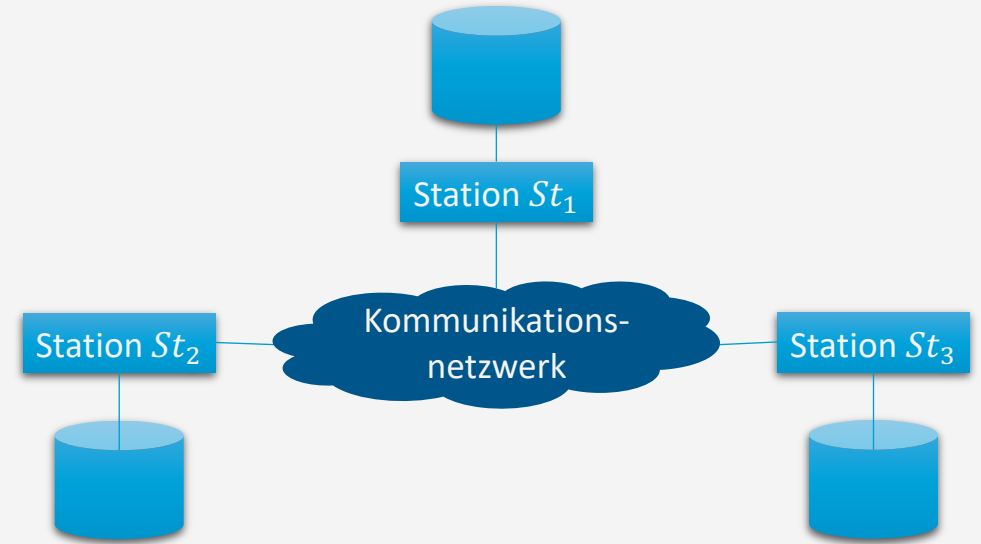


# Verteilte Datenbanken

Datenbanken



# Inhalte: Datenbanken (DBs)

## 1. Einführung

- Anwendungen
- Datenbankmanagementsysteme

## 2. Datenbank-Modellierung

- Entity-Relationship-Modell (ER-Modell)
- Beziehung zwischen ER und UML

## 3. Das relationale Modell

- Relationales Datenmodell (RM)
- Vom ER-Modell zum RM
- Relationale Algebra als Anfragesprache

## 4. Datenbank-Entwurf

- Funktionale Abhängigkeiten
- Normalformen

## 5. Structured Query Language (SQL)

- Datendefinition
- Datenmanipulation

## 6. Anfrageverarbeitung

- Architektur
- Indexierung
- Anfragepläne, Optimierung

## 7. Transaktionen

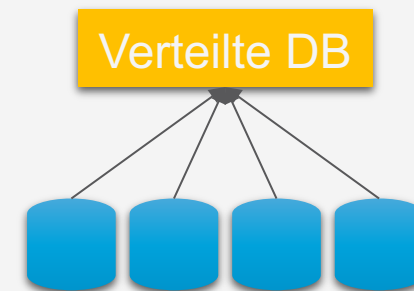
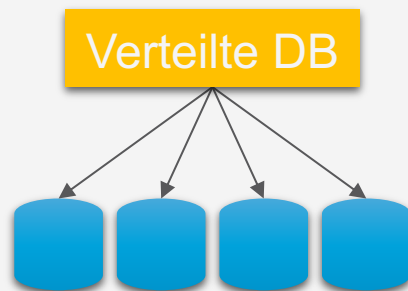
- Transaktionsverarbeitung, Schedules, Sperren
- Wiederherstellung

## 8. Erweiterung

- ~~Noch offen: verteilte DBs, deduktive DBs (DataLog  
⇒ Logik-Verbindung), XML, Graph-DBs~~

## Motivation für verteilte Datenbanken

- Verteilung / Dezentralisierung
  - Vor allem bei Neuentwicklungen
  - (Verteilte) Realisierung von Anwendungen auf Basis verteilter (DB-)Systeme
  - (Kontrolliert) verteilte Datenspeicherung zur Lastverteilung, Nutzung von Parallelität Verkürzung von Antwortzeiten, Erhöhung der Ausfallsicherheit
- Integration
  - Bei bestehenden Systemen
  - Verteilt gespeicherte und unabhängig voneinander verwaltete, aber inhaltlich zusammengehörige Daten logisch zusammenbringen
  - Einheitlichen Zugriff auf Gesamtdatenbestand ermöglichen



# Überblick: 8. Verteilte Datenbanken

## A. *Verteilte DBMS*

- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

## B. *Anfragenbeantwortung in verteilten Systemen*

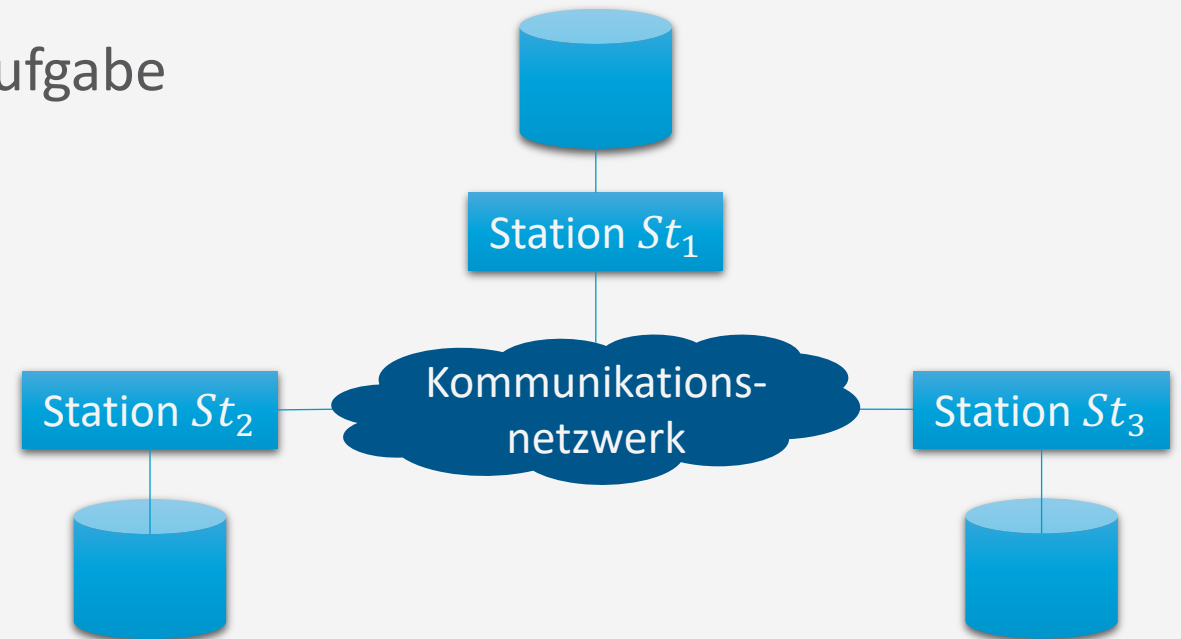
- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

## C. *Föderierte DBS*

- Integration
- Migration

## Verteiltes DBMS (VDBMS)

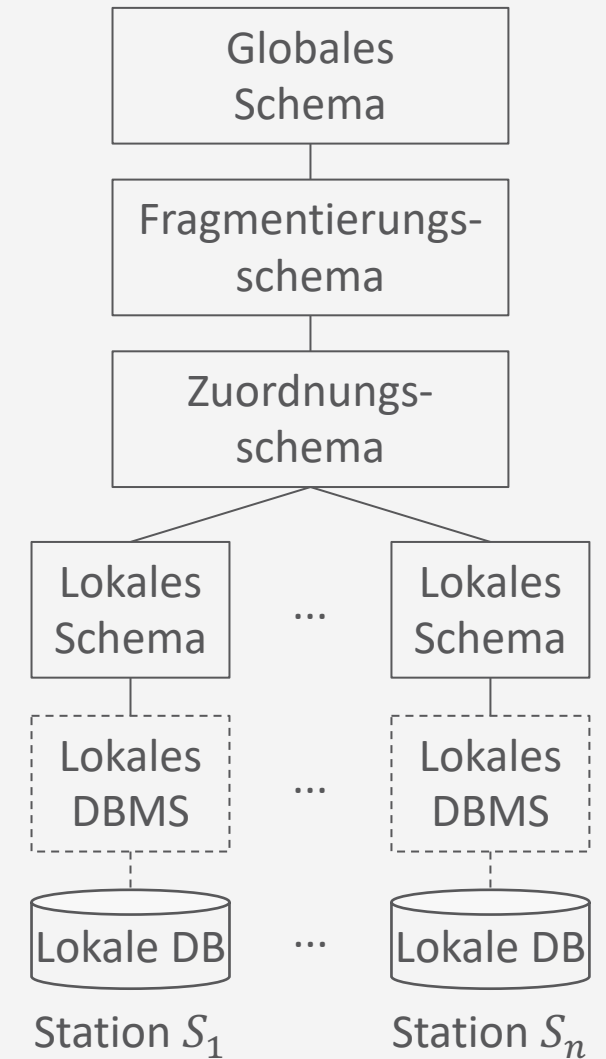
- Sammlung von Informationseinheiten, verteilt auf mehreren Rechnern, verbunden mittels Kommunikationsnetz (nach Ceri & Pelagatti, 1984)
- Kooperation zwischen autonom arbeitenden Stationen, zur Durchführung einer globalen Aufgabe



# Aufbau eines VDBMS

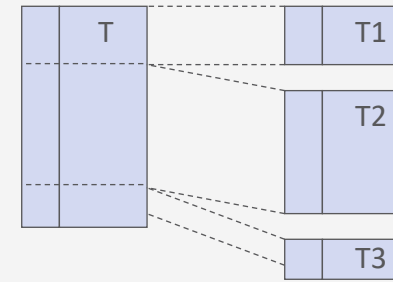
- Globales Schema
  - Relationales Schema wie zuvor
- **Fragmentierungsschema**
  - Verteilung der Daten in Fragmente
- Zuordnungsschema
  - Physische Zuordnung (**Allokation**) der Fragmente auf lokale DBs
  - Möglicherweise mit **Replikation**
- Lokales Schema / DBMS / DB
  - DB: Menge der lokal gespeicherten Daten
  - DBMS: Lokales System zur Verwaltung und Anfragebeantwortung
  - Schema: Lokale Sicht auf die Daten in lokaler DB

## Verteilte DBs

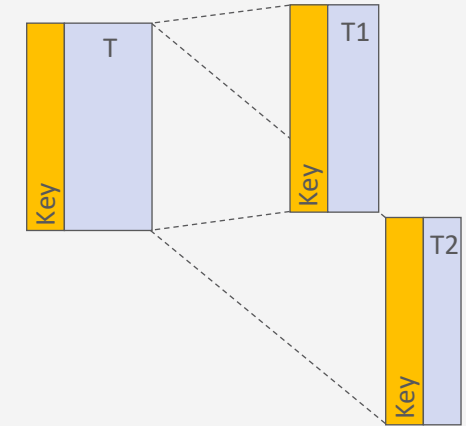


# Fragmentierung

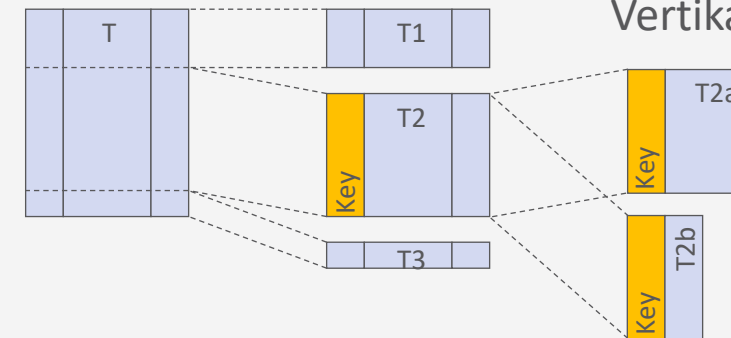
- Ziel: Verteilung von Daten
- Arten der Fragmentierung
  - **Horizontal:** Entlang der Tabellenzeilen
    - Aufteilung durch Selektion
  - **Vertikal:** Entlang der Tabellenspalten
    - Aufteilung durch Projektion
- Gemischt: Verschachtelung von Fragmentierung
  - Aufteilung durch verschachtelte Selektion und Projektion
- **Abgeleitet:** Anhand von Bedingungen
  - Z.B. über Fremdschlüsselrelation



Horizontale Fragmentierung



Vertikale Fragmentierung



Gemischte Fragmentierung

# Beispiel für horizontale Fragmentierung

$$\begin{aligned}
 Profs &= PhilProfs \\
 &\cup PhysProfs \\
 &\cup TheoProfs
 \end{aligned}$$

$$PhilProfs = \sigma_{Fakultaet='Philosophie'}(Profs)$$

$$TheoProfs = \sigma_{Fakultaet='Theologie'}(Profs)$$

$$PhysProfs = \sigma_{Fakultaet='Physik'}(Profs)$$

Profs	PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
	2125	Sokrates	W3	226	Philosophie	85000	1
	2126	Russel	W3	232	Philosophie	80000	3
	2127	Kopernikus	W2	310	Physik	65000	5
	2133	Popper	W2	52	Philosophie	68000	1
	2134	Augustinus	W2	309	Theologie	55000	5
	2136	Curie	W3	36	Physik	95000	3
	2137	Kant	W3	7	Philosophie	98000	1

PhilProfs	PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
	2125	Sokrates	W3	226	Philosophie	85000	1
	2126	Russel	W3	232	Philosophie	80000	3
	2133	Popper	W2	52	Philosophie	68000	1
	2137	Kant	W3	7	Philosophie	98000	1

TheoProfs	PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
	2134	Augustinus	W2	309	Theologie	55000	5

PhysProfs	PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
	2127	Kopernikus	W2	310	Physik	65000	5
	2136	Curie	W3	36	Physik	95000	3



# Beispiel für vertikale Fragmentierung

$$Prof_s = ProfVerw \bowtie Pfs$$

Prof_s	PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
	2125	Sokrates	W3	226	Philosophie	85000	1
	2126	Russel	W3	232	Philosophie	80000	3
	2127	Kopernikus	W2	310	Physik	65000	5
	2133	Popper	W2	52	Philosophie	68000	1
	2134	Augustinus	W2	309	Theologie	55000	5
	2136	Curie	W3	36	Physik	95000	3
	2137	Kant	W3	7	Philosophie	98000	1

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

ProfVerw	PersNr	Name	Gehalt	St.Klasse
	2125	Sokrates	85000	1
	2126	Russel	80000	3
	2127	Kopernikus	65000	5
	2133	Popper	68000	1
	2134	Augustinus	55000	5
	2136	Curie	95000	3
	2137	Kant	98000	1

$$Pfs = \pi_{PersNr, Name, Rang, Raum, Fakultaet}(Prof_s)$$

Pfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2127	Kopernikus	W2	310	Physik
	2133	Popper	W2	52	Philosophie
	2134	Augustinus	W2	309	Theologie
	2136	Curie	W3	36	Physik
	2137	Kant	W3	7	Philosophie

# Beispiel für gemischte Fragmentierung

$$Prof s = ProfVerw \bowtie (PhilPfs \cup PhysPfs \cup TheoPfs)$$

$$ProfVerw = \pi_{PersNr, Name, Gehalt, Steuerklasse}(Prof s)$$

$$Pfs = \pi_{PersNr, Name, Rang, Raum, Fakultaet}(Prof s)$$

$$PhilPrs = \sigma_{Fakultaet='Philosophie'}(Pfs)$$

$$TheoPfs = \sigma_{Fakultaet='Theologie'}(Pfs)$$

$$PhysPfs = \sigma_{Fakultaet='Physik'}(Pfs)$$

Pfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2127	Kopernikus	W2	310	Physik
	2133	Popper	W2	52	Philosophie
	2134	Augustinus	W2	309	Theologie
	2136	Curie	W3	36	Physik
	2137	Kant	W3	7	Philosophie

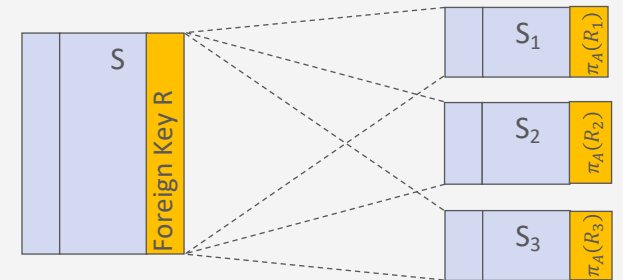
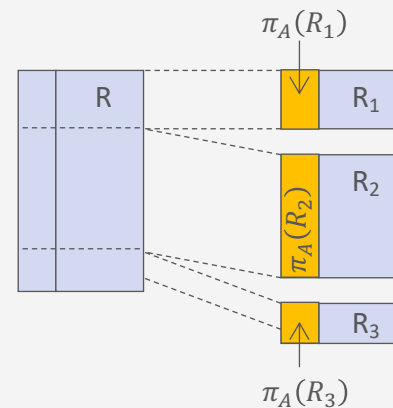
PhilPfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2133	Popper	W2	52	Philosophie
	2137	Kant	W3	7	Philosophie

