Verteilte Datenbanken

VL Datenbanksysteme

Ingo Feinerer

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

Grundbegriffe

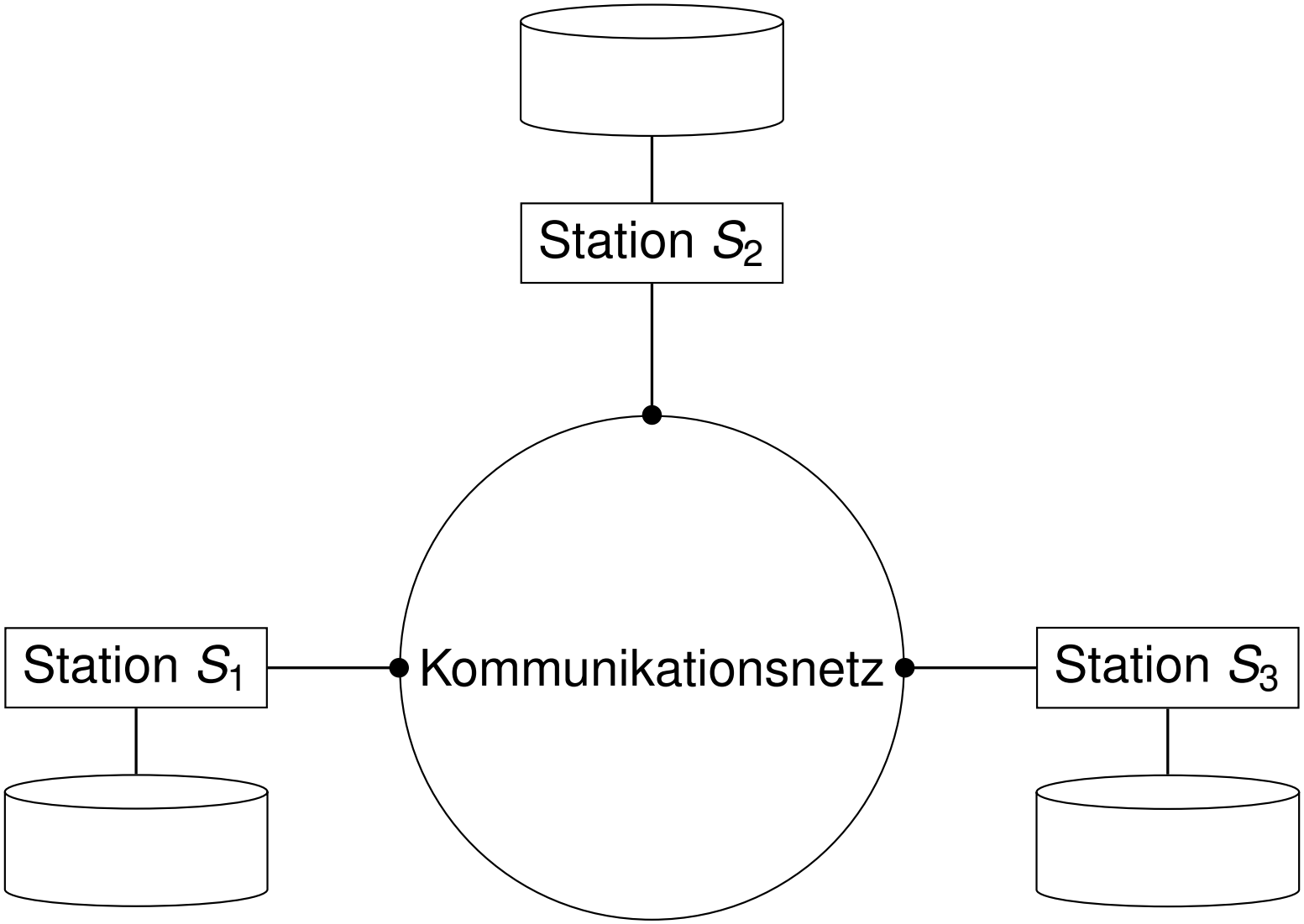
- VDBMS
- Entwurf eines VDBMS
- Transparenz

Verteiltes Datenbanksystem

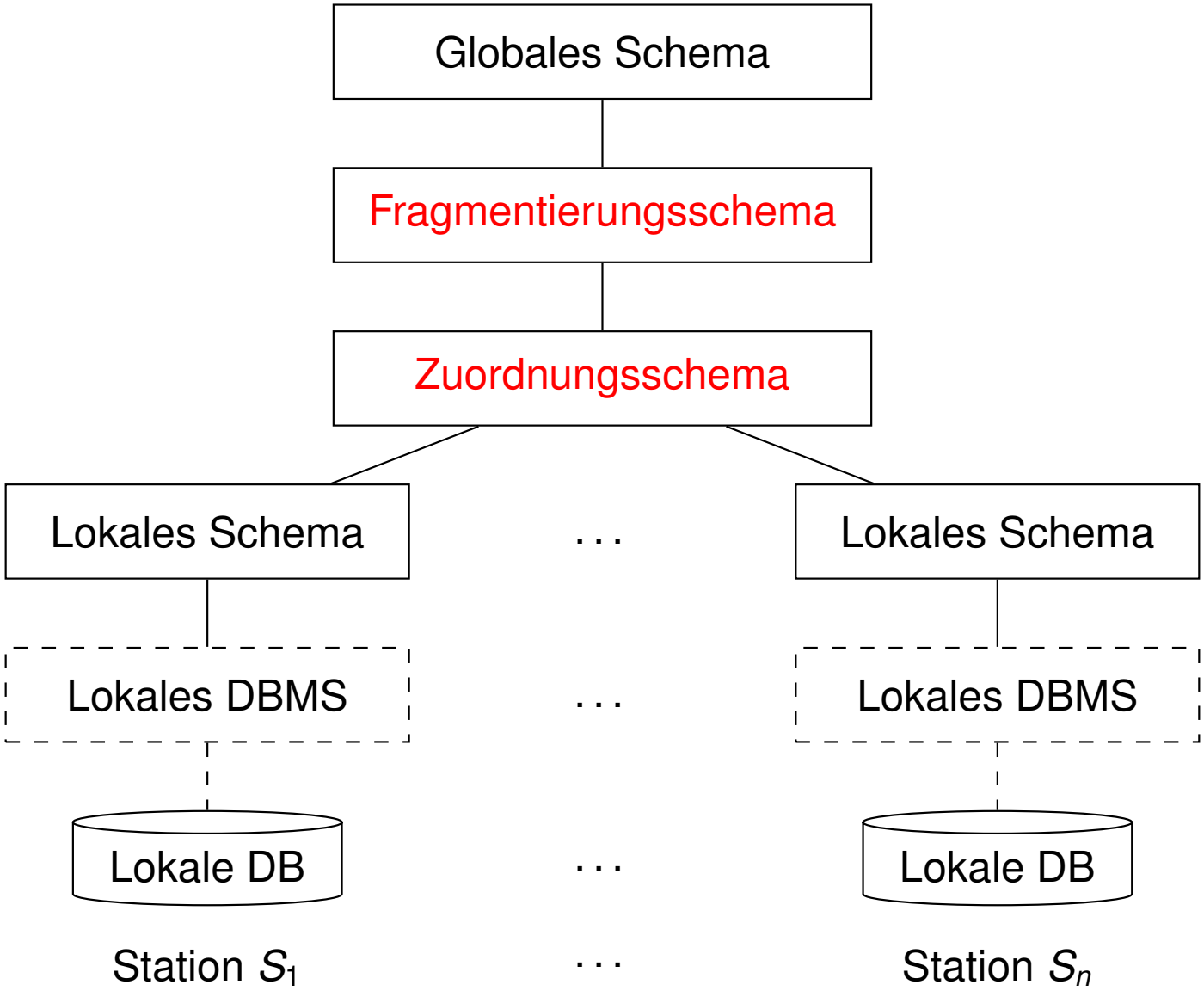
Merkmale eines VDBMS:

