoperationen nötig)

Optimierung(2)

- Wenn aber jedes S_i eine abgeleitete horizontale Fragmentierung ist, d.h.

$S_i = S \ltimes_p R_i \quad \text{mit} \quad S = S_1 \cup \cdots \cup S_n$

- dann gilt

$R_i \bowtie_p S_j = \varnothing \quad \text{für} \quad i \neq j$

- und somit

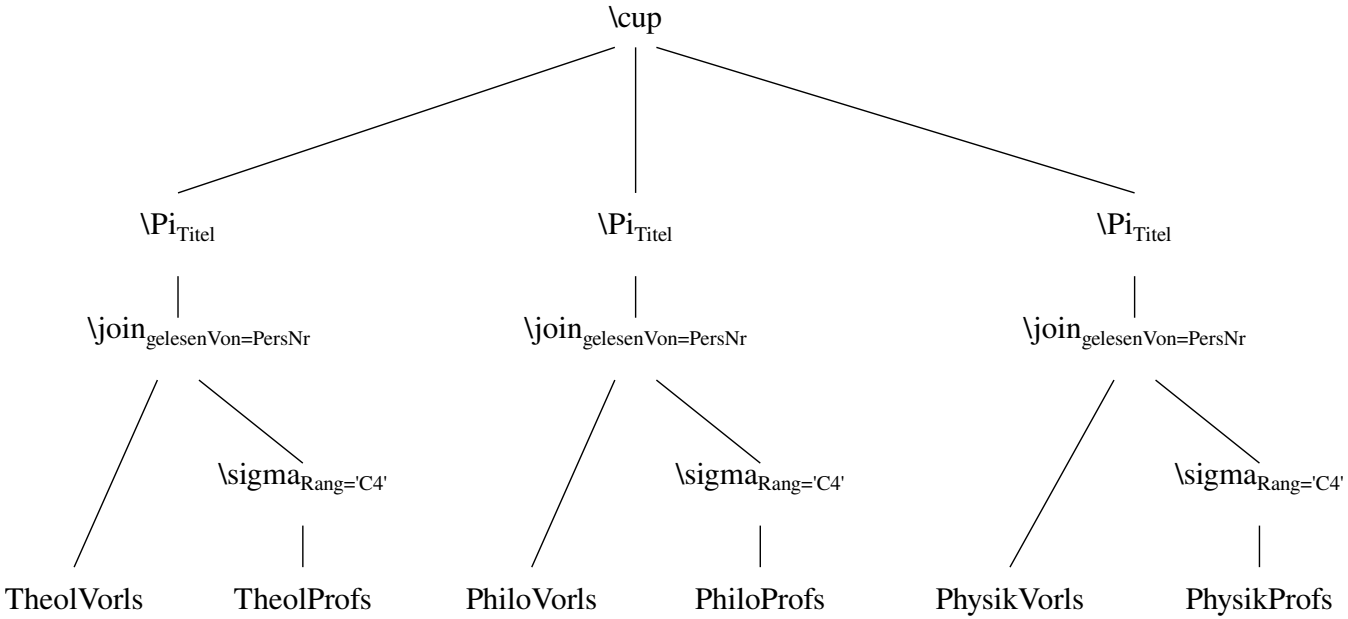
$$(R_1 \cup \cdots \cup R_n) \bowtie_p (S_1 \cup \cdots \cup S_n) = \left(R_1 \bowtie_p S_1\right) \cup \left(R_2 \bowtie_p S_2\right) \cup \cdots \cup \left(R_n \bowtie_p S_n\right)$$

Optimierung(3)

- Damit können die Joins aus unserem Beispiel lokal ausgeführt werden
- Außerdem gibt es Regeln, um Selektionen und Projektionen nach unten zu schieben:

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

Optimierter Plan

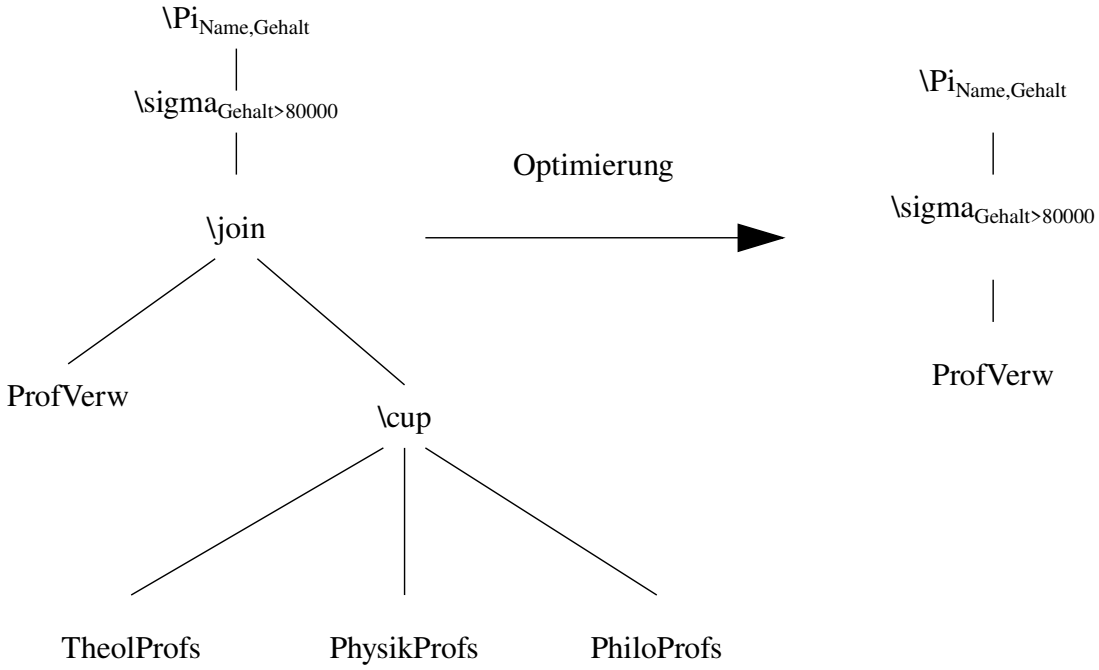


Vertikale Fragmentierung

```
select Name, Gehalt
from   Professoren
where  Gehalt > 80000
```

- Naiver Ansatz: globale Relation rekonstruieren, dann Anfrage auswerten
- Sinnvoller: nur relevante Fragmente holen

Optimierung



- Problem: zu joinende Relationen können auf verschiedenen Stationen liegen
- Betrachtung des allgemeinsten Falls:
  - Äußere Relation R ist auf Station St_R
  - Innere Relation S ist auf Station St_S
  - Ergebnis wird auf Station St_{Result} benötigt

Auswertung ohne Filterung

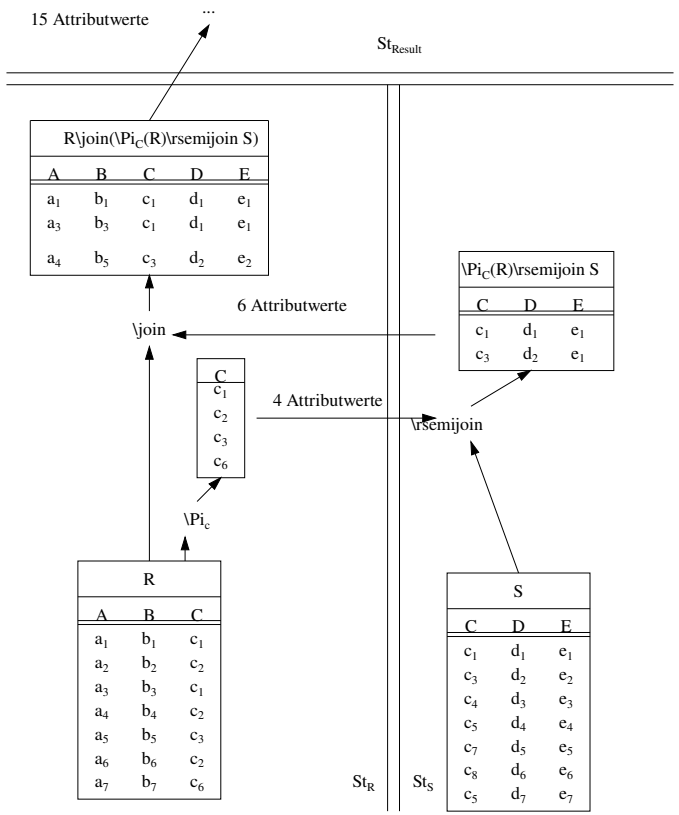
- Nested Loop: iteriere durch Tupel von R, schicke jedes Tupel zu St_S, suche passende Tupel, joine und schicke Ergebnis nach St_{Result}
- Transfer einer Relation: schicke komplette Relation zum anderen Knoten und führe dort Join aus, schicke Ergebnis nach St_{Result}
- Transfer beider Relationen: schicke beide Relationen zu St_{Result} und führe dort den Join aus