TheoPfs	PersNr	Name	Rang	Raum	Fakultaet
	2134	Augustinus	W2	309	Theologie

PhysPfs	PersNr	Name	Rang	Raum	Fakultaet
	2127	Kopernikus	W2	310	Physik
	2136	Curie	W3	36	Physik

# Abgeleitete Fragmentierung über Fremdschlüsselbeziehung

- Relation  $S$  wird auf Basis der Fragmentierung der Relation  $R$  abgeleitet (fragmentiert)
  - Fremdschlüssel  $B$  in  $S$  auf Primärschlüssel  $A$  in  $R$
- $R$  durch Selektion horizontal in Fragmente  $R_i$  zerlegt
  - Referenzpartitionierung für  $S$
- $S$  aufgrund Fremdschlüsselbeziehungen partitioniert
  - $S_i = S \bowtie_{S.B=A} (\pi_A(R_i))$
- $R_i.A$  bilden disjunkte Menge  
 → auch  $S_i$  disjunkt



## Beispiel für abgeleitete Fragmentierung

- Horizontale Fragmentierung der Relation  $Pfs$ 
  - Entspricht der Relation  $R$
  - Disjunkte Mengen der Primärschlüssel  $PersNr$  von  $PhilPfs, TheoPfs, PhysPfs$

$$\pi_{PersNr}(PhilPfs)$$

PersNr
2125
2126
2133
2137

$$\pi_{PersNr}(TheoPfs)$$

PersNr
2134

$$\pi_{PersNr}(PhysPfs)$$

PersNr
2127
2136

$$Pfs$$

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2127	Kopernikus	W2	310	Physik
2133	Popper	W2	52	Philosophie
2134	Augustinus	W2	309	Theologie
2136	Curie	W3	36	Physik
2137	Kant	W3	7	Philosophie

$$PhilPfs$$

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

$$TheoPfs$$

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

$$PhysPfs$$

PersNr	Name	Rang	Raum	Fakultaet
2127	Kopernikus	W2	310	Physik
2136	Curie	W3	36	Physik

## Beispiel für abgeleitete Fragmentierung

- Horizontale Fragmentierung der Relation *Pfs*
- Ableitung der Fragmentierung der Relation *Vorlesungen*
  - *Vorlesungen* entspricht *S*, *Dozierende* Fremdschlüssel auf *PersNr*

$\pi_{PersNr}(PhilPfs)$	PersNr	Vorlesungen	VorNr	Titel	SWS	Dozierende
	2125		4052	Logik	4	2125
	2126		4630	Die 3 Kriterien	4	2137
	2133		5022	Glaube und Wissen	2	2134
	2137		5049	Mäeutik	2	2125
			5052	Wissenschaftstheorie	3	2126
$\pi_{PersNr}(TheoPfs)$	2134		5216	Bioethik	2	2126
			5259	Der Wiener Kreis	2	2133
$\pi_{PersNr}(PhysPfs)$	2127		6001	Grundlagen	3	2127
	2136		6003	Thermodynamik	4	2136
			6010	Statik	4	2127

## Beispiel für abgeleitete Fragmentierung

- Abgeleitete Fragmentierung:

PhilVorl	VorlNr	Titel	SWS	Dozierende
	4052	Logik	4	2125
	4630	Die 3 Kriterien	4	2137
	5049	Mäeutik	2	2125
	5052	Wissenschaftstheorie	3	2126
	5216	Bioethik	2	2126
	5259	Der Wiener Kreis	2	2133

TheoVorl	VorlNr	Titel	SWS	Dozierende
	5022	Glaube und Wissen	2	2134

PhysVorl	VorlNr	Titel	SWS	Dozierende
	6001	Grundlagen	3	2127
	6003	Thermodynamik	4	2136
	6010	Statik	4	2127

### Vorlesungen

VorlNr	Titel	SWS	Dozierende
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5022	Glaube und Wissen	2	2134
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
6001	Grundlagen	3	2127
6003	Thermodynamik	4	2136
6010	Statik	4	2127

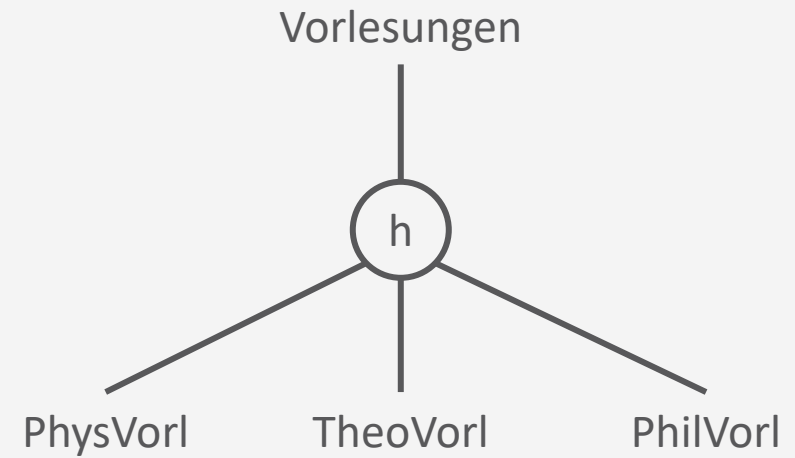
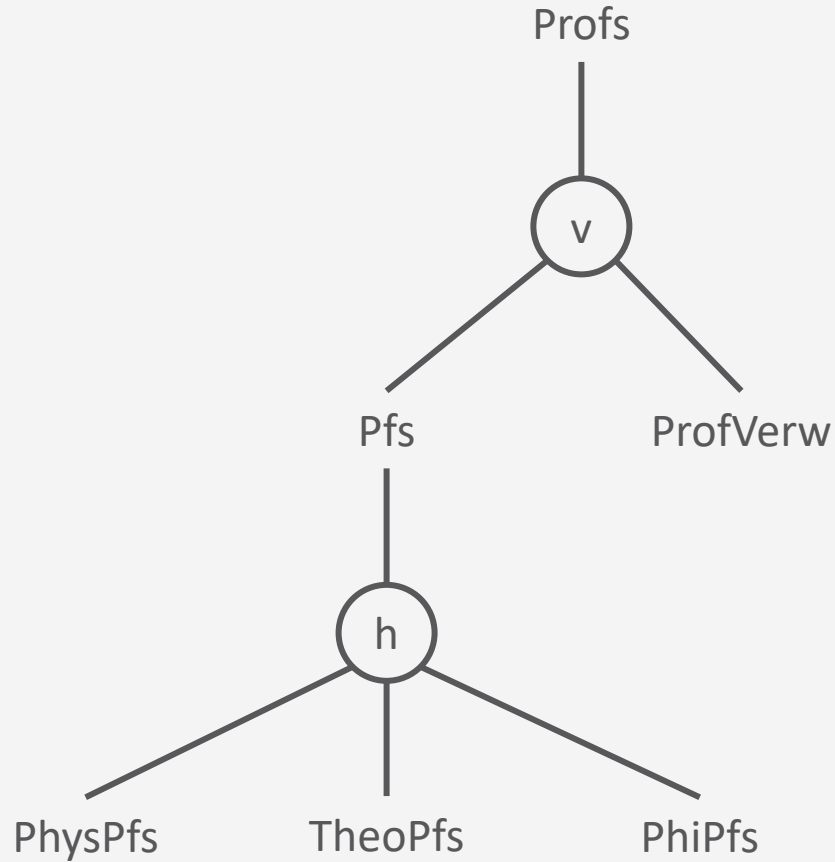
PhilPfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2133	Popper	W2	52	Philosophie
	2137	Kant	W3	7	Philosophie

TheoPfs	PersNr	Name	Rang	Raum	Fakultaet
	2134	Augustinus	W2	309	Theologie

PhysPfs	PersNr	Name	Rang	Raum	Fakultaet
	2127	Kopernikus	W2	310	Physik
	2136	Curie	W3	36	Physik

# Beispiele Fragmentierungen: Baumdarstellung

$$\begin{aligned}
 Profs &= ProfVerw \bowtie (PhilPfs \cup PhysPfs \cup TheoPfs) \\
 Vorlesungen &= PhilVorl \cup PhysVorl \cup TheoVorl
 \end{aligned}$$



## Fragmentierung: Korrektheitsanforderungen

- **Korrektheit** der Fragmentierung
  - *Rekonstruierbarkeit*
    - Original-Relation lässt sich aus den Fragmenten wiederherstellen
  - *Vollständigkeit*
    - Jedes Tupel ist einem Fragment zugeordnet
  - *Disjunktheit*
    - Die Fragmente überlappen sich nicht
- Bisher betrachtete Fragmentierungen
  - Horizontal
    - Selektion darf keine Tupel auslassen
    - Dann rekonstruierbar, vollständig, disjunkt
  - Vertikal
    - Projektion darf keine Attribute auslassen
    - Zur Rekonstruktion der Originalrelation, Übernahme der Primärschlüssel in die Fragmente notwendig
    - Dann rekonstruierbar, vollständig, aber nicht mehr disjunkt (Primärschlüssel vielfach)
  - Abgeleitet und gemischt
    - Selbe Überlegungen wie oben

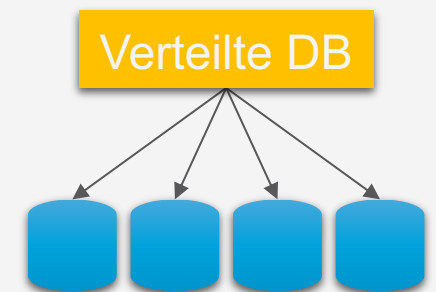


Was heißt das für die betrachteten Fragmentierungen?



# Fragmentierung

- Motivation
  - Lastverteilung
  - Nutzung von Parallelität zur Verkürzung von Antwortzeiten
- Art der Fragmentierung abhängig von häufigen Anfragen und Zugriffsmustern
- **Herausforderungen**
  - DBS muss Verteilung der Daten kennen und bei Anfragen berücksichtigen
    - An welche DB muss welche Teilanfrage?
    - Wie müssen Ergebnismengen zusammengefügt werden?
    - Was passiert, wenn eine DB ausfällt?



# Replikation

- (Logisches) Duplizieren von Daten nach strategischen Gesichtspunkten
- Motivation: Erhöhung der
  - Effizienz (schnellerer Zugriff „vor Ort statt entfernt“)
  - Ausfallsicherheit und Verfügbarkeit (Replikat als „Sicherungskopie“)
  - Autonomie (Unabhängigkeit von ansonsten nur einmal verfügbaren Daten)
- Zielkonflikte bei der Replikation
  - Deutliche Erhöhung der Zugriffseffizienz, Verfügbarkeit und Autonomie  
→ Große Anzahl von Replikaten auf möglichst vielen Knoten
  - Erhaltung der Datenkonsistenz  
→ Alle Replikate möglichst synchron aktualisieren
  - Erhöhung der Effizienz bei Änderungsoperationen  
→ Wenige Replikate wünschenswert; schneller synchron aktualisierbar

# Allokation

- (Physische) gezielte Zuordnung von Daten auf Rechner
- Die (physisch orientierte) Allokation legt fest,
  - Auf welchem Rechner ein Fragment gehalten wird
  - Welche Fragmente auf welchen Rechnern repliziert gespeichert werden
- Physische Umsetzung der Überlegung aus Fragmentierung und Replikation
- Bei der Allokation zu berücksichtigende Aspekte:
  - Effizienz
    - Minimierung von Zugriffskosten für Remote-Zugriffe
    - Vermeidung von Flaschenhälsen (Übertragungskapazität im Netz, Leistung einzelner Rechenknoten, ...)
  - Datensicherheit
    - Auswahl von Knoten hinsichtlich Verlässlichkeit
    - Redundante Speicherung von Daten

## Fortsetzung Beispiel

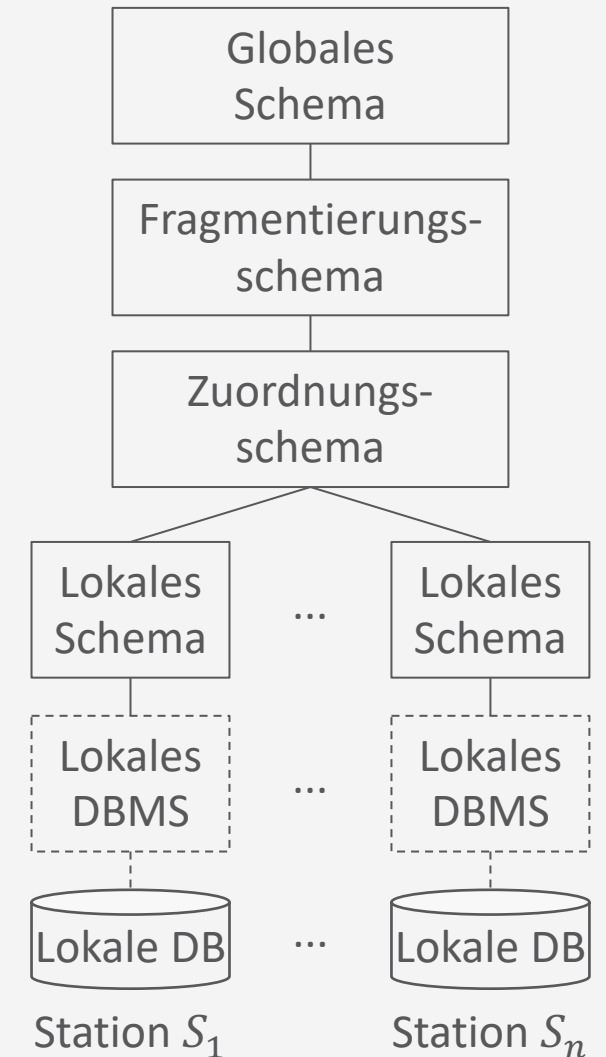
- Allokation ohne Replikation

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St <sub>Verw</sub>	Verwaltungsrechner	{ProfVerw}
St <sub>Phil</sub>	Dekanat Philosophie	{PhilVorl, PhilPfs}
St <sub>Phys</sub>	Dekanat Physik	{PhysVorl, PhysPfs}
St <sub>Theo</sub>	Dekanat Theologie	{TheoVorl, TheoPfs}

- Allokation mit Replikation, Beispiele:
  - Erhöhung der Ausfallsicherheit: Zweite St<sub>Verw</sub>-Station
  - Prüfungsamt benötigt Vorlesungsinformation: Weitere Station St<sub>Vorl</sub> mit den wiedervereinigten Vorlesungsfragmenten

# Transparenz in verteilten Datenbanken

- Grad der Unabhängigkeit, den ein VDBMS dem Benutzer beim Zugriff auf verteilte Daten vermittelt
- Verschiedene Stufen:
  - **Fragmentierungstransparenz:** Nutzer stellen Anfragen an das globale Schema, haben kein Wissen über Fragmentierung / Allokation
    - VDBMS muss Anfragen / Änderungen korrekt auf Fragmenten und in Stationen umsetzen
  - **Allokationstransparenz:** Nutzer müssen die Fragmentierung kennen, aber nicht die Allokation
    - Anfragen an Fragmente/lokale Schemata
  - **Lokale Schema-Transparenz:** Nutzer müssen Station kennen, auf dem Fragment liegt
    - Transparenz bezieht sich darauf, dass zumindest alle Stationen dasselbe Datenmodell und dieselbe Anfragesprache verwenden



# Fragmentierungstransparenz: Beispiel

- Anfrage, die Fragmentierungstransparenz voraussetzt:
  - **select** Titel, Name  
**from** Vorlesungen, Profs  
**where** Dozierende = PersNr;
  - Jeweils ein Join von \*Vorl mit \*Pfs auf der jeweiligen lokalen Station
  - Vereinigung der Join-Ergebnisse auf der Station, auf der die Anfrage liegt

Verteilte DBs

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St <sub>Verw</sub>	Verwaltungsrechner	{ProfVerw}
St <sub>Phil</sub>	Dekanat Physik	{PhilVorl, PhilPfs}
St <sub>Phys</sub>	Dekanat Philosophie	{PhysVorl, PhysPfs}
St <sub>Theo</sub>	Dekanat Theologie	{TheoVorl, TheoPfs}

PhilPfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2133	Popper	W2	52	Philosophie
	2137	Kant	W3	7	Philosophie