- (Globale) Gesamtinformation ist auf mehrere (lokale) Stationen verteilt.
- Stationen sind mittels Kommunikationsnetz verbunden (LAN, WAN, . . .)
- Die einzelnen Stationen sind „gleichberechtigt“ (sonst Client-Server Architektur)
- Homogenes DBMS und Datenmodell an den einzelnen Stationen (sonst „Multi-DB-System“)

Verteiltes Datenbanksystem



Entwurf eines VDBMS

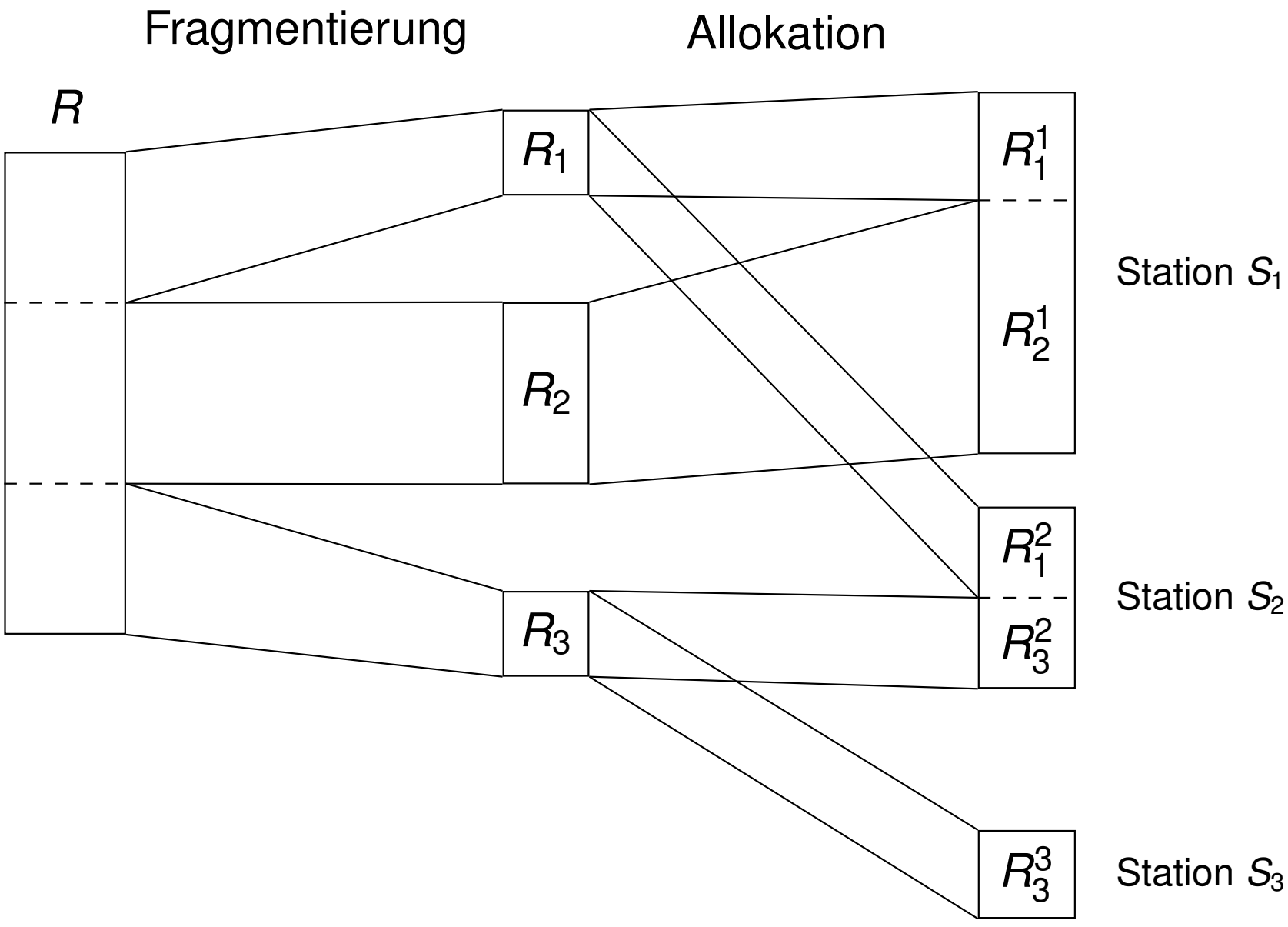


Entwurf eines VDBMS

Zusätzlich zum relationalen Implementierungsschema (= globales Schema) sind bei einem VDBMS folgende Entwurfsschritte erforderlich:

- Fragmentierung: Zerlegung der Relationen in (disjunkte) Teile
  - horizontal: Fragmentierung nach Zeilen
  - vertikal: Fragmentierung nach Spalten
  - kombiniert
- Allokation: Verteilung der Fragmente auf die Stationen
  - ohne Replikation (Fragmente zu Stationen: $N : 1$)
  - mit Replikation (Fragmente zu Stationen: $N : M$)

Fragmentierung und Allokation



Transparenz in verteilten Datenbanken

- Transparenz: Grad der Unabhängigkeit, den ein VDBMS dem Benutzer beim Zugriff auf verteilte Daten vermittelt.
- Stufen der Transparenz:

  Fragmentierungstransparenz Benutzer sieht nur das globale Schema (keine Fragmentierung, keine Allokation)

  Allokationstransparenz Benutzer sieht die Fragmente aber nicht deren Allokation

  Lokale Schema-Transparenz Benutzer sieht Fragmente und Allokation. Aber er kann sich zumindest auf ein einheitliches Datenmodell verlassen.

Fragmentierung

- Korrektheitsanforderungen
- Horizontale Fragmentierung
- Vertikale Fragmentierung
- Allokation

Korrektheitsanforderungen

Rekonstruierbarkeit Die ursprüngliche Relation R lässt sich aus den Fragmenten $R_1, \ldots, R_n$ (mit Hilfe von relationaler Algebra) wiederherstellen.

Vollständigkeit Jedes Datum ist mindestens einem Fragment zugeordnet.

Disjunktheit Jedes Datum ist höchstens einem Fragment zugeordnet.