Auswertung mit Filterung

- Ohne Filterung müssen große Datenmengen über das Netz, obwohl Ergebnis eventuell sehr klein
- Idee: verschicke nur Tupel, die auch Joinpartner finden
- Folgende Eigenschaften werden dabei genutzt
  (C ist Joinattribut):

$$R \bowtie S = R \bowtie (R \rtimes S)$$
$$R \rtimes S = \Pi_C(R) \rtimes S$$

Beispiel


15 Attributwerte
...
St_{Result}
R\join(\Pi_C(R)\rsemijoin 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_4 b_5 c_3 d_2 e_2
\Pi_C(R)\rsemijoin S
C D E
c_1 d_1 e_1
c_3 d_2 e_1
6 Attributwerte
\join
C
c_1
c_2
c_3
c_6
4 Attributwerte
\rsemijoin
\Pi_c
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
St_R
St_S

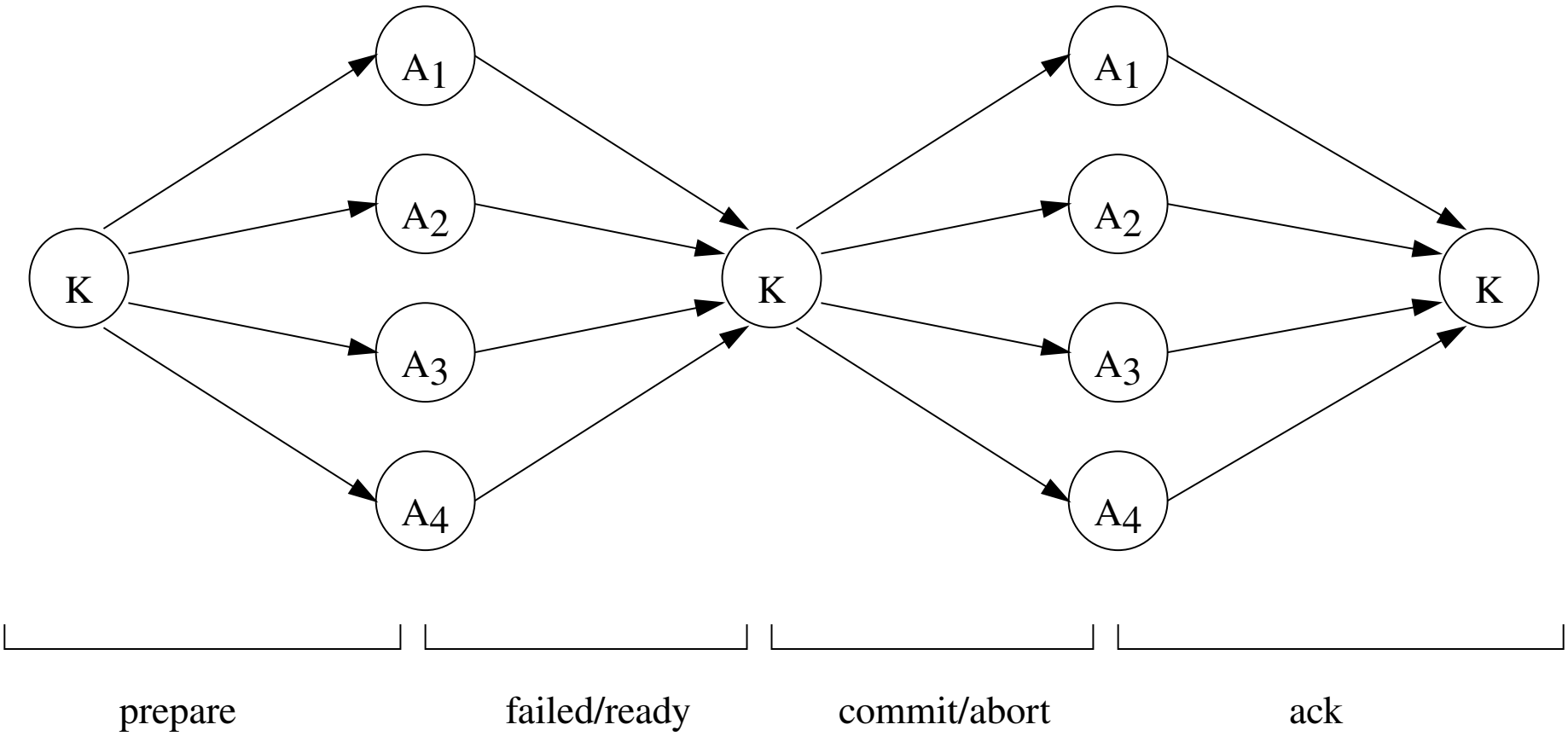
Transaktionskontrolle

- Transaktionen können sich über mehrere Rechnerknoten erstrecken
- Alle Stationen schreiben lokale Protokolleinträge über ausgeführte Operationen
- Wird beim Wiederanlauf benötigt, um Daten einer abgestürzten Station zu rekonstruieren

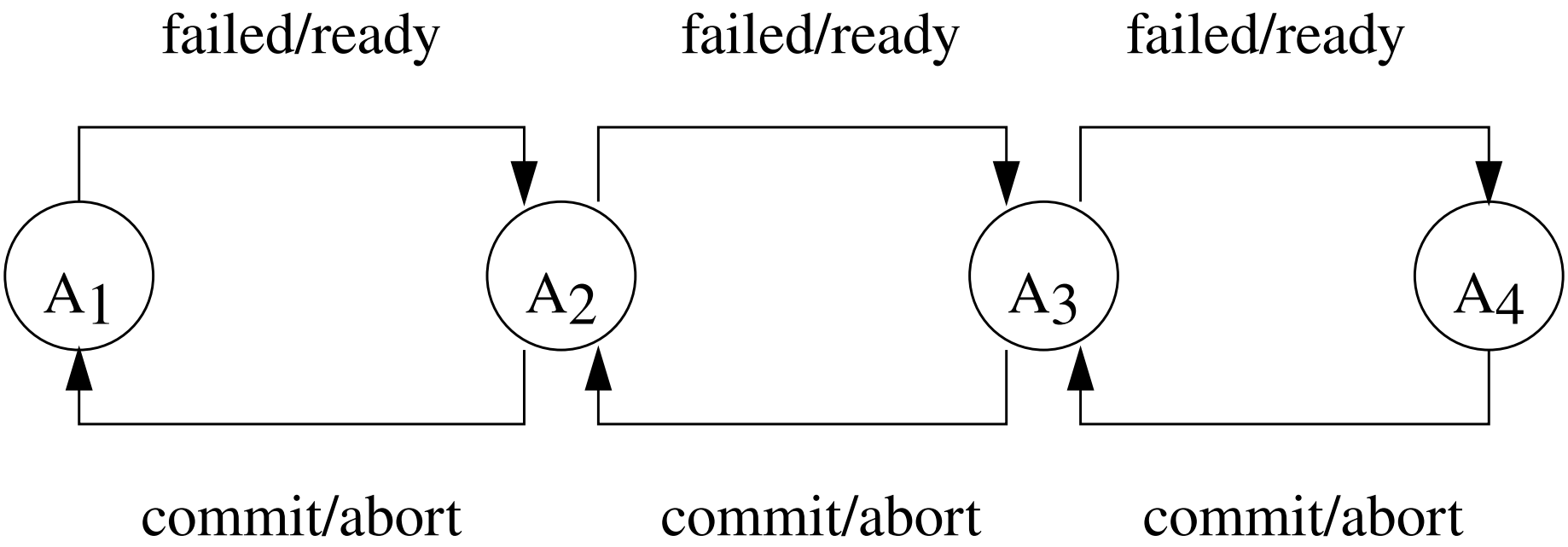
Transaktionskontrolle(2)