PhilVorl	VorlNr	Titel	SWS	Doziernde
	4052	Logik	4	2125
	4630	Die 3 Kriterien	4	2137
	5049	Mäeutik	2	2125
	5052	Wissenschaftstheorie	3	2126
	5216	Bioethik	2	2126
	5259	Der Wiener Kreis	2	2133

# Fragmentierungstranparenz: Beispiel

- Änderungsoperation, die Fragmentierungstransparenz voraussetzt:
  - **update** Professoren  
**set** Fakultät='Theologie'  
**where** Name='Sokrates';
  - Transferieren des Tupels aus Fragment PhilPfs in das Fragment TheoPfs
    - Löschen aus PhilPfs, Einfügen in TheoPfs
  - Ändern der abgeleiteten Fragmentierung von Vorlesungen
    - Einfügen der von Sokrates gehaltenen Vorlesungen in TheoVorl, Löschen aus PhilVorl

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St <sub>Verw</sub>	Verwaltungsrechner	{ProfVerw}
St <sub>Phil</sub>	Dekanat Physik	{PhilVorl, PhilPfs}
St <sub>Phys</sub>	Dekanat Philosophie	{PhysVorl, PhysPfs}
St <sub>Theo</sub>	Dekanat Theologie	{TheoVorl, TheoPfs}

PhilPfs	PersNr	Name	Rang	Raum	Fakultaet
	2125	Sokrates	W3	226	Philosophie
	2126	Russel	W3	232	Philosophie
	2133	Popper	W2	52	Philosophie
	2137	Kant	W3	7	Philosophie

PhilVorl	VorlNr	Titel	SWS	Doziernde
	4052	Logik	4	2125
	4630	Die 3 Kriterien	4	2137
	5049	Mäeutik	2	2125
	5052	Wissenschaftstheorie	3	2126
	5216	Bioethik	2	2126
	5259	Der Wiener Kreis	2	2133

# Allokationstransparenz: Beispiel

- Nutzer\*innen müssen Fragmentierung kennen, aber nicht Aufenthaltsort von Fragmenten

- Beispielanfrage:

```

select Gehalt
from ProfVerw
where Name='Sokrates';

```

- Unter Umständen muss Originalrelation rekonstruiert werden

- Beispiel: Verwaltung möchte wissen, wieviel die W3-Professoren der Theologie insgesamt verdienen

```

select sum(Gehalt)
from ProfVerw, TheoPfs
where ProfVerw.PersNr=TheoPfs.PersNr and Rang='W3';

```

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St <sub>Verw</sub>	Verwaltungsrechner	{ProfVerw}
St <sub>Phil</sub>	Dekanat Physik	{PhilVorl, PhilPfs}
St <sub>Phys</sub>	Dekanat Philosophie	{PhysVorl, PhysPfs}
St <sub>Theo</sub>	Dekanat Theologie	{TheoVorl, TheoPfs}

ProfVerw	PersNr	Name	Gehalt	St.Klasse
	2125	Sokrates	85000	1
	2126	Russel	80000	3
	2127	Kopernikus	65000	5
	2133	Popper	68000	1
	2134	Augustinus	55000	5
	2136	Curie	95000	3
	2137	Kant	98000	1

TheoPfs	PersNr	Name	Rang	Raum	Fakultaet
	2134	Augustinus	W2	309	Theologie



# Lokale Schema-Transparenz

- Der Benutzer muss auch noch den Rechner kennen, auf dem ein Fragment liegt
  - Beispielanfrage:
    - `select Name`  
`from TheoPfs at StTheo`  
`where Rang='W2';`
  - Ist überhaupt Transparenz gegeben?
    - Lokale Schema-Transparenz setzt voraus, dass alle Rechner dasselbe Datenmodell und dieselbe Anfragesprache verwenden.
      - Anfrage kann somit analog auch an Station St<sub>Phil</sub> ausgeführt werden
      - Nicht möglich bei Kopplung unterschiedlicher DBMS

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St <sub>Verw</sub>	Verwaltungsrechner	{ProfVerw}
St <sub>Phil</sub>	Dekanat Physik	{PhilVorl, PhilPfs}
St <sub>Phys</sub>	Dekanat Philosophie	{PhysVorl, PhysPfs}
St <sub>Theo</sub>	Dekanat Theologie	{TheoVorl, TheoPfs}

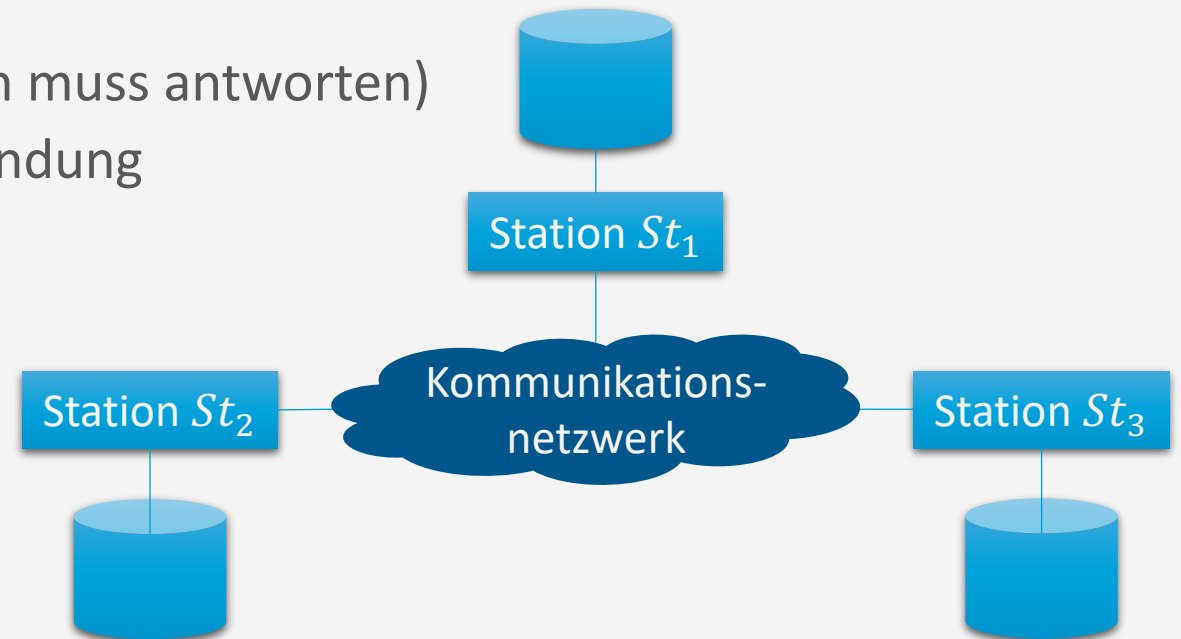
ProfVerw	PersNr	Name	Gehalt	St.Klasse
	2125	Sokrates	85000	1
	2126	Russel	80000	3
	2127	Kopernikus	65000	5
	2133	Popper	68000	1
	2134	Augustinus	55000	5
	2136	Curie	95000	3
	2137	Kant	98000	1

TheoPfs	PersNr	Name	Rang	Raum	Fakultaet
	2134	Augustinus	W2	309	Theologie

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

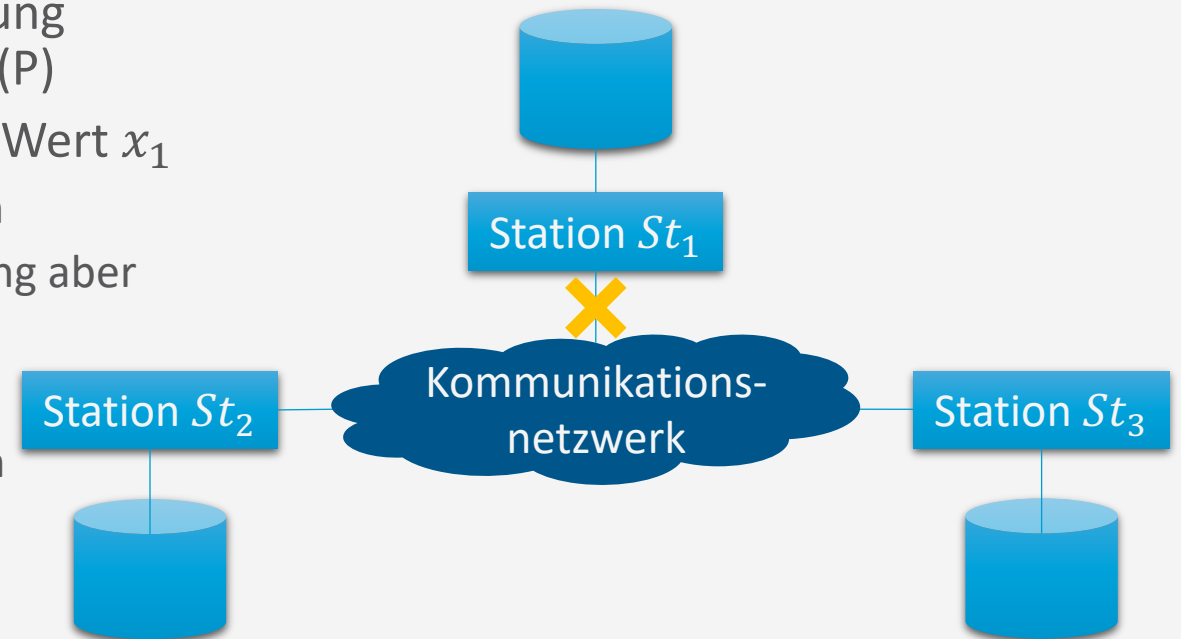
## Konsistenzmodell: CAP

- Neben Transparenz, weitere Anforderungen an verteilte DBs:  
**Konsistenz, Verfügbarkeit, Ausfalltoleranz**
  - Consistency, Availability, Partition tolerance = CAP
  - *Konsistenz*: Alle Knoten liefern identische Daten
  - *Verfügbarkeit*: Akzeptable Reaktionszeit (Station muss antworten)
  - *Ausfalltoleranz*: Knoten / Kommunikationsverbindung kann ausfallen
- **CAP-Theorem**:  
Nur zwei der drei Anforderungen in verteilten Systemen erfüllbar
  - Vermutung von Brewer (2000)
  - Beweis von Gilbert und Lynch (2002)



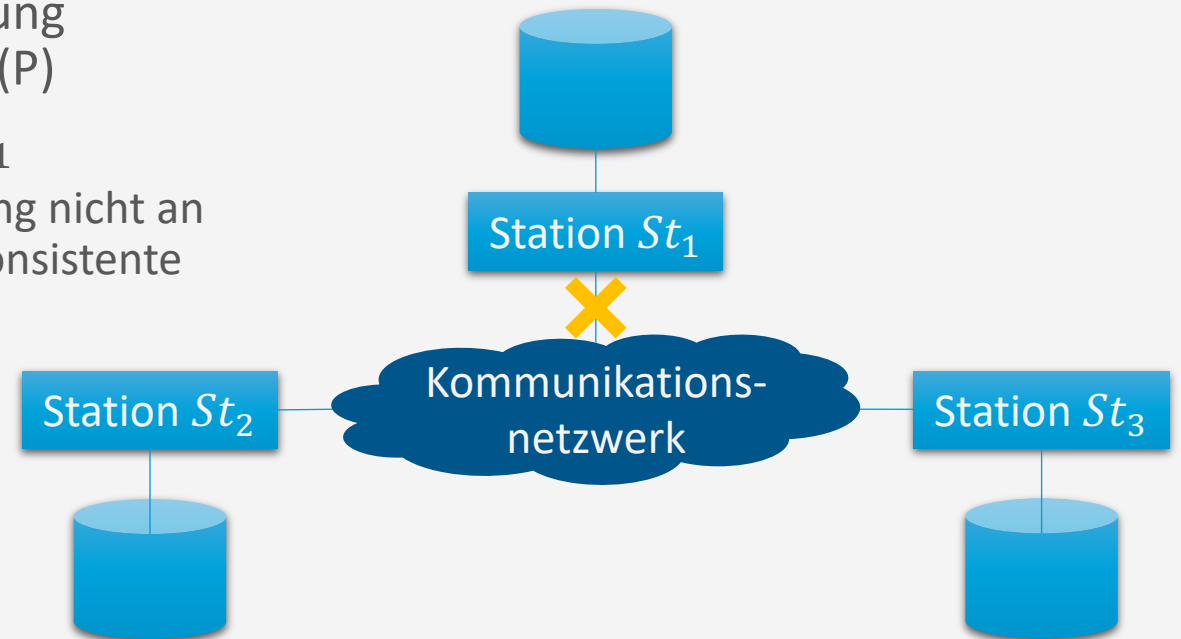
# CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
    1. AP geht nicht mit C
      - Annahme: System ist partitioniert (keine Verbindung zwischen  $St_1$  und  $St_2$ ), was nach zu tolerieren ist (P)
      - Anfrage an Station  $St_1$  Datum  $X$  zu schreiben mit Wert  $x_1$ 
        - Da das System verfügbar ist (A), muss  $St_1$  antworten
        - Da das System partitioniert ist, kann  $St_1$  die Änderung aber nicht an die anderen Stationen weitergeben
      - Dann Anfrage an Station  $St_2$  Datum  $X$  zu lesen
        - Da das System verfügbar ist (A), muss  $St_2$  antworten
        - Da das System partitioniert ist, kann  $St_2$  nur den veralteten Wert  $x_0$  zurückgeben  
 → Inkonsistenz (kein C)



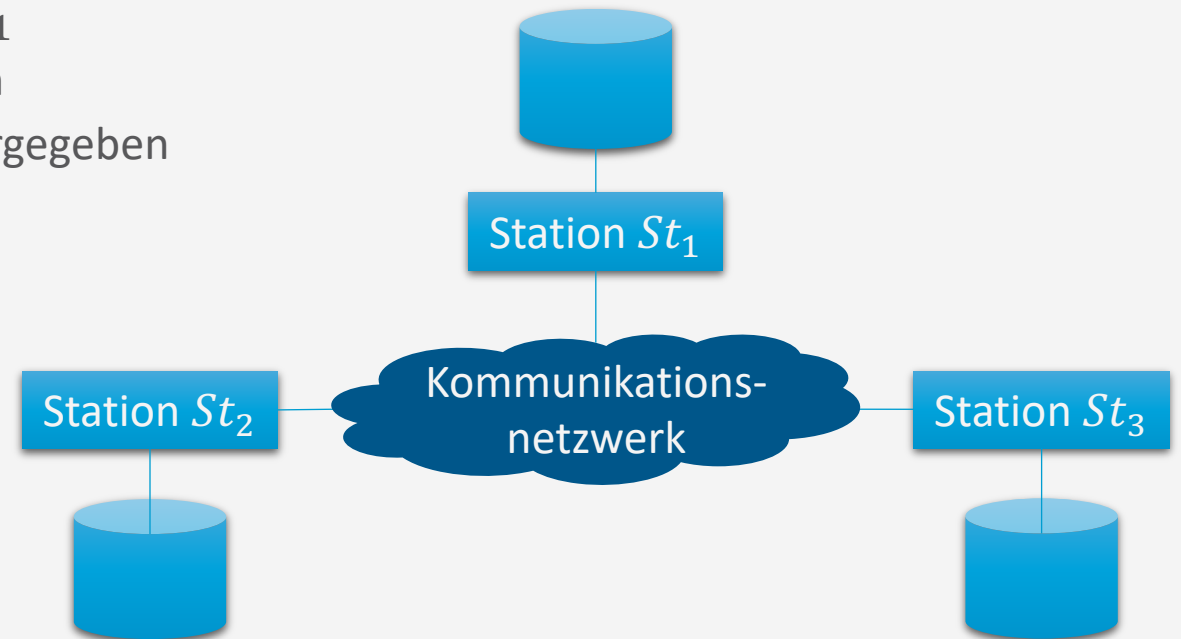
## CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
- 2. CP geht nicht mit A
  - Annahme: System ist partitioniert (keine Verbindung zwischen  $St_1$  und  $St_2$ ), was nach zu tolerieren ist (P)
  - Anfrage an Station  $St_1$   $X$  zu schreiben mit Wert  $x_1$ 
    - Da das System partitioniert ist, kann  $St_1$  die Änderung nicht an die anderen Stationen weitergeben, da es sonst inkonsistente Daten hätte (C erhalten)
    - $St_1$  muss die Anfrage ablehnen bzw. kann nicht antworten → Keine Verfügbarkeit (kein A)



## CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
- 3. CA geht nicht mit P
  - Anfrage an Station  $St_1$   $X$  zu schreiben mit Wert  $x_1$ 
    - Da das System verfügbar ist (A), muss  $St_1$  antworten
    - Die Änderung muss an die anderen Stationen weitergegeben werden (C), darf also nicht partitioniert sein  
→ Ausfalltoleranz nicht gegeben (kein P)