Bemerkung:

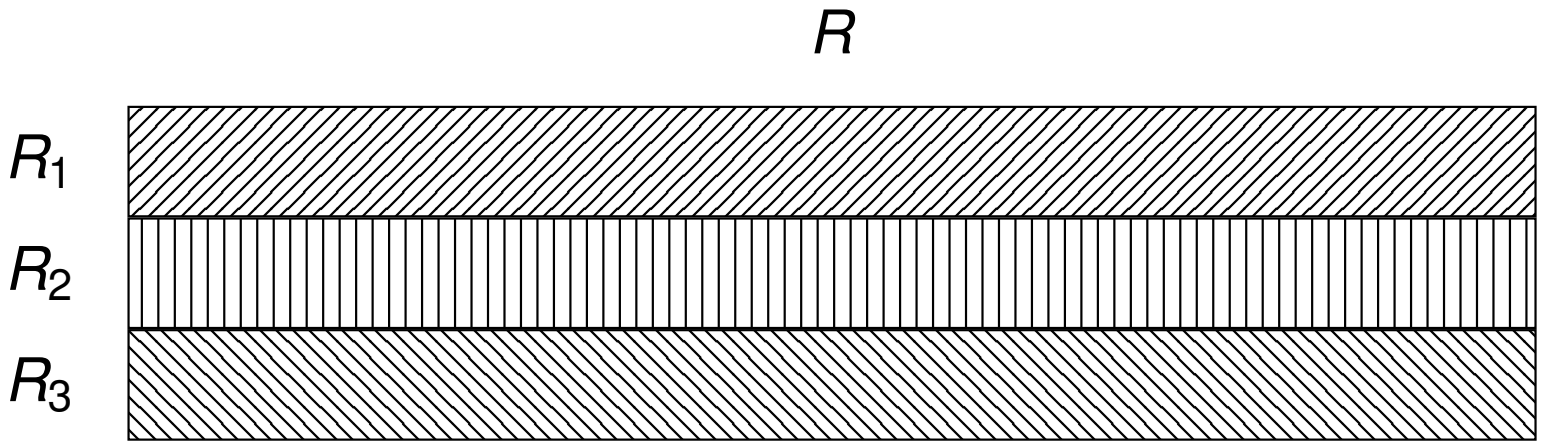
- Rekonstruierbarkeit setzt Vollständigkeit voraus.
- Disjunktheit darf eventuell verletzt werden (z.B. mehrfaches Ablegen von Attributen bei vertikaler Fragmentierung).

Relation 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

Horizontale Fragmentierung

Abstrakte Darstellung:



Für 2 Prädikate p_1 und p_2 ergeben sich 4 Fragmente:

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

Für n Prädikate $p_1, \ldots, p_n$ ergeben sich 2^n Fragmente.

Beispiel

Gruppierung der Professoren nach Fakultätszugehörigkeit, d.h. 3 Zerlegungsprädikate:

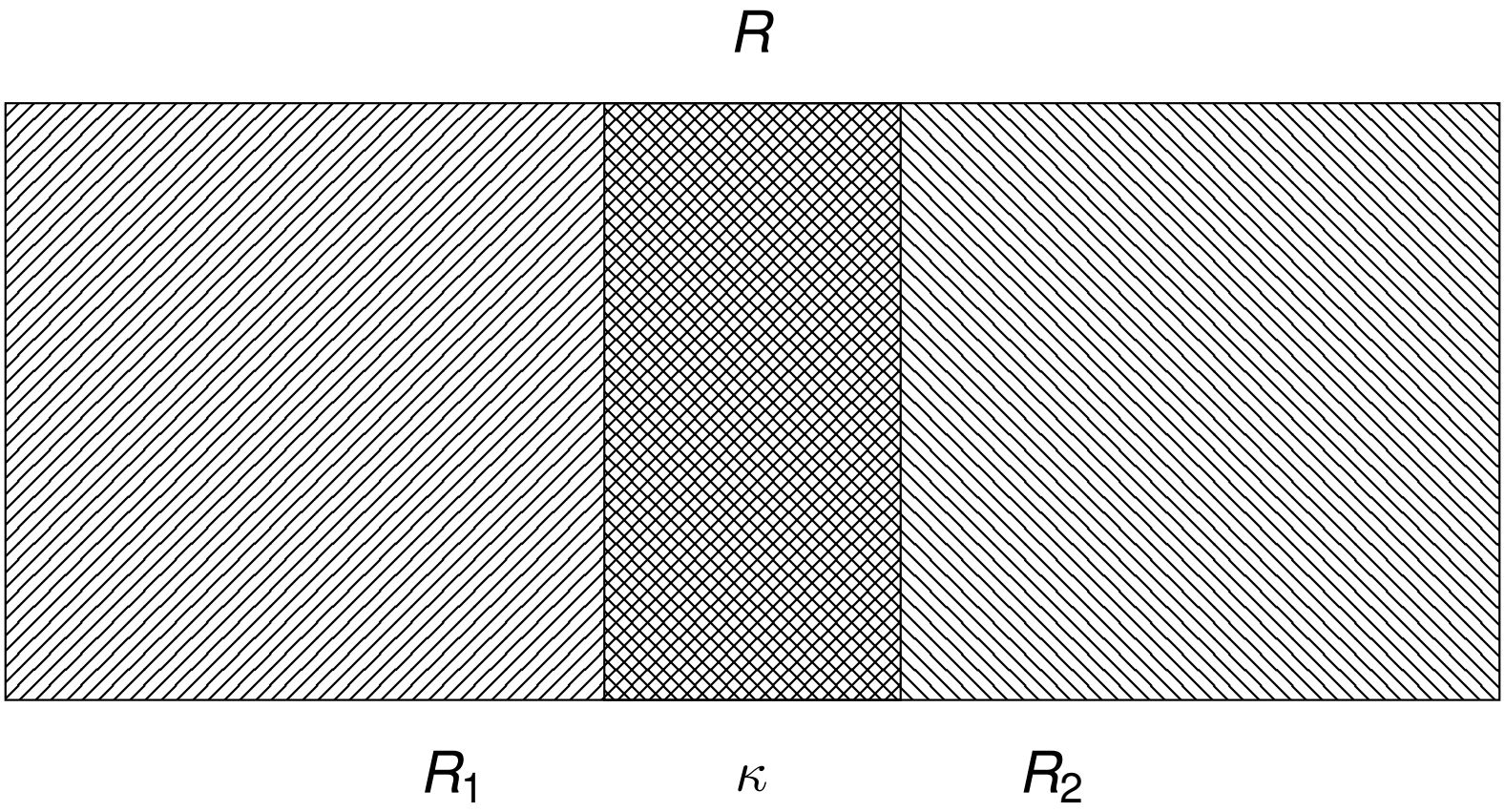
$$p_1 : \text{Fakultät} = \text{‘Theologie’}$$
$$p_2 : \text{Fakultät} = \text{‘Physik’}$$
$$p_3 : \text{Fakultät} = \text{‘Philosophie’}$$

$$\begin{aligned}
\text{TheolProfs} &= \sigma_{p_1 \wedge \neg p2 \wedge \neg p_3}(\text{Professoren}) = \sigma_{p_1}(\text{Professoren}) \\
\text{PhysikProfs} &= \sigma_{\neg p_1 \wedge p2 \wedge \neg p_3}(\text{Professoren}) = \sigma_{p_2}(\text{Professoren}) \\
\text{PhiloProfs} &= \sigma_{\neg p_1 \wedge \neg p2 \wedge p_3}(\text{Professoren}) = \sigma_{p_3}(\text{Professoren}) \\
\text{AndereProfs} &= \sigma_{\neg p_1 \wedge \neg p2 \wedge \neg p_3}(\text{Professoren})
\end{aligned}$$

Alle anderen Selektionsprädikate ergeben leere Tabellen, z.B. $\sigma_{p_1 \wedge p_2 \wedge \neg p_3}(\text{Professoren})$.