- Abort: bei einem Abbruch einer globalen Transaktion müssen alle lokalen Teile zurückgesetzt werden
- Commit: prinzipielle Schwierigkeit beim Beenden:
  - Atomare Beendigung der Transaktion muss gewährleistet sein

Two-Phase Commit (2PC)



K=Koordinator, A=Agent


failed/ready
failed/ready
failed/ready
A1
A2
A3
A4
commit/abort
commit/abort
commit/abort

Fehlerbehandlung

- Während des verteilten Commits kann es zu folgenden Fehlerfällen kommen:
  - Absturz des Koordinators
  - Absturz eines Agenten
  - Verlorene Nachricht
- Im schlimmsten Fall blockieren Knoten

- Lokale Serialisierbarkeit garantiert noch keine globale Serialisierbarkeit
- Beispiel:

S_1		
Schritt	T_1	T_2
1.	$r(A)$	
2.		$w(A)$

S_2		
Schritt	T_1	T_2
3.		$w(B)$
4.	$r(B)$	

- Reguläres 2PL reicht im verteilten Fall nicht aus
- Erst strenges 2PL garantiert Serialisierbarkeit
- Verwaltung der Sperren:
  - zentral
  - lokal

Zentrale Verwaltung

- Alle Transaktionen fordern Sperren auf einer dedizierten Station an
- Diese Station kann leicht zum “Bottleneck” werden
- Außerdem verstößt dieses Verfahren gegen lokale Autonomie der Stationen
- Deswegen wird dieses Verfahren nicht angewendet

Lokale Verwaltung

- Globale Transaktionen (TAs, die auf mehr als einer Station laufen) müssen sich vor Modifikation eines Datenelements die Sperre vom lokalen Sperrverwalter holen
- Lokale Transaktionen müssen nur mit ihrem eigenen Verwalter kommunizieren
- Erkennung von Deadlocks ist allerdings schwieriger als bei der zentralen Verwaltung

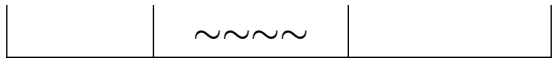
Deadlocks

- Eine lokale Deadlockerkennung reicht nicht:

S_1		
Schritt	T_1	T_2
0.	**BOT**	
1.	**lockS**(A)	
2.	$r(A)$	
6.		**lockX**(A)
		$\sim\sim\sim\sim$

S_2		
Schritt	T_1	T_2
3.		**BOT**
4.		**lockX**(B)
5.		$w(B)$
7.	**lockS**(B)	

Deadlocks



Erkennung von Deadlocks

- Timeouts: nach Verstreichen eines Zeitintervalls wird TA zurückgesetzt (Wahl des Intervalls kritisch)
- Zentralisierte Deadlockerkennung: ein Knoten baut einen zentralen Wartegraphen (hoher Aufwand, Phantomdeadlocks)
- Dezentrale Deadlockerkennung: Lokale Wartegraphen + spezieller Knoten *External*

Dezentrale Erkennung

- Jeder TA wird ein Heimatknoten zugeordnet (i.A. dort, wo TA begonnen wurde)
- Eine TA kann externe Subtransaktionen auf anderen Stationen starten
- Im Deadlockbeispiel ist S_1 Heimat von T_1 und S_2 Heimat von T_2

Dezentrale Erkennung(2)