## CAP-Theorem

- Welche der Eigenschaften nicht berücksichtigt wird, gilt nicht unbedingt systemweit
  - CAP Eigenschaften in der Regel nicht binär, sondern graduell
- CA-Systeme (Konsistenz und Verfügbarkeit)
  - Beispiel: relationale Datenbank mit verteilten Transaktionen
  - Voraussetzung: Verbindung zwischen den Knoten besteht; sonst Schreiben ggf. blockieren
- CP-Systeme (Konsistenz und Ausfalltoleranz)
  - Beispiel: Bank-Anwendungen
  - Hohe Verfügbarkeit bei lesenden Operationen, bei Unterbrechung der Netzwerkverbindung geringere Verfügbarkeit
- AP-Systeme (Verfügbarkeit und Ausfalltoleranz)
  - Beispiel: Domain Name System (DNS)
  - Hohe Verfügbarkeit bei schreibenden und lesenden Operationen, eingeschränkte Konsistenz

## Konsistenzmodell: **CAP**

Not-only SQL (NoSQL)-Datenbanken verzichten zeit- bzw. teilweise auf Konsistenz und betonen Verfügbarkeit und Ausfalltoleranz

- Relaxiertes Konsistenzmodell:
  - Basically Available, Soft State, Eventually Consistent (**BASE**)
    - Ordnet Konsistenz der Verfügbarkeit unter
      - Konsistenz kein fester Zustand nach einer Transaktion, wird später erreicht
    - Replizierte Daten haben nicht alle den neuesten Zustand
      - Vermeidung des (teuren) Zwei-Phasen-Commit-Protokolls (später)
      - Transaktionen könnten veraltete Daten zu lesen bekommen
    - Eventual Consistency
      - Würde man das System anhalten, würden alle Kopien irgendwann in denselben Zustand übergehen
  - Read your Writes-Garantie
    - $T_i$  leistet auf jeden Fall ihre eigenen Änderungen
  - Monotonic Read-Garantie
    - $T_i$  würde beim wiederholten Lesen keinen älteren Zustand als den vorher mal sichtbaren lesen

## Zwischenzusammenfassung

- Fragmentierung: Verteilung der Daten in Fragmente
  - Horizontal, vertikal, gemischt (hor. + vert.), abgeleitet (i.d.R. über Fremdschlüssel)
  - Korrektheit: Rekonstruierbarkeit, Vollständigkeit, Disjunktheit
- Replikation: Erhöhung der Verfügbarkeit / Effizienz
  - Nachteil: Daten müssen konsistenz gehalten werden
- Allokation: Physische Zuordnung auf Stationen (lokale DBs)
- Transparenz
  - Fragmentierungstransparenz, Allokationstransparenz, lokale Schema-Transparenz
  - Von nur globales Schema genutzt bis hin zu lokale Stationen müssen bekannt sein
- CAP: Konsistenz, Verfügbarkeit, Ausfalltoleranz
  - CAP-Theorem: Nur zwei von drei Eigenschaften zur Zeit erreichbar



# Überblick: 8. Verteilte Datenbanken

## A. *Verteilte DBMS*

- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

## B. *Anfragenbeantwortung in verteilten Systemen*

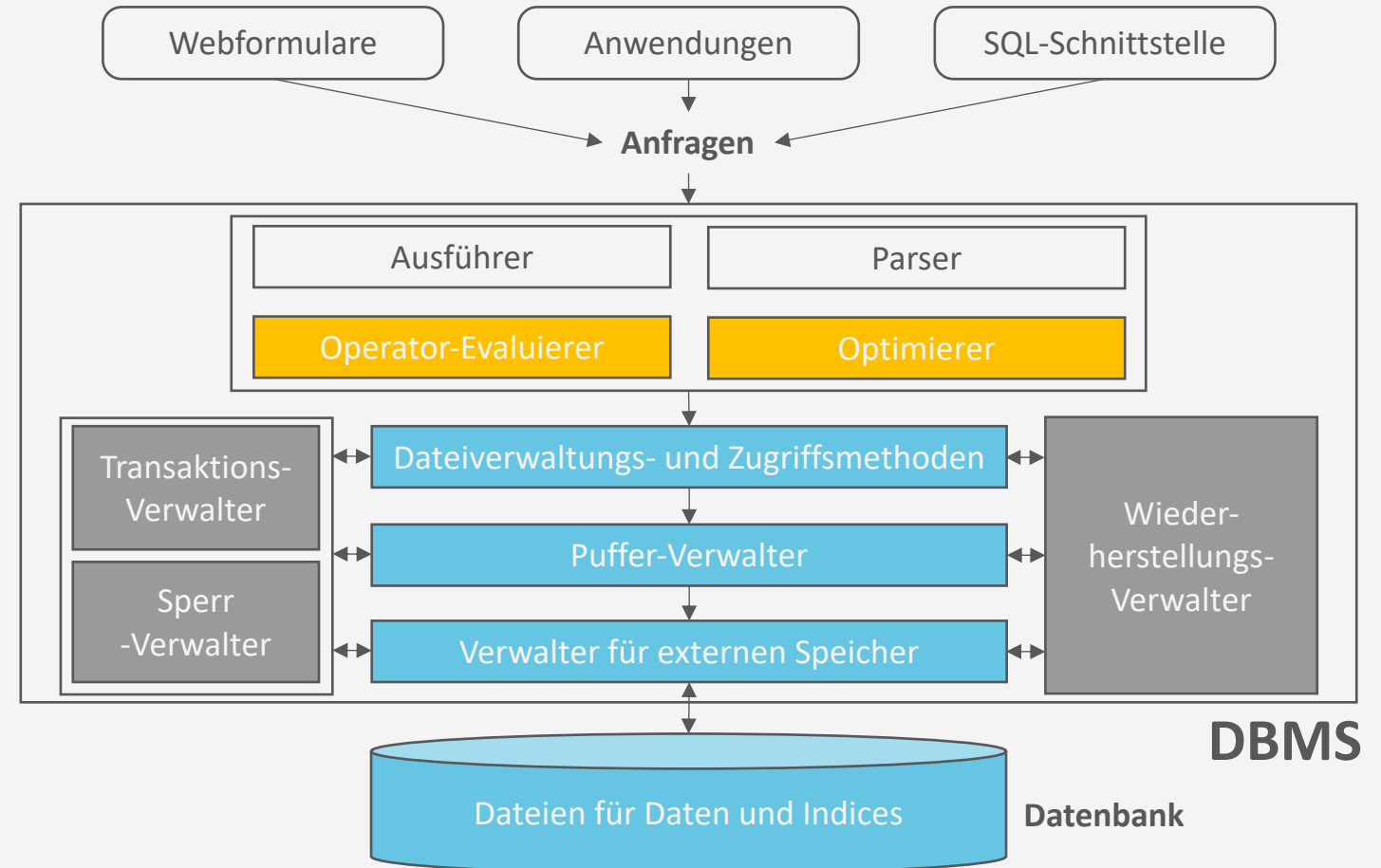
- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

## C. *Föderierte DBS*

- Integration
- Migration

# Anfrageübersetzung und Anfrageoptimierung

- Voraussetzung:  
Fragmentierungstransparenz
- Aufgabe des Operator-Evaluierers:  
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

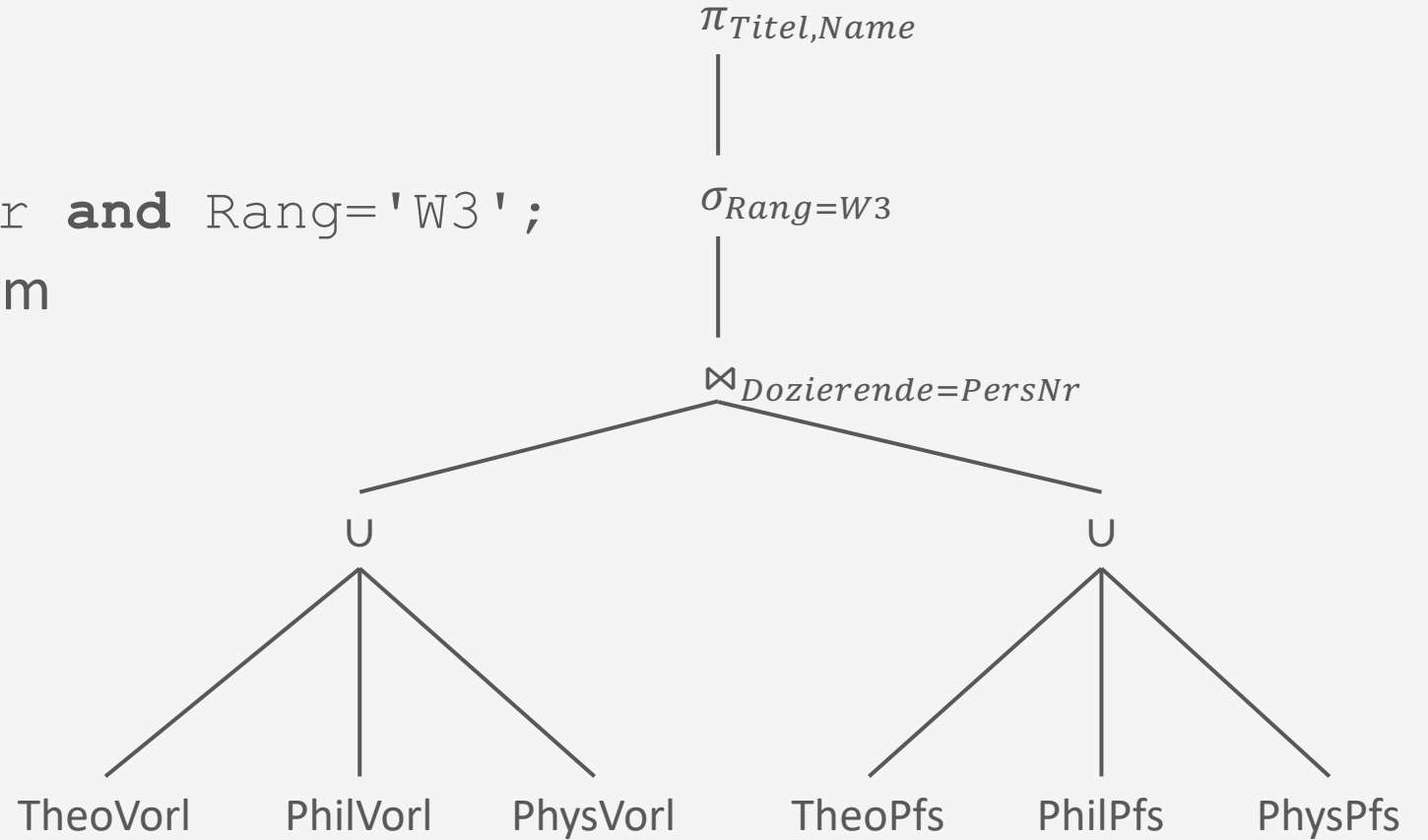


## Anfragebearbeitung bei horizontaler Fragmentierung

- Übersetzung einer SQL-Anfrage auf dem globalen Schema in eine äquivalente Anfrage auf den Fragmenten benötigt zwei 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 = **kanonische Form** der Anfrage
  2. Kombination des Rekonstruktionsausdrucks mit dem algebraischen Anfrageausdruck, der sich aus der Übersetzung der SQL-Anfrage ergibt

# Algebraischer Ausdruck: Beispiel

- Anfrage
  - **select** Titel, Name
  - **from** Vorlesungen, Profs
  - **where** Dozierende = PersNr **and** Rang='W3';
- Algebraischer Ausdruck in Baumform
  - I.e., kanonische Form der Anfrage



# Algebraische Äquivalenzen

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

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

- Verallgemeinerung auf  $n$  horizontale Fragmente  $R_1, \dots, R_n$  von  $R$  und  $m$  Fragmente  $S_1, \dots, S_m$  von  $S$  ergibt:

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

- Falls gilt  $S_i = S \bowtie_p R_i$  mit  $S = S_1 \cup \dots \cup S_m$ , dann gilt immer:  $R \bowtie_p S = \bigcup_{i=1}^m (R_i \bowtie_p S_i)$

- Semi-Join**  $\bowtie_p$ :  $T \bowtie_p U$  wählt diejenigen Tupel aus  $T$ , die ein Join-Tupel in  $U$  haben

- Gegeben Attribute  $A_1, \dots, A_k$  in  $T$ ,  $\bowtie_p$  ausdrückbar als:  $T \bowtie_p U = \pi_{A_1, \dots, A_k}(T \bowtie_p U)$

- Bei derart abgeleiteten horizontalen Fragmentierungen gilt somit immer:

$$(R_1 \cup \dots \cup R_n) \bowtie_p (S_1 \cup \dots \cup S_m) = (R_1 \bowtie_p S_1) \cup \dots \cup (R_m \bowtie_p S_m)$$

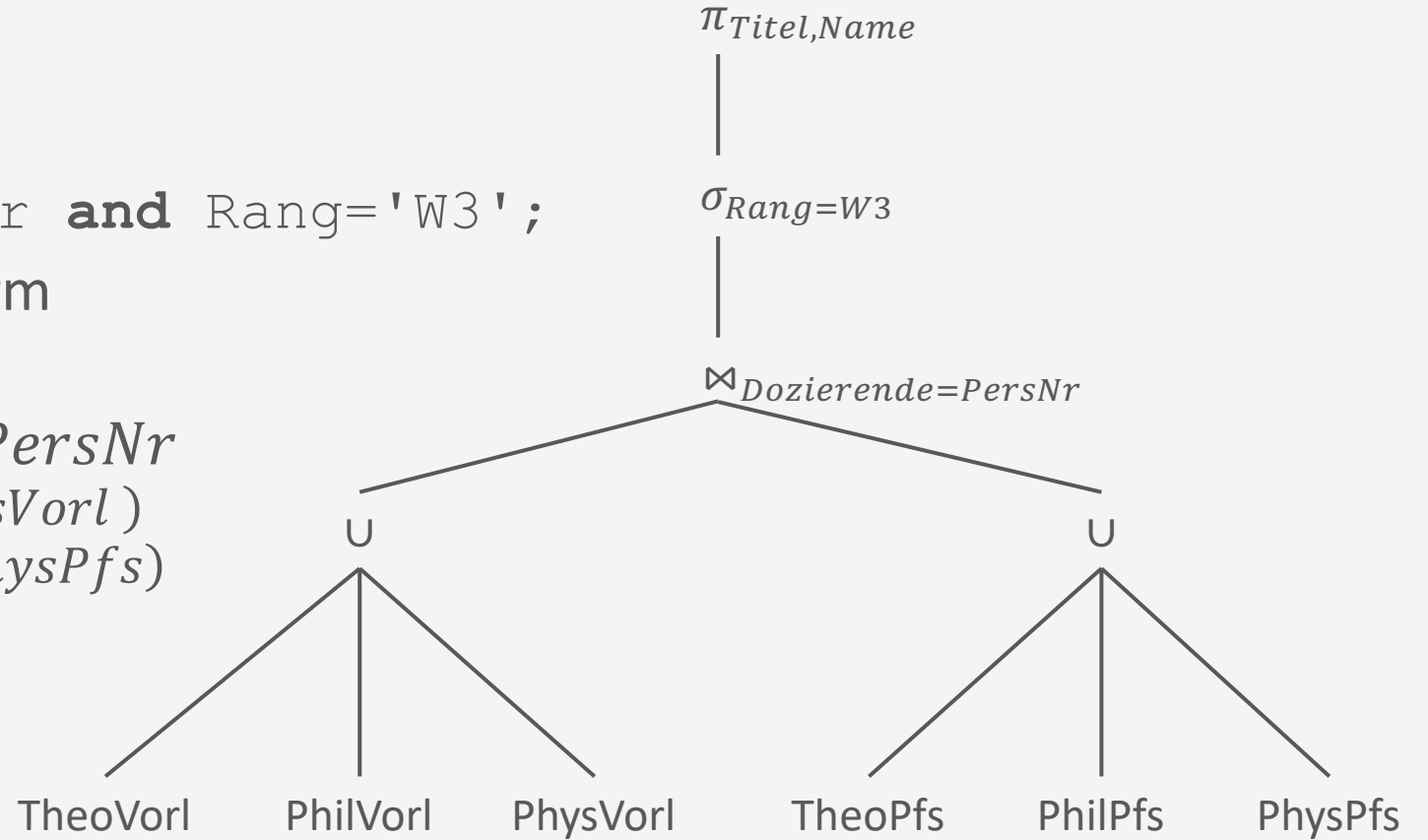
# Algebraischer Ausdruck: Beispiel

- Anfrage
  - **select** Titel, Name
  - **from** Vorlesungen, Profs
  - **where** Dozierende = PersNr **and** Rang='W3';