Vertikale Fragmentierung

Abstrakte Darstellung:



Vertikale Fragmentierung

- Beliebige vertikale Fragmentierung gewährleistet keine Rekonstruierbarkeit!
- Zwei mögliche Ansätze, um Rekonstruierbarkeit zu garantieren:
  - Jedes Fragment enthält den *Primärschlüssel* der Originalrelation (auch wenn dadurch die Disjunktheit verletzt wird).
  - Jedem Tupel der Originalrelation wird ein eindeutiges *Surrogat* (= künstlich erzeugte Objekt-ID) zugeordnet, welches in jedes vertikale Fragment des Tupels mit aufgenommen wird.

Vertikale Fragmentierung

Beispiel

- Für die Universitätsverwaltung sind PersNr, Name, Gehalt und Steuerklasse interessant:

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

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

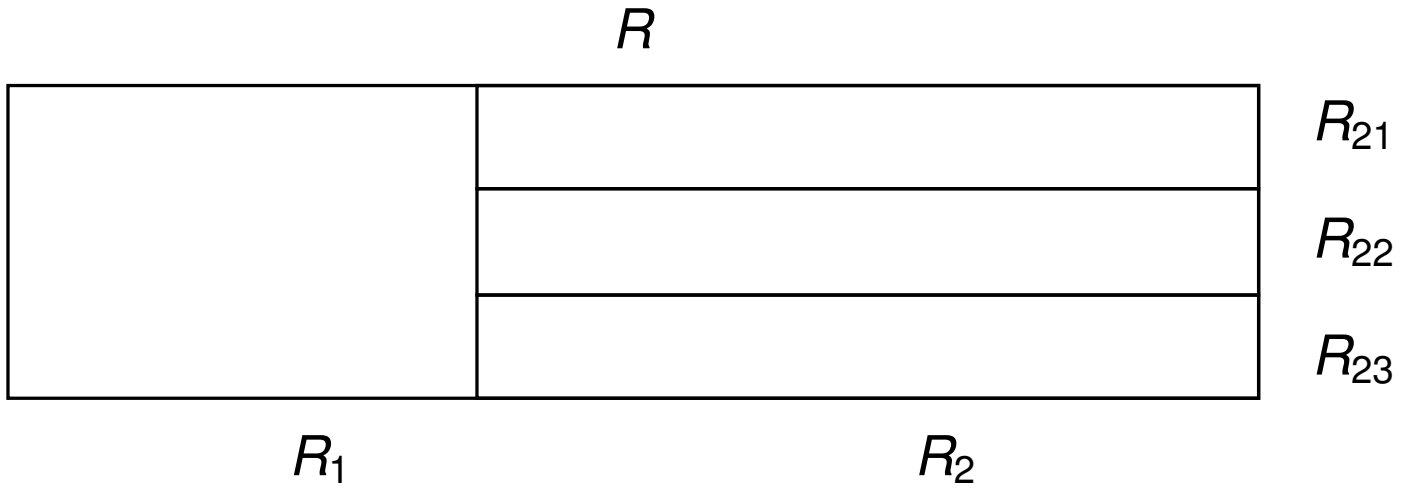
$$\text{Profs} = \pi_{\text{PersNr,Name,Rang,Raum,Fakultät}}(\text{Professoren})$$

- Rekonstruktion der Originalrelation Professoren:

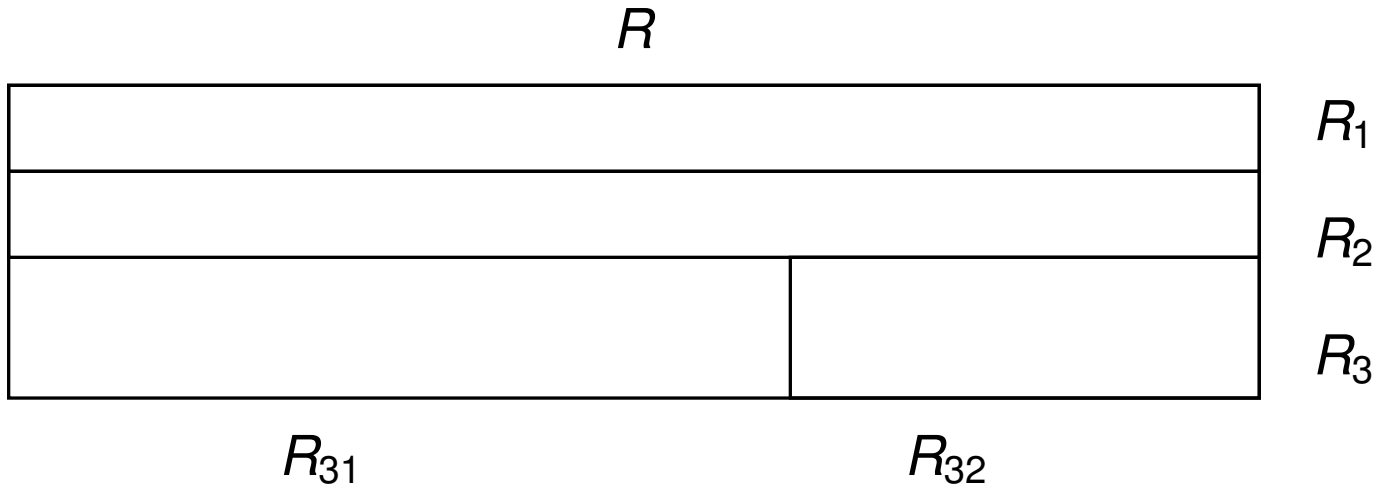
$$\text{Professoren} = \text{ProfVerw} \bowtie_{\text{ProfVerw.PersNr=Profs.PersNr}} \text{Profs}$$