- Für eine externe Subtransaktion T_i wird folgende Kante eingeführt:

$$External \rightarrow T_i$$

- Auf einer anderen Station wird auf Fertigstellung von T_i gewartet (nämlich von der TA, die die externe Subtransaktion initiiert hat)
- Für eine TA T_j, die eine Subtransaktion initiiert, die Kante

$$T_j \rightarrow External$$

- T_j wartet auf Fertigstellung der auf einer anderen Station angestoßenen Subtransaktion

Dezentrale Erkennung(3)

- Für unser Beispiel bedeutet dies

$$S_1 : \boxed{External \rightarrow T_2 \rightarrow T_1 \rightarrow External}$$

$$S_2 : \boxed{External \rightarrow T_1 \rightarrow T_2 \rightarrow External}$$

- Ein Zyklus, der *External* enthält, ist nicht notwendigerweise ein Deadlock
- Zur Feststellung eines Deadlocks müssen Stationen Informationen austauschen

Dezentrale Erkennung(4)

- Station mit lokalem Wartegraph

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

  schickt ihren lokalen Graphen an die Station, wo T_n' eine Subtransaktion angestoßen hat
- Für unser Beispiel:

$$S_2: \boxed{External \leftrightarrow T_1 \leftrightarrow T_2 \leftrightarrow External}$$

$$T_1 \rightarrow T_2 \rightarrow T_1$$

$$T_2 \rightarrow T_1 \rightarrow T_2$$

Dezentrale Erkennung(5)

- Um redundante Nachrichten zu vermeiden (im obigen Beispiel schickt S_1 Informationen an S_2 und umgekehrt), wird nicht immer Graph verschickt
- Bei (lokalem) Wartegraph

$$External \rightarrow T'_1 \rightarrow T'_2 \rightarrow ... \rightarrow T'_n \rightarrow External$$

wird Information nur verschickt, wenn TA-Identifikator von T'_n größer ist als TA-Identifikator von T'_1

Deadlockvermeidung

- Es gibt Verfahren, die Zeitstempel einsetzen, um Deadlocks zu vermeiden
  - Zeitstempelbasierte Synchronisation
  - Deadlockvermeidung bei sperrbasierten Verfahren: wound/wait, wait/die
- Setzt voraus, dass global eindeutige Zeitstempel generiert werden können

Zeitstempelgenerierung

- Gängigste Methode:

Stations-ID	lokale Zeit

- Die Stations-ID muss in den niedrigstwertigen Bits stehen
- Ansonsten würden immer TAs bestimmter Stationen bevorzugt
- Außerdem sollten Uhren nicht zu weit voneinander abweichen

Synchronisation bei Replikation

- Was ist, wenn es mehrere Kopien eines Datenelements gibt?
- Wenn immer nur gelesen wird, ist dies unproblematisch
- Es reicht, irgendeine Kopie zu lesen
- Problematisch wird es bei Änderungen

Write All/Read Any

- Bei einer Änderungsoperation müssen alle Kopien angepasst werden
- Favorisiert Leseoperationen, hier muss nur eine Kopie gelesen werden
- Bei Ausfall einer Kopie können Änderungsoperationen nicht mehr ausgeführt werden bzw. werden verzögert

- Idee: Kopien bekommen Gewichte (je nach Robustheit und Leistung der Station)
- Es reicht, Kopien mit einem bestimmten Gesamtgewicht einzusammeln

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

Quorum-Consensus(2)

$W(A) = \sum \texttt{limits}_{i=1}^{4} w_i(A) = 8.$

Lesequorum $Q_r(A)$

Schreibquorum $Q_w(A)$

- $Q_w(A) + Q_w(A) > W(A)$ und
- $Q_r(A) + Q_w(A) > W(A)$.

Beispiel:

- $Q_r(A) = 4$
- $Q_w(A) = 5$

Änderungsoperation

- Vor dem Schreiben:

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

- Nach dem Schreiben:

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

Zusammenfassung

- In verteilten Datenbanksystemen werden die Daten auf räumlich (weit) getrennte Rechner verteilt
- Durch die Verteilung der Daten werden einige der üblich verwendeten Mechanismen in DBMS wesentlich komplizierter