- Algebraischer Ausdruck in Baumform

- I.e., kanonische Form der Anfrage

- Umformung:  $p \triangleq Dozierende = PersNr$   
 $(TheoVorl \cup PhilVorl \cup PhysVorl)$   
 $\bowtie_p (TheoPfs \cup PhilPfs \cup PhysPfs)$   
 $= (TheoVorl \bowtie_p TheoPfs)$   
 $\cup (PhilVorl \bowtie_p PhilPfs)$   
 $\cup (PhysVorl \bowtie_p PhysPfs)$



# Optimale Form der Anfrage

- Selektionen und Projektionen über den Vereinigungsoperator hinweg „nach unten drücken“:

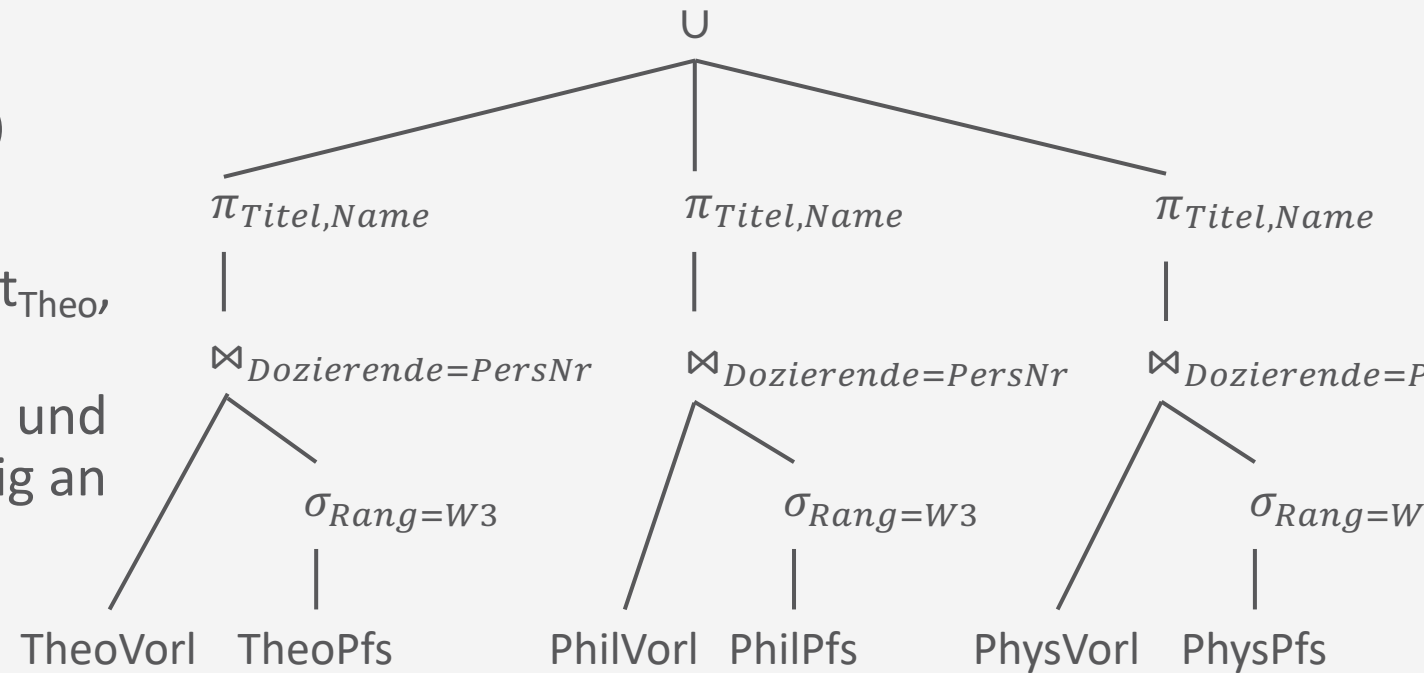
$$\sigma_p(R_1 \cup R_2) = \sigma_p(R_1) \cup \sigma_p(R_2)$$

$$\pi_{List}(R_1 \cup R_2) = \pi_{List}(R_1) \cup \pi_{List}(R_2)$$

- Ergibt folgenden Auswertungsplan:

- Auswertungen lokal auf den Stationen  $St_{Theo}$ ,  $St_{Phys}$  und  $St_{Phil}$  ausführen  
 → Stationen können parallel abarbeiten und lokales Ergebnis voneinander unabhängig an die Station, die die abschließende Vereinigung durchführt, übermitteln

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
$St_{Verw}$	Verwaltungsrechner	{ProfVerw}
$St_{Phil}$	Dekanat Physik	{PhilVorl, PhilPfs}
$St_{Phys}$	Dekanat Philosophie	{PhysVorl, PhysPfs}
$St_{Theo}$	Dekanat Theologie	{TheoVorl, TheoPfs}

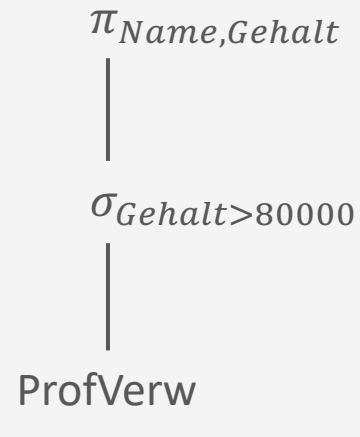


# Anfragebearbeitung bei vertikaler Fragmentierung

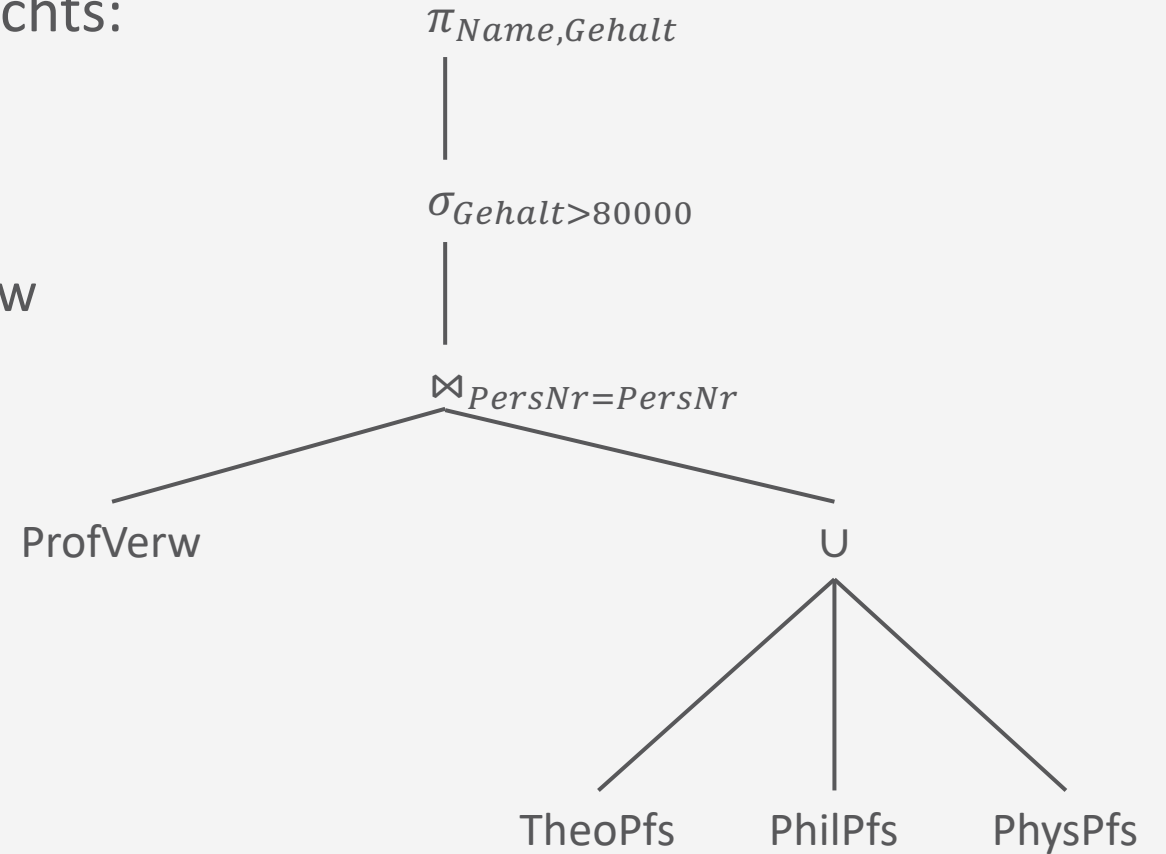
- Beispiel mit kanonischem Auswertungsplan rechts:

- **select** Name, Gehalt  
**from** Profs  
**where** Gehalt > 80000;

- Alle notwendigen Informationen sind in ProfVerw enthalten → Teil mit Vereinigung und Join kann abgeschnitten werden



- Das ergibt einen optimierten Auswertungsplan
- Beispiel für schlecht zu optimierende Anfrage
  - Dazu noch Rang ausgeben wollen





# Join-Auswertung in VDBMS

- Join-Auswertung spielt kritischere Rolle als in zentralisierten Datenbanken
  - Problem: Argumente eines Joins zweier Relationen können auf unterschiedlichen Stationen liegen
  - Allgemeinster Fall:
    - 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
- Zwei Möglichkeiten: Join-Auswertung mit und ohne Filterung
  - Ohne Filterung: Nested-Loops, Transfer einer Argumentrelation, Transfer beider Argumentrelationen
  - Mit Filterung: Semi-Join zur Filterung

$$R \bowtie S$$

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	$S$	C	D	E
	$a_1$	$b_1$	$c_1$		$c_1$	$d_1$	$e_1$
	$a_2$	$b_2$	$c_2$		$c_3$	$d_2$	$e_2$
	$a_3$	$b_3$	$c_1$		$c_4$	$d_3$	$e_3$
	$a_4$	$b_4$	$c_2$		$c_5$	$d_4$	$e_4$
	$a_5$	$b_5$	$c_3$		$c_7$	$d_5$	$e_5$
	$a_6$	$b_6$	$c_2$		$c_8$	$d_6$	$e_6$
	$a_7$	$b_7$	$c_6$		$c_5$	$d_7$	$e_7$

# 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  $St_S$ )
  - $C$  die Join-Variable
- Diese Vorgehensweise benötigt pro Tupel aus  $R$  eine Anforderung und eine passende Tupelmenge aus  $S$  (welche bei vielen Anforderungen leer sein könnte)
  - Es werden  $2 \cdot |R|$  Nachrichten benötigt

$$R \bowtie S$$

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	$S$	C	D	E
	$a_1$	$b_1$	$c_1$		$c_1$	$d_1$	$e_1$
	$a_2$	$b_2$	$c_2$		$c_3$	$d_2$	$e_2$
	$a_3$	$b_3$	$c_1$		$c_4$	$d_3$	$e_3$
	$a_4$	$b_4$	$c_2$		$c_5$	$d_4$	$e_4$
	$a_5$	$b_5$	$c_3$		$c_7$	$d_5$	$e_5$
	$a_6$	$b_6$	$c_2$		$c_8$	$d_6$	$e_6$
	$a_7$	$b_7$	$c_6$		$c_5$	$d_7$	$e_7$

## 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

$$R \bowtie S$$

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

$R$	$A$	$B$	$C$	$S$	$C$	$D$	$E$
	$a_1$	$b_1$	$c_1$		$c_1$	$d_1$	$e_1$
	$a_2$	$b_2$	$c_2$		$c_3$	$d_2$	$e_2$
	$a_3$	$b_3$	$c_1$		$c_4$	$d_3$	$e_3$
	$a_4$	$b_4$	$c_2$		$c_5$	$d_4$	$e_4$
	$a_5$	$b_5$	$c_3$		$c_7$	$d_5$	$e_5$
	$a_6$	$b_6$	$c_2$		$c_8$	$d_6$	$e_6$
	$a_7$	$b_7$	$c_6$		$c_5$	$d_7$	$e_7$

## 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)

$$R \bowtie S$$

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	$S$	C	D	E
$a_1$	$b_1$	$c_1$	$c_1$	$d_1$	$e_1$		
$a_2$	$b_2$	$c_2$	$c_3$	$d_2$	$e_2$		
$a_3$	$b_3$	$c_1$	$c_4$	$d_3$	$e_3$		
$a_4$	$b_4$	$c_2$	$c_5$	$d_4$	$e_4$		
$a_5$	$b_5$	$c_3$	$c_7$	$d_5$	$e_5$		
$a_6$	$b_6$	$c_2$	$c_8$	$d_6$	$e_6$		
$a_7$	$b_7$	$c_6$	$c_5$	$d_7$	$e_7$		

# Join-Auswertung mit Filterung

- Idee: Nur Transfer von Tupeln mit passendem Join-Partner über Semi-Join

$$R \bowtie S$$

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

- Benutzung der folgenden algebraischen Eigenschaften:

- Bei zwei Relationen  $R, S$  mit Filterung von  $S$  über Join-Attribut  $C$

- $R \bowtie S = R \bowtie (R \bowtie S)$

- $R \bowtie S = \pi_C(R) \bowtie S$

- Vorgehen

1. Transfer der unterschiedlichen  $C$ -Werte von  $R (= \pi_{List}(R))$  nach  $St_S$

2. Auswertung des Semi-Joins  $R \bowtie S = \pi_C(R) \bowtie S$  auf  $St_S$  und Transfer nach  $St_R$

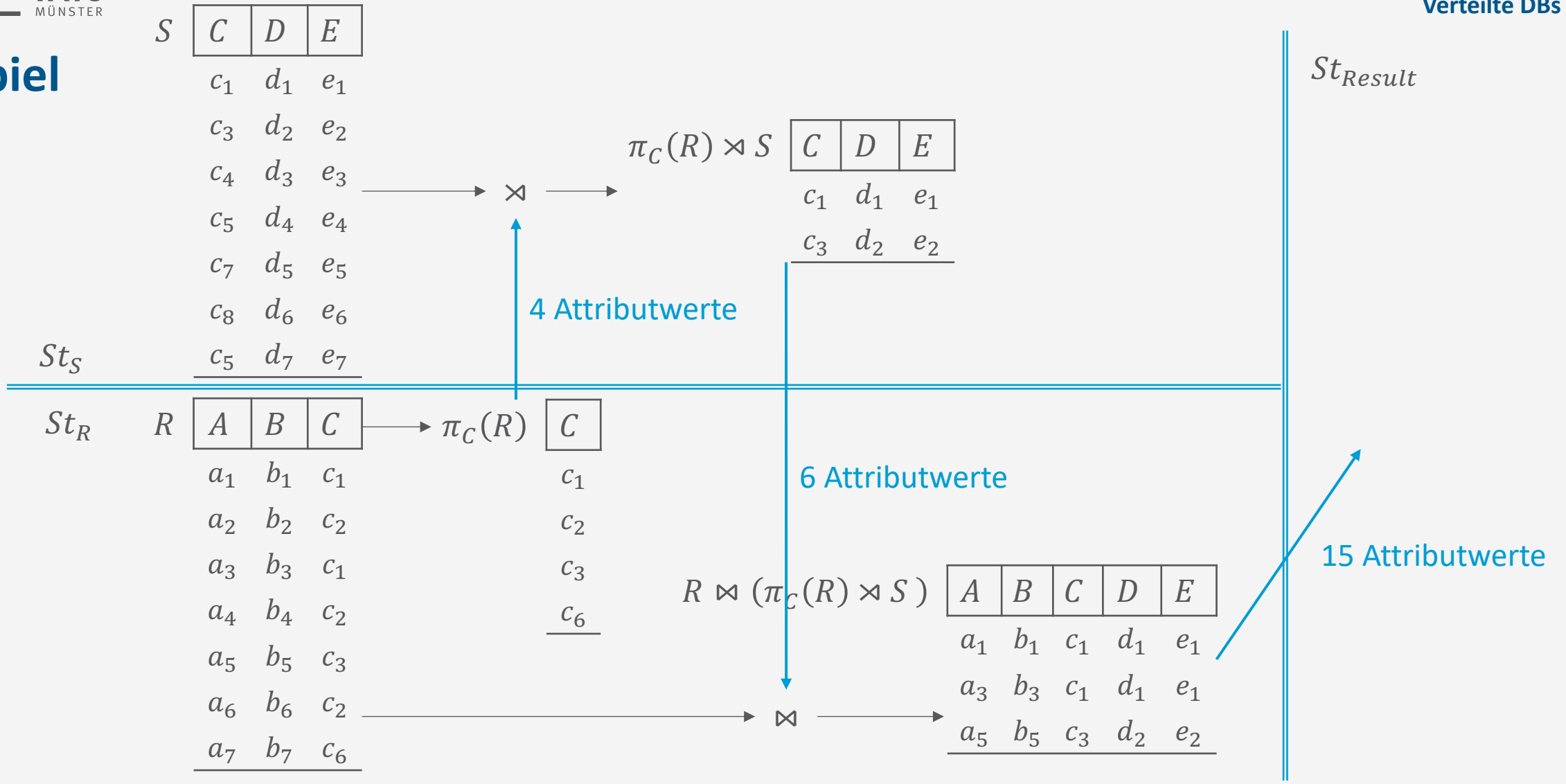
3. Auswertung des Joins auf  $St_R$ , der nur diese transferierten Ergebnistupel des Semi-Joins braucht