Kombinierte Fragmentierung

- Horizontale Fragmentierung nach vertikaler Fragmentierung



- Vertikale Fragmentierung nach horizontaler Fragmentierung



Rekonstruktion nach kombinierter Fragmentierung

- Horizontale Fragmentierung nach vertikaler Fragmentierung

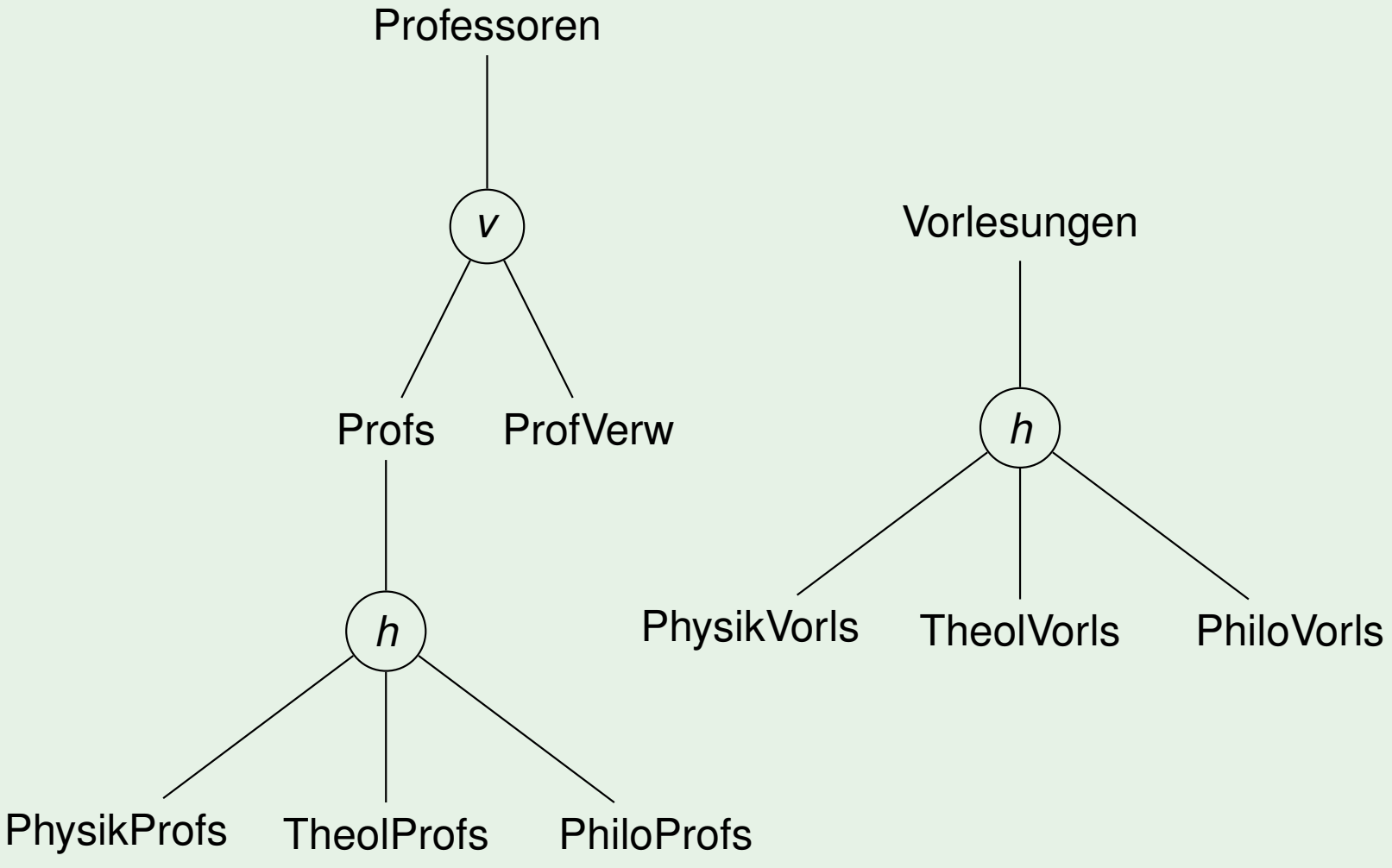
$$R = R_1 \bowtie_p (R_{21} \cup R_{22} \cup R_{23})$$

- Vertikale Fragmentierung nach horizontaler Fragmentierung

$$R = R_1 \cup R_2 \cup (R_{31} \bowtie_p R_{32})$$

Baumdarstellung der Fragmentierung

Beispiel



Allokation

- Allokation ohne Replikation (= redundanzfreie Allokation): Jedes Fragment wird exakt einer Station zugeordnet.
- Allokation mit Replikation: Fragmente können mehreren Stationen zugeordnet werden.

Beispiel

Allokation 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	$\{TheolVorls, TheolProfs\}$

Anfragebearbeitung

- Anfrageübersetzung und -optimierung
- bei horizontaler Fragmentierung
- bei vertikaler Fragmentierung

Anfrageübersetzung und Anfrageoptimierung

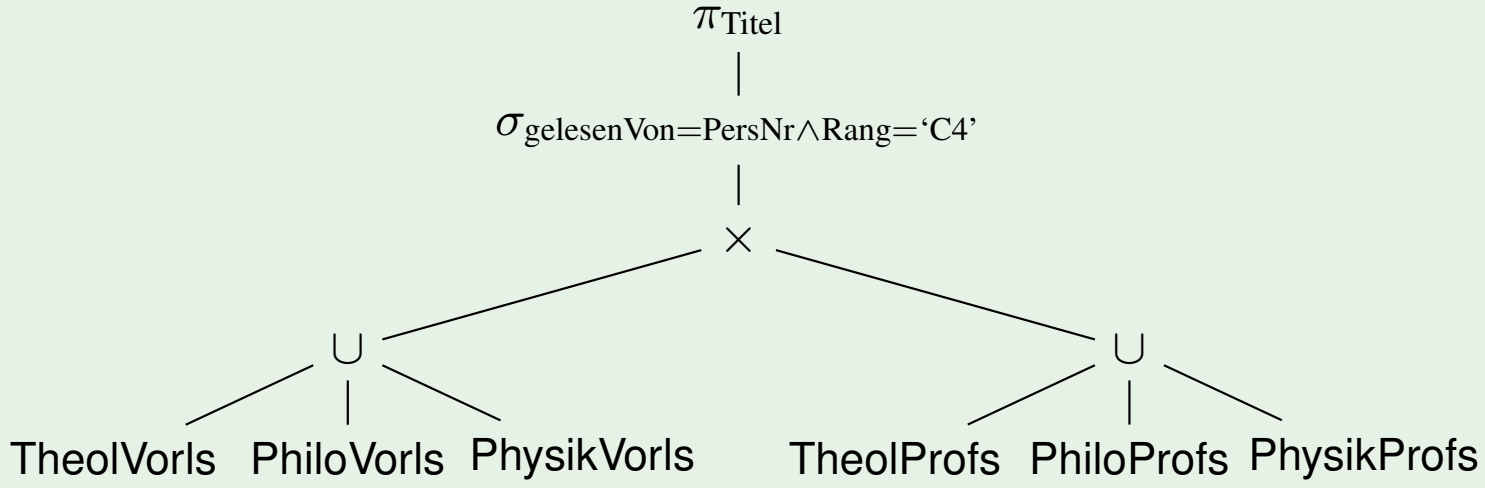
- Aufgabe des Anfrageübersetzers: Generierung eines Anfrageauswertungsplans auf den Fragmenten
- Aufgabe des Anfrageoptimierers: Generierung eines möglichst effizienten Auswertungsplanes:
  - Abhängig von der Allokation der Fragmente auf die verschiedenen Stationen des Rechnernetzes
  - Transferkosten für Datenkommunikation berücksichtigen (Verbindungsaufbau und vom Datenvolumen abhängige Kosten)
  - Auslastung der einzelnen Stationen berücksichtigen

Anfragebearbeitung bei horizontaler Fragmentierung

Beispiel

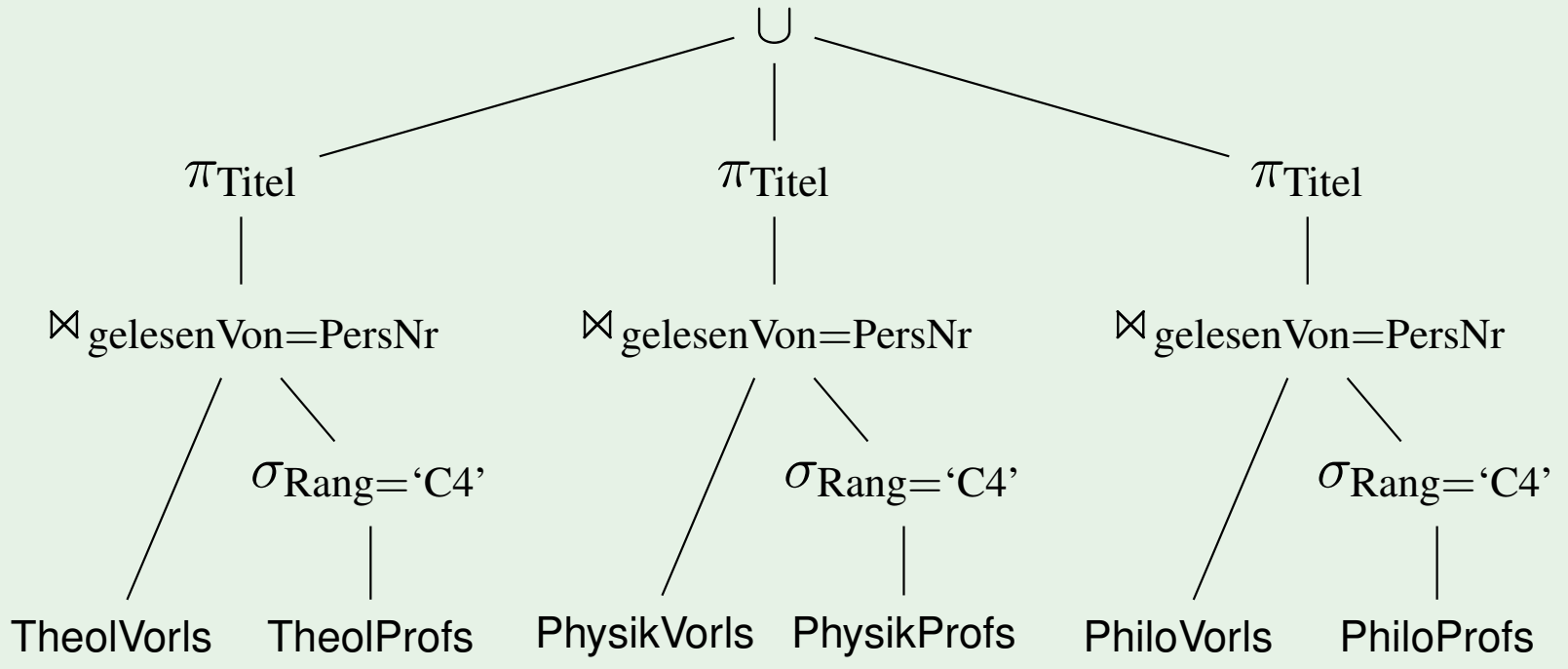
```
SELECT Titel
FROM Vorlesungen, Professoren
WHERE gelesenVon = PersNr AND Rang = 'C4'
```

Kanonische Übersetzung:



Optimierter Ausdruck

Beispiel

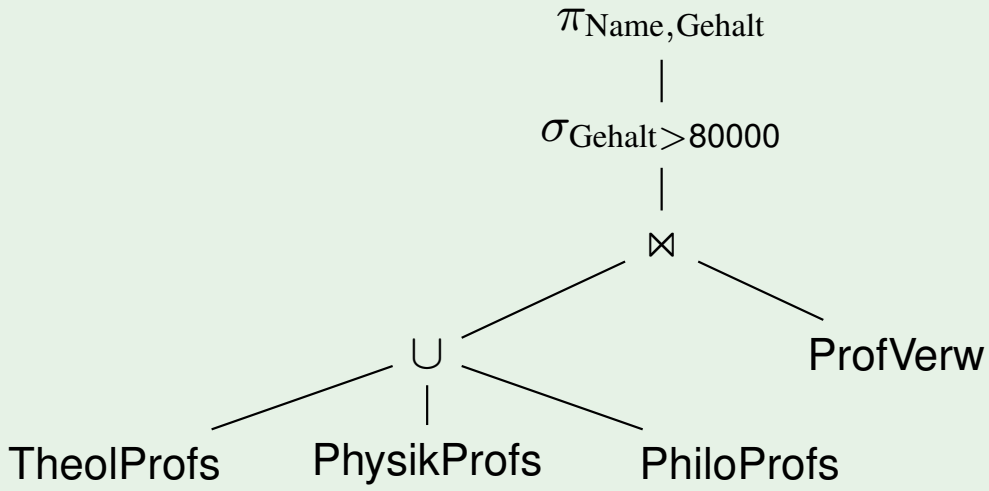


Anfragebearbeitung bei vertikaler Fragmentierung

Beispiel

```
SELECT Name, Gehalt
FROM Professoren
WHERE Gehalt > 80000
```

Kanonische Übersetzung:



Optimierter Ausdruck

Beispiel

In diesem speziellen Beispiel gilt, dass alle notwendigen Informationen in der Tabelle *ProfVerw* enthalten sind.
Optimierter Auswertungsplan:

$$\pi_{\text{Name,Gehalt}}$$
$$|$$
$$\sigma_{\text{Gehalt}>80000}$$
$$|$$
ProfVerw

Beispiel

Eine schlecht zu optimierende Anfrage (Attribut *Rang* fehlt in ProfVerw):

```
SELECT Name, Gehalt, Rang
FROM Professoren
WHERE Gehalt > 80000
```

Transaktionskontrolle

- Globable Transaktionen
- Zweiphasen-Commit-Protokoll (2PC)
- Fehlerbehandlung und Recovery

Globale Transaktionen

Definition

Globale Transaktionen sind Transaktionen im VDBMS, die sich über mehrere Stationen erstrecken.

- Anforderung: ACID-Eigenschaft muss auch für eine globale Transaktion gelten.
- Spezielles Problem: Atomicity.
  EOT-Behandlung von globalen Transaktionen:
  - **commit**: globale Transaktion wird an allen (relevanten) lokalen Stationen festgeschrieben.
  - **abort**: globale Transaktion wird an keiner einzigen Station festgeschrieben.

Zweiphasen-Commit-Protokoll (2PC)

Modell:

- **Koordinator** K: überwacht die EOT-Behandlung
- **Agenten** $A_1, \ldots, A_n$ (= Stationen des VDBMS, die an einer Transaktion beteiligt waren): schreiben alle Änderungen der Transaktion lokal fest oder rollen alle Änderungen der Transaktion lokal zurück.