- Transferkosten werden nur reduziert, wenn gilt:  $\|\pi_C(R)\| + \|R \bowtie S\| < \|S\|$

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

$R$	A	B	C	$S$	C	D	E
	$a_1$	$b_1$	$c_1$		$c_1$	$d_1$	$e_1$
	$a_2$	$b_2$	$c_2$		$c_3$	$d_2$	$e_2$
	$a_3$	$b_3$	$c_1$		$c_4$	$d_3$	$e_3$
	$a_4$	$b_4$	$c_2$		$c_5$	$d_4$	$e_4$
	$a_5$	$b_5$	$c_3$		$c_7$	$d_5$	$e_5$
	$a_6$	$b_6$	$c_2$		$c_8$	$d_6$	$e_6$
	$a_7$	$b_7$	$c_6$		$c_5$	$d_7$	$e_7$

# Beispiel



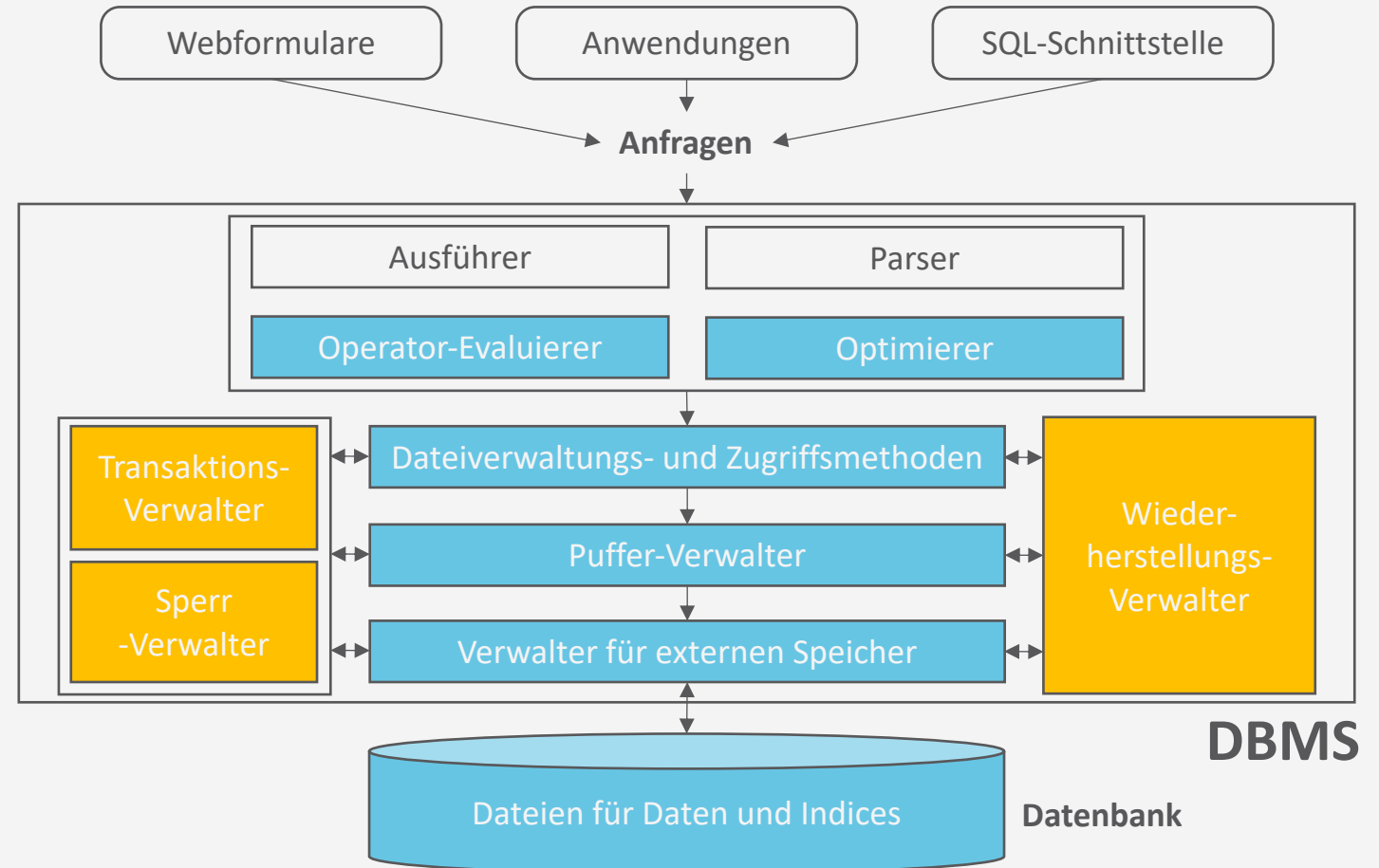
## Parameter für die Kosten eines Auswertungsplan

- Kardinalitäten von Argumentrelationen
  - Wie bisher
- Selektivitäten von Joins und Selektionen
  - Wie bisher
- *Transferkosten für Datenkommunikation (Verbindungsaufbau + von Datenvolumen abhängiger Anteil für Transfer)*
- *Auslastung der einzelnen VDBMS-Stationen*

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

# Transaktionsmanagement in VDBMS

- Transaktionen können sich bei VDBMS über mehrere Rechnerknoten erstrecken





# Widerherstellungsverwaltung in VDBMS

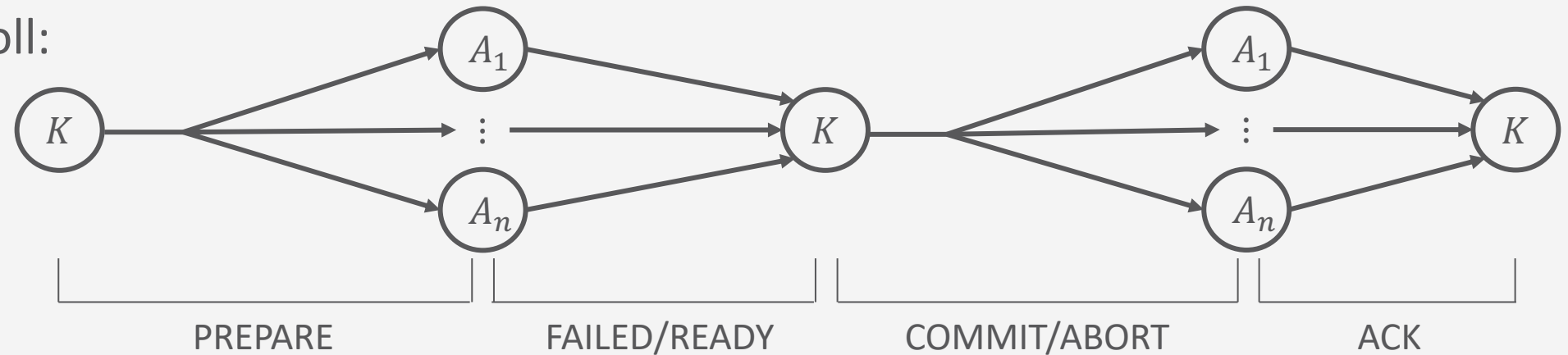
- Recovery:
  - **Redo**
    - Wenn eine Station nach einem Fehler wieder anläuft, müssen alle Änderungen einmal abgeschlossener Transaktionen - seien sie lokal auf dieser Station oder global über mehrere Stationen ausgeführt worden - auf den an dieser Station abgelegten Daten wiederhergestellt werden
  - **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
    - abort: Globale Transaktion wird gar nicht festgeschrieben
- Problem in verteilter Umgebung, da die Stationen eines VDBMS unabhängig voneinander abstürzen können

## Problemlösung: Zwei-Phasen-Commit-Protokoll

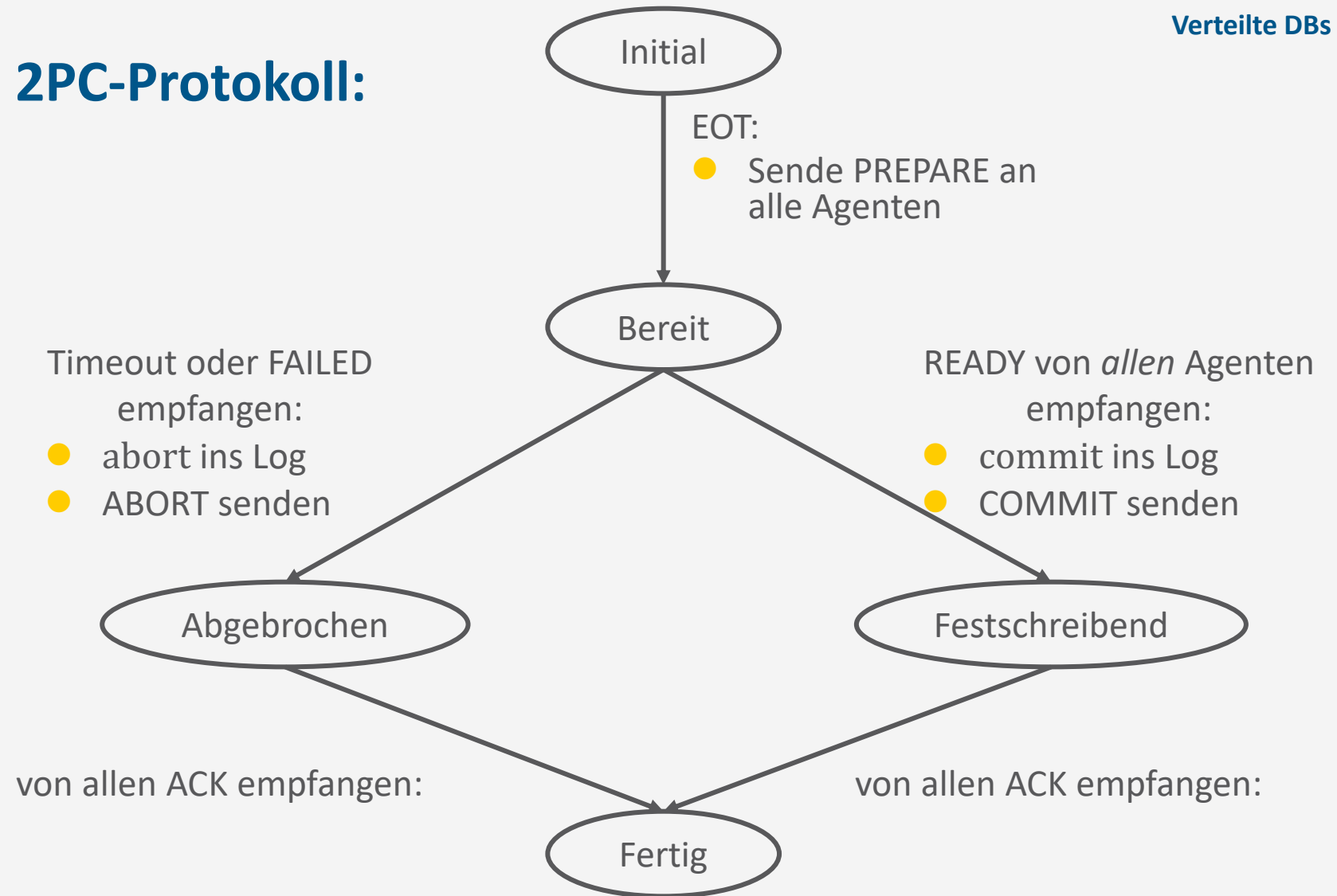
- Zwei-Phasen-Commit-Protokoll (2PC) gewährleistet Atomarität der EOT-Behandlung
- 2PC-Verfahren wird von sogenanntem Koordinator  $K$  überwacht
- Gewährleistet, dass die  $n$  Agenten (Stationen im VDBMS)  $A_1, \dots, A_n$ , die an einer Transaktion beteiligt waren, entweder alle von Transaktion  $T$  geänderten Daten festschreiben oder alle Änderungen von  $T$  rückgängig machen
- Nachrichtenaustausch beim 2PC-Protokoll:



## 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  $A_1, \dots, A_n$  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 oder 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

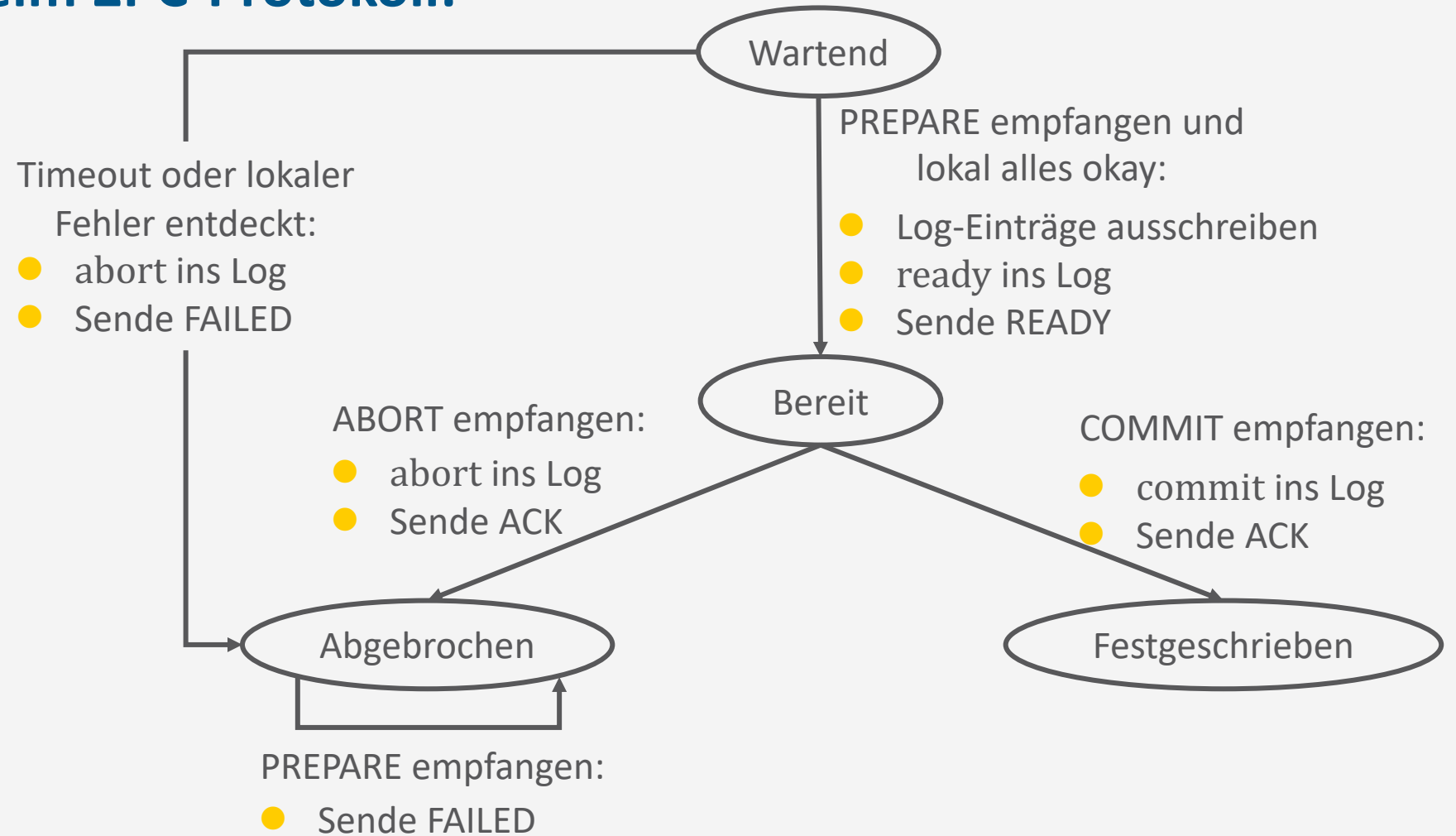
# Zustandsübergang beim 2PC-Protokoll: Koordinator



„Bullet“ ●  
= wichtigste Aktion(en)

# Zustandsübergang beim 2PC-Protokoll:

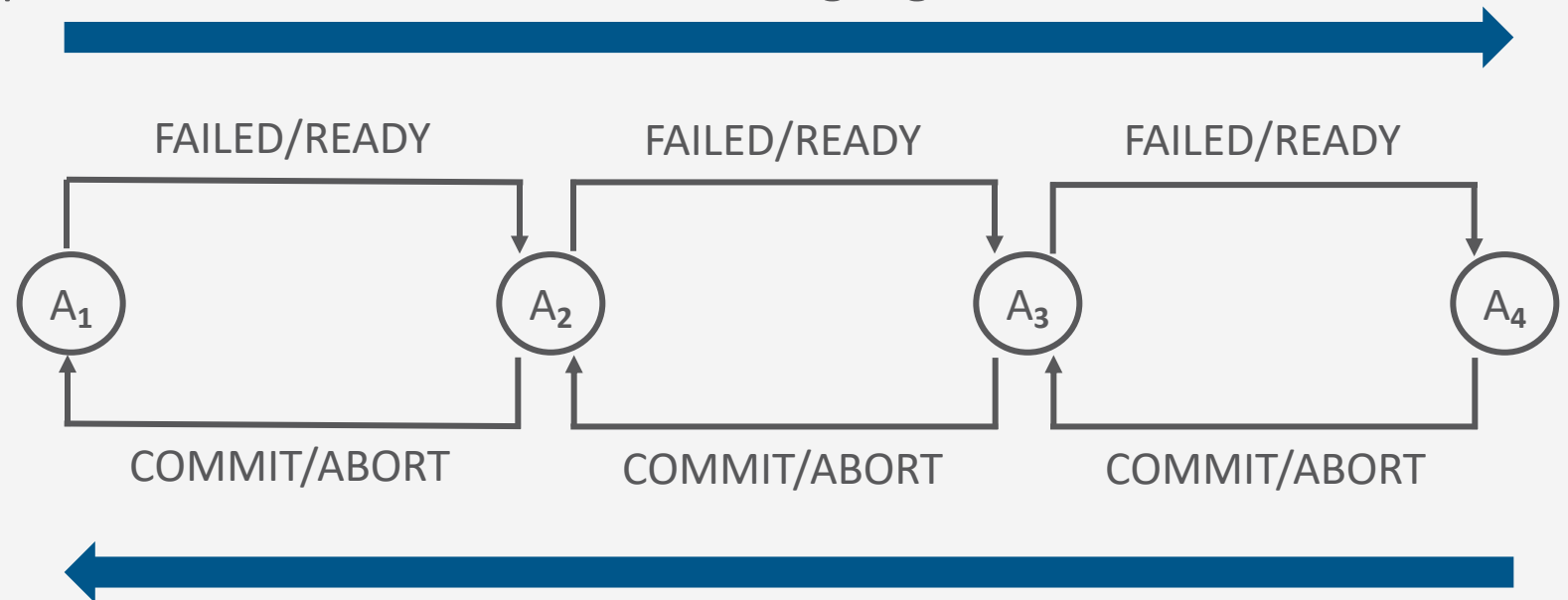
## Agent



„Bullet“ ●  
= wichtigste Aktion(en)

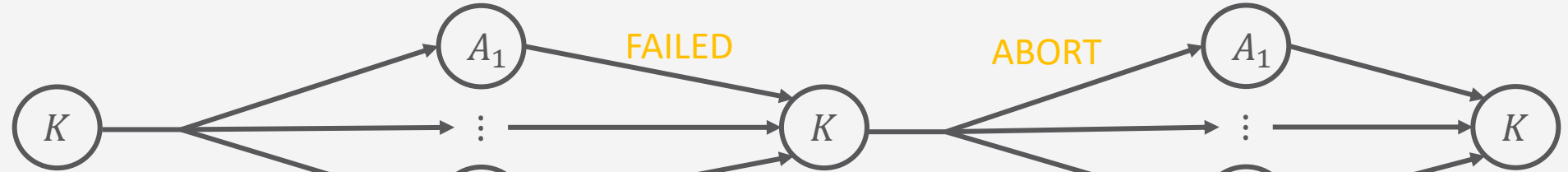
## 2PC-Protokoll ohne Koordinator

- Lineare Organisationsform: Reihenfolge der Agenten festlegen
- Nacheinander kommunizieren
  1. Agenten reichen ihren eigenen Status und den der vorherigen Nachbarn an den nächsten weiter, nachdem sie den entsprechenden Statusbericht vom Vorgänger bekommen haben
  2. Der letzte Agent trifft die Entscheidung und reicht sie zurück

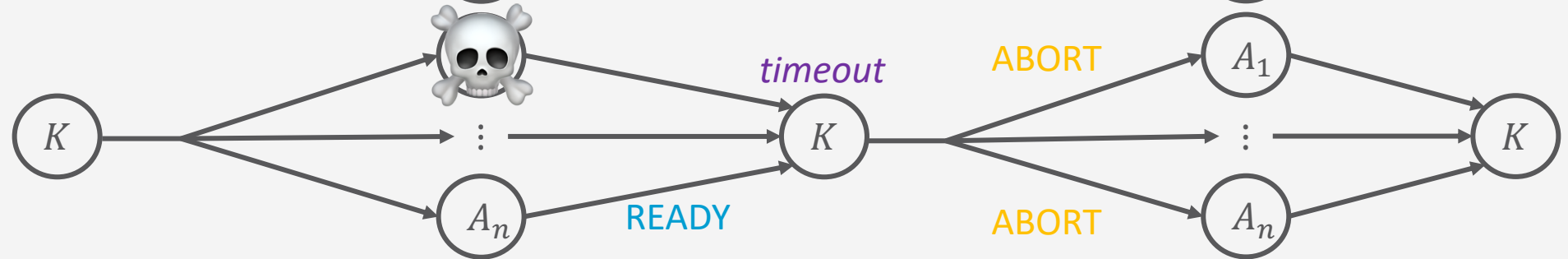


# Problemlösung: Zweiphasen-Commit-Protokoll

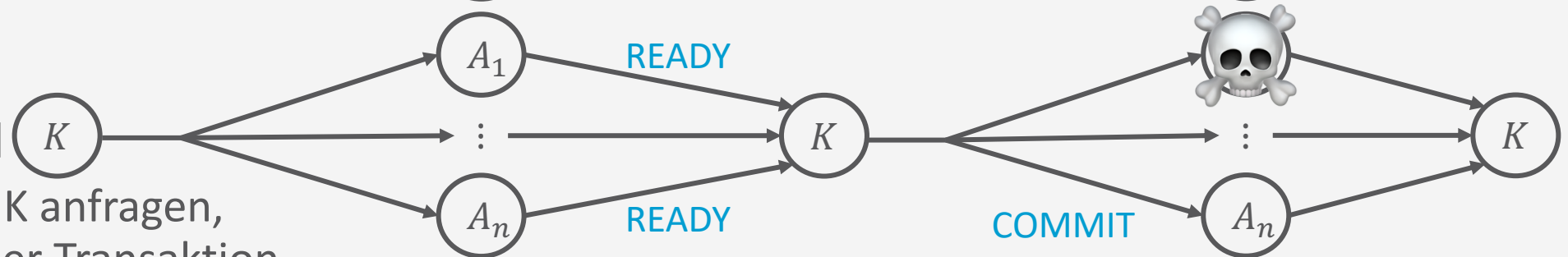
1. Transaktion kann nicht durchgeführt werden



2. Station crasht, bevor READY gesendet wird (timeout bei  $K$ )



3. Station crasht, bevor COMMIT empfangen wird



- Nach Restart bei  $K$  anfragen, ob undo / redo der Transaktion



## 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 (ACK senden)

## 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)

## 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
  - 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
      - $St_1$ : lokal ausgeführte Operationen seriell ausgeführt
      - $St_2$ : lokal ausgeführte Operationen seriell ausgeführt
    - Global nicht serialisierbar
      - $p_i^a$ :  $a$  identifiziert die Station,  $i$  die Transaktion
      - Schedule:  $r_1^1(X), w_2^1(X), w_2^2(X), r_1^2(X)$
      - Nicht umformbar zu seriellen Schedule

$St_1$	$T_1$	$T_2$	$St_2$	$T_1$	$T_2$
1	$r(X)$				
2		$w(X)$			
			3		$w(Y)$
			4	$r(Y)$	

## Sperrverwaltung in VDBMS

- **Lokale Sperrverwaltung:** Globale Transaktion muss vor Zugriff / Modifikation eines Datums  $X$ , 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
      - Verletzung der lokalen Autonomie der Stationen, da auch lokale Transaktionen ihre Sperren bei der zentralisierten Sperrverwaltung anfordern müssen
- Zentrale Sperrverwaltung im Allgemeinen nicht akzeptabel

# Deadlocks in VDBMS

- Erkennung von Deadlocks (Verklemmungen)

- Beispiel: Verteilter Deadlock

- $T_2$  bei  $St_2$  wartet auf Freigabe von  $X$  bei  $St_1$
- $T_1$  bei  $St_1$  wartet auf Freigabe von  $Y$  bei  $St_2$
- Lokale Wartegraphen: zyklensfrei



- Globaler Wartegraph: mit Zyklus



- Vorgehen zur Erkennung:

- Timeout, zentralisiert, dezentral

## Erkennung von Deadlocks: Wartegraph

- Wartegraph: Gerichteter Graph
- Enthält für jede aktive Transaktion  $T_i$  einen Knoten
- Wenn  $T_i$  auf eine Sperre von  $T_j$  wartet:
  - Füge Kante  $(T_i \rightarrow T_j)$  in den Graphen ein (Kante bei Freigabe löschen)
- Weist der Wartegraph Zyklen auf, so liegt ein Deadlock vor

Verteilte DBs


$St_1$	$T_1$	$T_2$	$St_2$	$T_1$	$T_2$
0	BOT				
1	lockS(X)				
2	r(X)				
			3		BOT
			4		lockX(Y)
			5		w(Y)
6		lockX(X)			
		...	7	lockS(Y)	
				...	

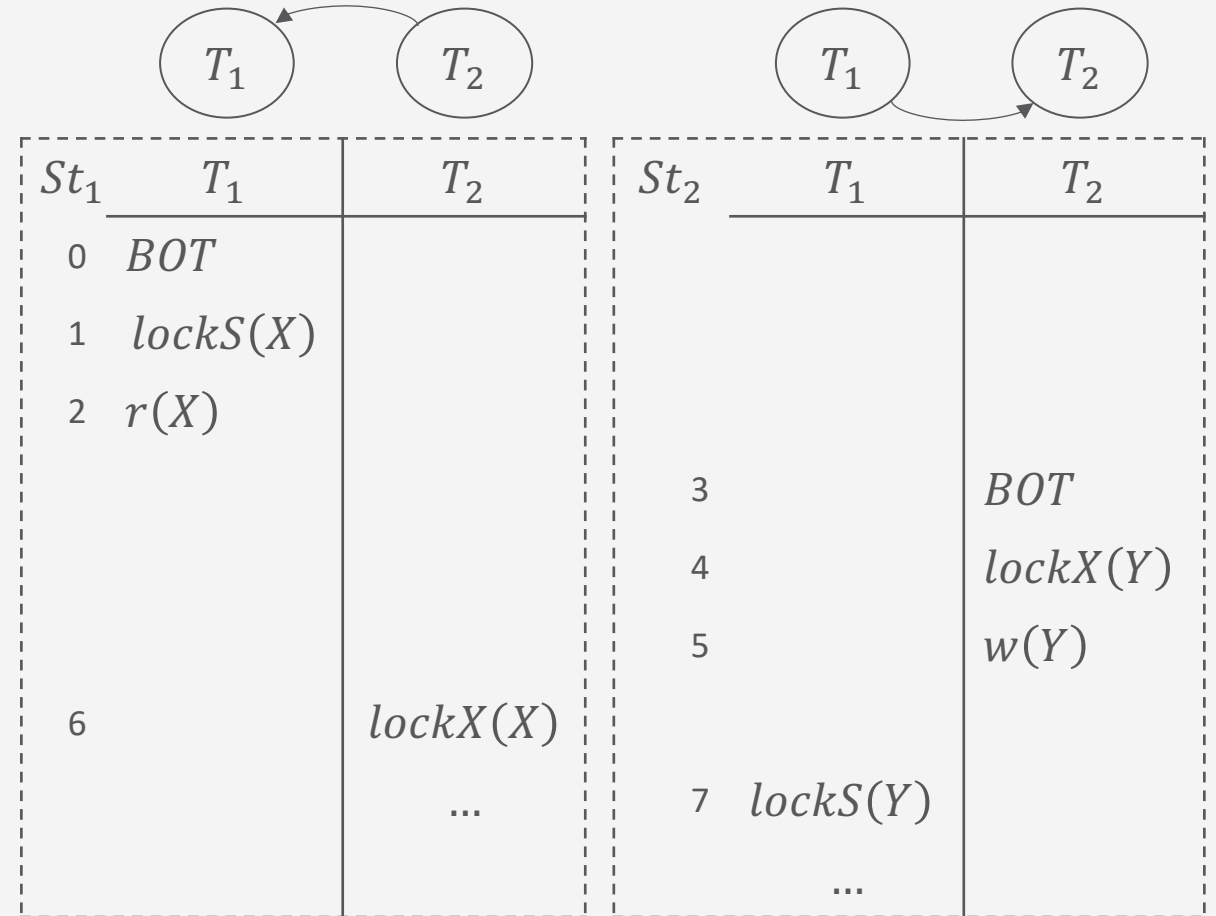
## Erkennung von Deadlocks: Timeout

- 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

$St_1$	$T_1$	$T_2$	$St_2$	$T_1$	$T_2$
0	<i>BOT</i>				
1	<i>lockS(X)</i>				
2	<i>r(X)</i>				
			3		<i>BOT</i>
			4		<i>lockX(Y)</i>
			5		<i>w(Y)</i>
6		<i>lockX(X)</i>			
		...	7	<i>lockS(Y)</i>	
				...	

# Zentralisierte Deadlock-Erkennung

- Stationen melden lokal vorliegende Wartebbeziehungen an neutralen Knoten, der daraus globalen Wartegraphen aufbaut (Zyklus im Graphen → Deadlock)
  - Sichere Lösung
  - Beispiel: Globaler Wartegraph
 
- Nachteile:
  - Hoher Aufwand (viele Nachrichten)
  - Entstehung von Phantom-Deadlocks (= nicht-existierende Deadlocks) durch Überholen von Nachrichten im Kommunikationssystem





## Dezentrale Deadlock-Erkennung: Obermarck's Path Pushing Algorithmus

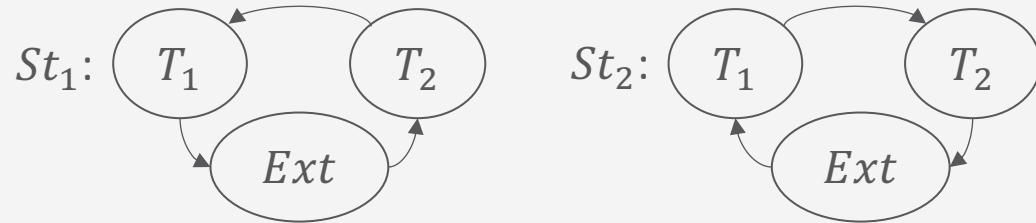
- Zur Erkennung von globalen Deadlocks
- Zuordnung jeder Transaktion zu einem Heimatknoten (Station), von wo aus externe Subtransaktionen auf anderen Stationen initiiert werden
  - Globale Ordnung der Transaktionen (z.B. eindeutiger String mit lexikographischer Ordnung)
- Jede Station hat einen lokalen Wartegraphen, erweitert um einen Knoten *Ext* (*external*)
  - *Ext* modelliert die stationenübergreifenden Wartebeziehungen zu externen Subtransaktionen
- Formal: Wartegraph  $G$  mit initial
  - Für jede Transaktion  $T_i$  ein Knoten  $T_i$  (beheimatete und fremde) mit Kanten wie bisher
    - Kante  $T_i \rightarrow T_j$ , wenn  $T_i$  auf  $T_j$  wartet
  - Dazu einen Knoten *Ext* mit Kanten wie folgt
    - Kante  $Ext \rightarrow T_i$  wird für jede von außen kommende Transaktion  $T_i$
    - Kante  $T_j \rightarrow Ext$  wird für jede Transaktion  $T_j$  dieser Station, falls  $T_j$  nach außen geht