Zwei Phasen:

- **1. Phase**: Koordinator verschafft sich einen Überblick über den Zustand der Agenten.
- **2. Phase**: Koordinator entscheidet „commit“ oder „abort“ und teilt seine Entscheidung allen Agenten mit, die diese Entscheidung dann lokal umsetzen.

Ablauf der EOT-Behandlung

1. Phase

- Koordinator K schickt allen Agenten eine **PREPARE**-Nachricht, um herauszufinden, ob sie die Transaktion festschreiben können.
- Jeder Agent A_i empfängt die PREPARE-Nachricht und schickt eine von zwei möglichen Nachrichten an K:
  - **READY**, falls A_i in der Lage ist, die Transaktion lokal festzuschreiben.
  - **FAILED**, falls A_i kein commit durchführen kann (wegen Fehler, Inkonsistenz, et cetera).

Ablauf der EOT-Behandlung

2. Phase

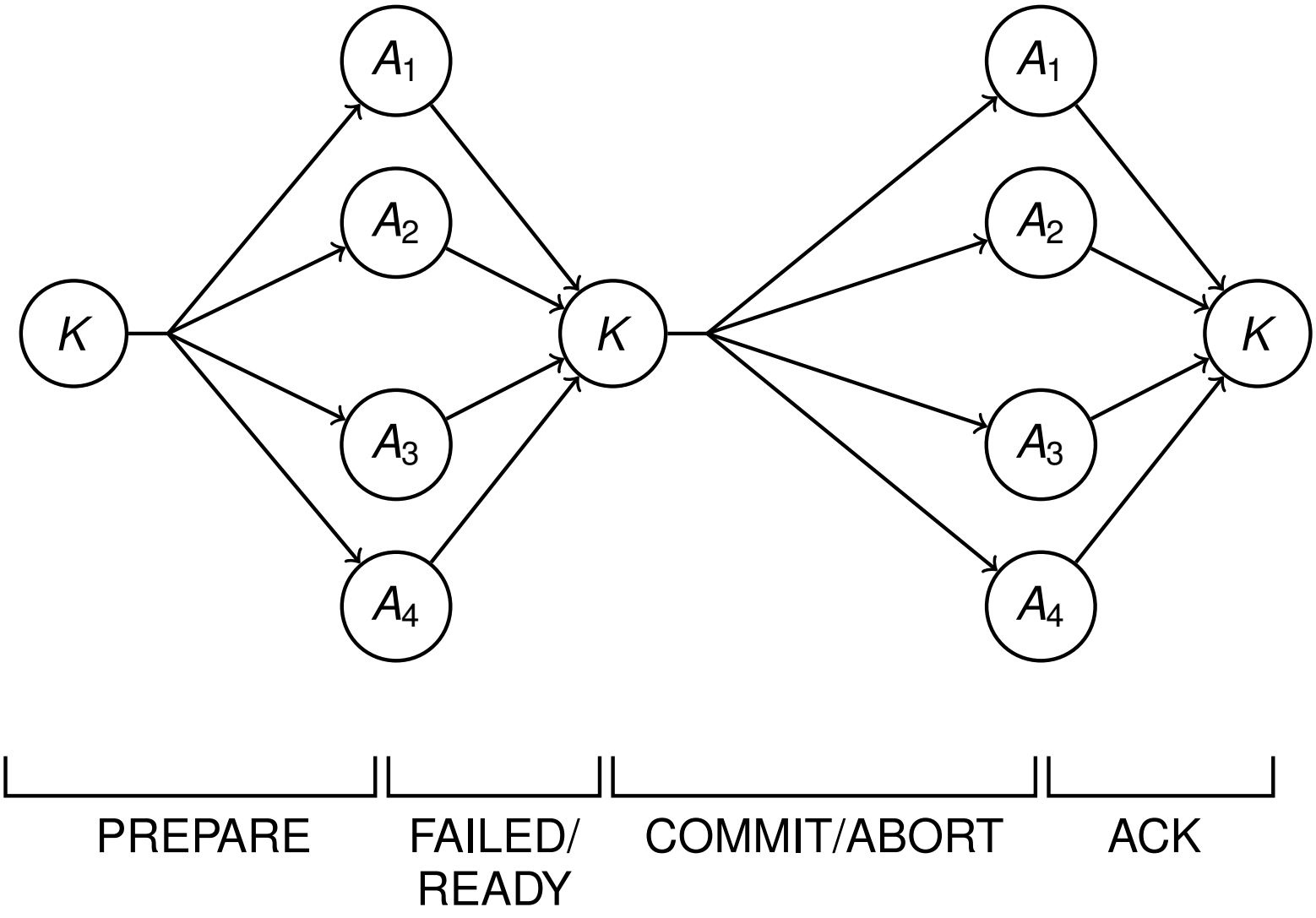
Abhängig von der Antwort der Agenten entscheidet *K* zwischen commit und abort:

- Fall 1: *K* hat von allen Agenten READY erhalten. Dann schickt *K* ein **COMMIT** an alle Agenten.
- Fallt 2: *K* hat von einem Agenten FAILED oder gar keine Antwort erhalten (timeout-Überwachung). Dann schickt *K* ein **ABORT** an alle Agenten.

Wenn die Agenten ihre lokale EOT-Behandlung abgeschlossen haben, schicken sie eine **ACK** Nachricht (= acknowledgement) an den Koordinator.

Nachrichtenaustausch beim 2PC-Protokoll

für 4 Agenten



Fehlerbehandlung und Recovery

Wie im nicht-verteilten Fall:

- Alle Stationen führen eine log-Datei mit den lokalen Änderungen.
- WAL-Prinzip
- Redo/Undo der lokalen Änderungen

Zusätzlich erforderlich:

- Auch die Kommunikation im Rahmen des 2PC muss in die log-Datei geschrieben werden, z.B.: Bevor der Koordinator K oder ein Agent A_i eine Nachricht schickt, trägt er diese in seine log-Datei ein.
- Beim Wiederanlauf ist eventuell ein Nachrichtenaustausch erforderlich, um commit/abort zu erkennen.

Typische Fehlersituationen des 2PC-Protokolls

- Absturz eines Koordinators
- Absturz eines Agenten
- Verlorengegangene Nachrichten

Absturz eines Koordinators

- Absturz vor dem Senden einer COMMIT-Nachricht: Rückrollen der Transaktion mittels ABORT-Nachricht.
- Absturz, nachdem ein Agent ein READY mitgeteilt hat:
  - Agent kann Transaktion nicht mehr eigenständig abbrechen.
  - Blockierung des Agenten (= Hauptproblem des 2PC-Protokolls)
  - Verfügbarkeit des Agenten stark eingeschränkt.

Absturz eines Agenten

Analyse der Log-Datei beim *Wiederanlauf des Agenten*:

- kein READY-Eintrag bzgl. Transaktion T: Agent führt ein abort durch und teilt dies dem Koordinator mittels FAILED-Nachricht mit.
- READY-Eintrag aber kein COMMIT-Eintrag: Agent fragt Koordinator, was aus Transaktion T geworden ist. Koordinator teilt COMMIT oder ABORT mit.
- COMMIT-Eintrag vorhanden: Agent weiß ohne Nachfragen, dass Transaktion T erfolgreich war.

Ausfallserkennung durch den Koordinator: mittels Timer-Überwachung.

Verlorengegangene Nachrichten

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

- Nach einem Timeout-Intervall geht Koordinator davon aus, dass der Agent nicht funktionsfähig ist.
- Koordinator sendet ABORT an alle Agenten.

COMMIT- bzw. ABORT-Nachricht des Koordinators geht verloren:

- Agent ist blockiert, bis COMMIT- oder ABORT-Nachricht vom Koordinator kommt.
- Agent schickt eine „Erinnerung“ an den Koordinator.

Mehrbenutzersynchronisation

- Serialisierbarkeit
- Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)
- Deadlocks

Serialisierbarkeit

- Definitionen von Serialisierbarkeit, Rücksetzbarkeit, etc. lassen sich einfach auf verteilten Fall übertragen.
- Bei globalen Transaktionen ist die lokale Serialisierbarkeit (d.h. Serialisierbarkeit auf den einzelnen Stationen) nicht ausreichend!

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)	

Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)

- Funktioniert nach dem selben Prinzip wie im nicht verteilten Fall, z.B.: Alle Sperren werden bis EOT gehalten.
- Schwierigkeit im verteilten Fall: *Sperrverwaltung*
  - **Lokale Sperrverwaltung**: globale Transaktion muss vor Zugriff auf ein Datum *A* auf Station *S* eine Sperre vom Sperrverwalter der Station *S* erwerben.
    Nachteil: Deadlock-Erkennung ist schwierig!
  - **Globale Sperrverwaltung**: alle Transaktionen fordern Sperren an einer einzigen, ausgezeichneten Station an.
    Nachteil: Zentraler Sperrverwalter als Engpass des VDBMS; bei seinem Absturz ist gesamtes VDBMS blockiert!

Deadlocks

- Globale Sperrverwaltung ist im allgemeinen nicht akzeptabel. Deshalb üblicherweise lokale Sperrverwaltung.
- Deadlockerkennung bei lokaler Sperrverwaltung ist wesentlich schwieriger als bei globaler Sperrverwaltung.

Beispiel

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)	
	~~	

Deadlockerkennung

3 Methoden:

- **Timeout**: Wenn eine Transaktion in einem bestimmten Zeitintervall keinen Fortschritt macht, wird sie zurückgesetzt.
  Problem: Richtige Wahl des Timeout-Intervalls.
- **Zentralisierte Deadlockerkennung**: Alle Stationen melden Wartebeziehungen an eine ausgezeichnete Station, die daraus einen globalen Wartegraphen erstellt.
  Nachteile:
  - Kommunikationsaufwand
  - Widerspricht VDBMS-Idee: Engpass, Probleme bei Absturz
  - „Phantom“-Deadlock bei Nachrichtenüberholung möglich

Deadlockerkennung

- **Dezentralisierte Deadlockerkennung**:
  - Jeder Transaktion wird eine „Heimatstation“ zugeordnet (d.h.: Station, wo die TA gestartet wurde).
  - Alle Stationen führen einen lokalen Wartegraphen.
  - Zusätzlicher Knoten „External“: Wenn T_i von der Station S_i Teile der Arbeit auf einer anderen Station S_j erledigt:
    - Kante $\text{External} \rightarrow T_i$ im Wartegraphen auf S_j
    - Kante $T_i \rightarrow \text{External}$ im Wartegraphen auf S_i
  - Wenn Wartegraph einen Zyklus ohne „External“-Knoten hat, liegt sicher ein Deadlock vor.
  - Wenn Wartegraph einen Zyklus mit „External“-Knoten hat, kann ein Deadlock vorliegen. Dann muss von anderen Stationen zusätzliche Information angefordert werden.

Deadlockvermeidung

Erweiterung des strengen 2PL-Protokolls mittels Zeitstempel-Verfahren (wie im nicht-verteilten Fall):

- wound-wait Strategie
- wait-die Strategie

Nicht-sperrbasierte Synchronisationsverfahren:

- Zeitstempel-Verfahren
- Optimistische Synchronisation

Voraussetzung für all diese Verfahren:

- Vergabe von **eindeutigen** Zeitstempeln (als Transaktions-IDs)
- Übliche Methode: $\text{Transaktions-ID} = \text{lokale Zeit} + \text{Stations-ID}$

Replizierte Daten

- Problemstellung
- Quorum-Consensus Verfahren

Problem bei replizierten Daten

- Zu einem Datum A gibt es mehrere Kopien $A_1, A_2, \ldots, A_n$, die auf unterschiedlichen Stationen liegen.
- Naheliegende Vorgangsweise („write all, read any“):
  - Eine Lesetransaktion erfordert nur eine Kopie.
  - Bei Änderungstransaktionen müssen aber alle bestehenden Kopien geändert werden.

  Nachteil dieser Lösung:
  - Hohe Laufzeit bei Änderungsoperationen
  - Verfügbarkeitsproblem: Wenn eine Station, die eine Kopie A_i enthält, nicht verfügbar ist, muss die gesamte Transaktion warten oder zurückgerollt werden.

Quorum-Consensus Verfahren

Idee:

- Jeder Kopie A_i eines replizierten Datums A wird ein Gewicht w_i zugeordnet. Gesamtgewicht $W(A)$.
- Zu jeder Kopie von A wird eine Versionsnummer abgespeichert. Der aktuelle Wert hat die höchste Nummer.
- Lesequorum $Q_r(A)$: Mindestgewicht, das beim Lesen von A erreicht werden muss, d.h. es müssen so viele Kopien von A gelesen werden, dass deren Gewicht $\geq Q_r(A)$ beträgt.
- Schreibquorum $Q_w(A)$: Mindestgewicht, das beim Schreiben von A erreicht werden muss.
- Folgende Bedingungen müssen gelten:

$$Q_w(A) + Q_w(A) > W(A) \quad \text{und} \quad Q_r(A) + Q_w(A) > W(A)$$

Quorum-Consensus Verfahren

Vorteile des Quorum-Consensus Verfahrens:

- Ausgleich der Leistungsfähigkeit zwischen Lese- und Änderungstransaktionen: teilweise Verlagerung des Overheads von den Änderungs- zu den Lesetransaktionen.
- Verfügbarkeitsproblem reduziert

Spezialfälle:

- „write all, read any“:

  $w_i = 1$ für alle i, $W(A) = n$, $Q_w(A) = n$, $Q_r(A) = 1$
- „majority consensus“:

  $w_i = 1$ für alle i, $W(A) = n$, $Q_w(A) = Q_r(A) = \lfloor n/2 \rfloor + 1$

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

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

$$Q_r(A) = 4$$

$$Q_w(A) = 5$$

Beispiel

Zustand vor dem Schreiben eines Schreibquorums:

Station	Kopie	Gewicht	Wert	Versionsnr.
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

Zustand nach dem Schreiben eines Schreibquorums:

Station	Kopie	Gewicht	Wert	Versionsnr.
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