# Dezentrale Deadlock-Erkennung: Obermarck's Path Pushing Algorithmus

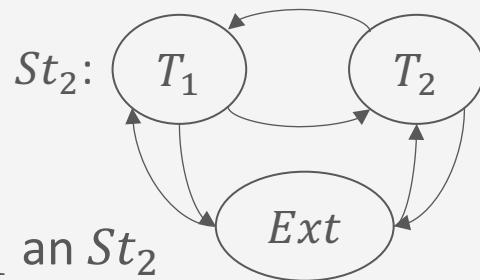
- Vorgehen an jeder Station (asynchron)
  1. Einkommende Informationen aus Schritt 3 von anderen Stationen einbauen
    - Gestoppte Transaktionen  $T_a$  und deren Kanten löschen
    - Knoten  $T_i$  für unbekannte  $T_i$  und Kanten einfügen aus Pfaden, die nicht  $T_a$  beinhalten
  2. Liste von elementaren Zyklen in Wartegraph erstellen
    - Elementare Zyklen: Zyklen, so dass keine Subzyklen enthalten sind
  3. Zyklen bearbeiten
    - Lokale Zyklen (ohne *Ext* Beteiligung) beheben durch Stoppen von Transaktionen
      - Entferne Knoten  $T_i$  sowie verbundene Kanten aus Wartegraph, lösche Zyklen mit  $T_i$  aus Liste
      - Falls  $T_i$  verteilt ausgeführte Transaktion, Information an andere Stationen senden
    - Für alle Zyklen  $Ext \rightarrow T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow Ext$ :
      - Pfad an Stationen schicken, die Subtransaktionen von  $T_n$  gestartet haben
      - Nachrichtenaufkommen reduzieren: Nur schicken, wenn  $T_1 < T_n$

# Dezentrale Deadlock-Erkennung: Beispiel

- $St_1$  Heimatknoten von  $T_1$ ,  $St_2$  Heimatknoten von  $T_2$ ; Ordnung  $T_1 < T_2$
- Wartegraphen



- $St_1$  Pfad  $Ext \rightarrow T_2 \rightarrow T_1 \rightarrow Ext: T_1 \not< T_2$
- $St_2$  sendet Pfad  $Ext \rightarrow T_1 \rightarrow T_2 \rightarrow Ext$  an  $St_1$ 
  - $St_1$ : Lokaler Zyklus
    - $T_1$  stoppen
    - Zyklenfrei
    - Info über  $T_1$  an  $St_2$
- $St_1$  sendet stop von  $T_1$  an  $St_2$ 
  - Nach Einbau Wartegraph zyklenfrei



$St_1$	$T_1$	$T_2$	$St_2$	$T_1$	$T_2$
0	<i>BOT</i>				
1	<i>lockS(X)</i>				
2	<i>r(X)</i>				
			3		<i>BOT</i>
			4		<i>lockX(Y)</i>
			5		<i>w(Y)</i>
6		<i>lockX(X)</i>			
		...	7	<i>lockS(Y)</i>	
				...	

# Dezentrale Deadlock-Erkennung

- Path Pushing Algorithmus hat seine Nachteile
  - Viele Nachrichten, Phantom-Deadlocks
- Weitere Ansätze, aber weiterhin offenes Forschungsproblem
  - **Edge Chasing** (entlang virtueller Kanten des Wartegraphen)
    - Wartende Transaktion sendet eine Nachricht an die Transaktion, auf die sie wartet
    - Transaktion leitet solch eine Nachricht an andere Transaktionen weiter, auf die sie selber wartet
    - Kommt die Nachricht an die Initiatorin zurück → Deadlock / Zyklus
  - **Deadlock Detection Agents (DDAs)**
    - DDA zuständig für eine Zusammenhangskomponente (ZHK) in einem Wartegraphen
    - Wenn Kante dazukommt, die ZHKs verbindet, verschmilzt jüngerer DDA mit älterem DDA
    - Irgendwann wird die gesamte ZHK des Wartegraphen (in der sich der Zyklus befindet) in einem DDA landen, der den Zyklus dann entdeckt

# Deadlock-Vermeidung

- Optimistische Mehrbenutzersynchronisation:  
Nach Abschluss der Transaktionsbearbeitung wird Validierung durchgeführt
- Zeitstempel-basierte 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)
- Sperrbasierte Synchronisation
  - **wound/wait**: Nur jüngere Transaktionen warten auf ältere (wie vorher)
  - **wait/die**: Nur ältere Transaktionen warten auf jüngere (wie vorher)
- Voraussetzungen für Deadlockvermeidungsverfahren:  
Vergabe global eindeutiger Zeitstempel als Transaktionsidentifikatoren
  - Lokale Uhren müssen hinreichend genau aufeinander abgestimmt sein

lokale Zeit	Stations-ID
-------------	-------------

## Synchronisation bei replizierten Daten: Read One, Write All (ROWA)

- Zu einem Datum  $X$  gibt es Kopien  $X_1, \dots, X_n$ , die auf unterschiedlichen Stationen liegen
- Eine logische Leseoperation auf einer beliebigen Kopie durchführen
  - Z. B. die am nächsten Netzwerk verfügbare Kopie
- Eine logische Schreiboperation wird zu physischen Schreiboperationen auf allen Kopien
  - Synchrones Schreiben aller Kopien in derselben Transaktion
  - 2PL sperrt alle Kopien von  $X$
  - 2PC um ein Commit für die Schreiboperation atomar auf allen Kopien durchzuführen
- Vorteil: Sehr einfach zu implementieren, da es nahtlos mit den Protokollen zusammenpasst; effizientes Lesen; alle Kopien auf gleichem Stand
- Nachteil: Hohe Laufzeit, Verfügbarkeitsprobleme (alle kopienhaltenden Stationen stehen in Abhängigkeit; Änderungstransaktionen werden  $n$  mal so lang  $\rightarrow$  hohe Deadlock-Rate)

## Synchronisation bei replizierten Daten: Quorum-Consensus Verfahren

- Idee: Änderung auf einer Kopie wird nur dann ausgeführt, wenn die Transaktion in der Lage ist, die Mehrheit der Kopien dafür zu gewinnen (z.B. geeignet zu sperren)
  - Gewichtung der individuellen Stimmen: Entweder eine Stimme pro Kopie oder Gewichte pro Stimme (z.B. mehr Gewicht für hoch verfügbare Stationen)
  - Anzahl der Stimmen, die für das Erreichen der Mehrheit erforderlich sind: Statisch (fest vorgegeben), dynamisch (nur Stationen, die beim letzten Update dabei waren, haben Stimme)
- Für Lesezugriffe (Lesequorum  $Q_r$ ) und Schreibzugriffe (Schreibquorum  $Q_w$ ) können unterschiedliche Anzahlen von erforderlichen Stimmen festgelegt werden
  - Ein-Kopie-Serialisierbarkeit bei  $Q_w + Q_w > Q$  und  $Q_w + Q_r > Q$ ,  $Q$  Gesamtgewicht
    - Dazu Zeitstempel pro Kopie; beim Lesen: Zeitstempel prüfen, neueste Kopie lesen
      - Durch  $Q_w + Q_r > Q$  müssen Lese- und Schreiboperationen überlappen, weswegen Lese-Operationen nicht nur veraltete Kopien lesen (mindestens eine ist aktuell)
- Auch ohne Sperren möglich (Stationen im Ring angeordnet); Erweiterung: Tree Quorum

## Zwischenzusammenfassung

- Anfrageoptimierung unter Berücksichtigung der Fragmentierung
- Verteilte Join-Auswertung aufwendig, da u.U. viele Nachrichten nötig
- Wiederherstellung: redo / undo
- EOT-Behandlung durch 2PC-Protokoll
  - Fehlersituationen: Absturz eines Agenten im Rahmen des Protokolls behandelbar, Absturz des Koordinators u.U. schwierig (Agenten blockiert), verlorengegangene Nachricht (Kordinator wartet behandelbar; Agent wartet → Agent blockiert)
- Transaktionsverwaltung
  - Lokale Serialisierbarkeit bedeutet keine globale Serialisierbarkeit
  - Sperrverwaltung: Lokale Sperren einfach, globale Sperren sicherer aber schwierig umzusetzen
  - Deadlockerkennung schwierig: dezentrale Lösung existiert, aber aufwendig
  - Synchronisation bei replizierten Daten schwierig



# Überblick: 8. Verteilte Datenbanken

## A. *Verteilte DBMS*

- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

## B. *Anfragenbeantwortung in verteilten Systemen*

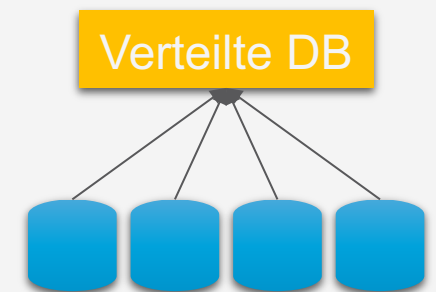
- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

## C. **Föderierte DBS**

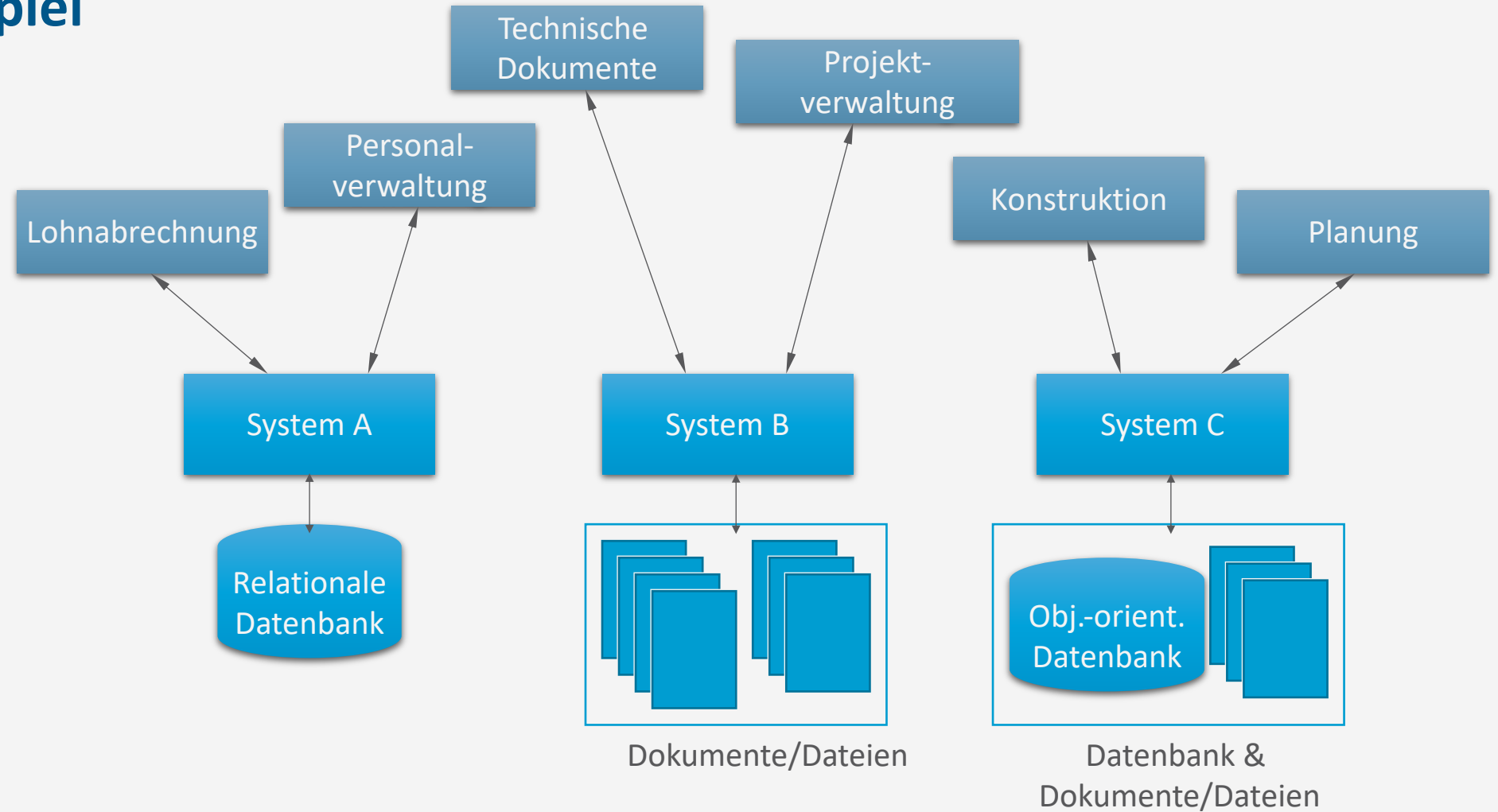
- Integration
- Migration

## Anwendungsbeispiel

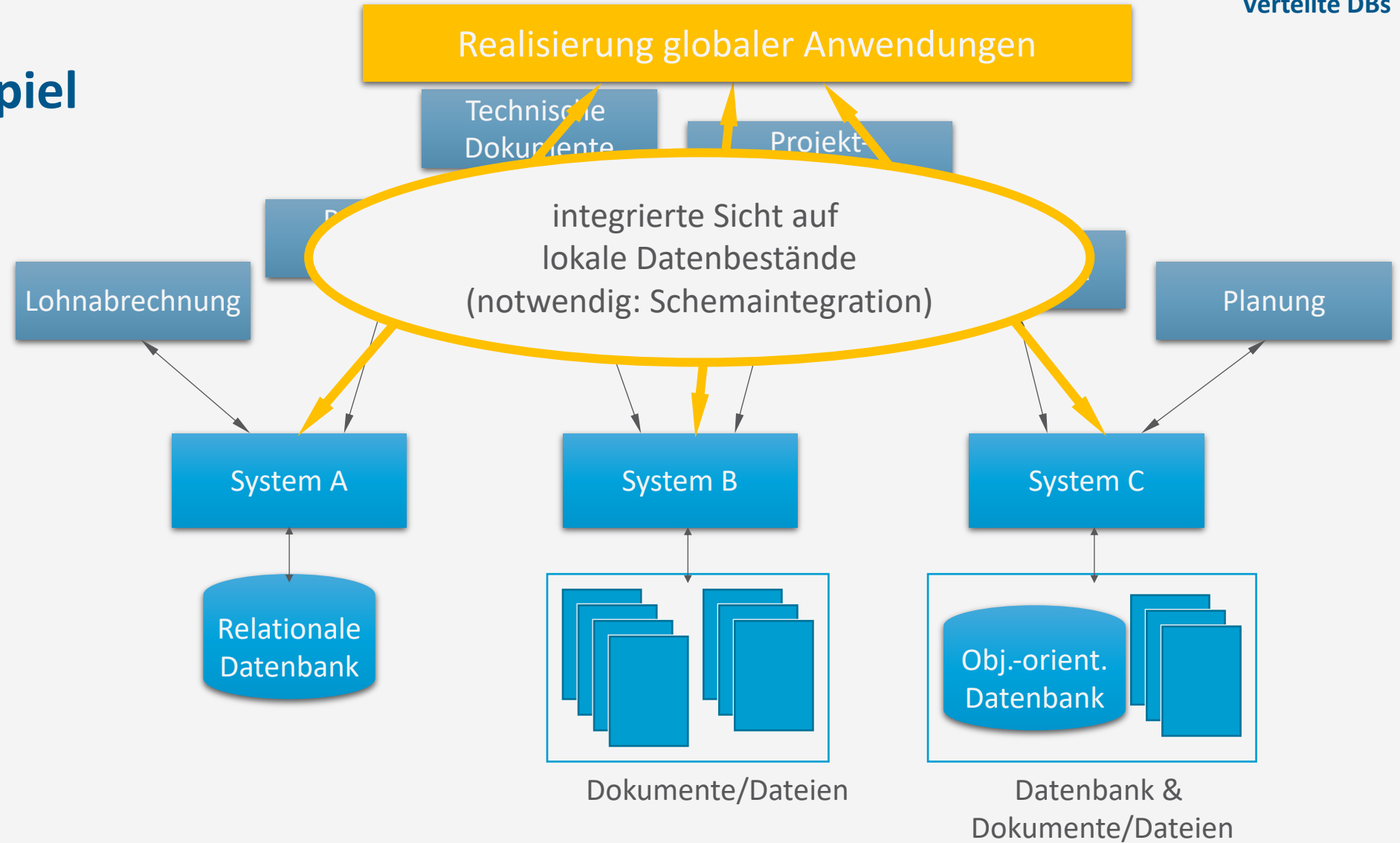
- Typische Situation in Unternehmen und Organisationen:
  - Unabhängig voneinander entstandene Datenbestände in verschiedenen Abteilungen / Arbeitsgruppen
    - Unterschiedliche Strukturierung gleich(artig)er Daten
    - Teilweise redundante Datenhaltung
    - Global inkonsistente Datensituation möglich
  - Autonome, nicht abgestimmte Entscheidungen über Anschaffung von Hardware und Software
- Entstehung von „Insellösungen“, die einen einheitlichen Zugriff auf verteilte Daten erschweren oder sogar unterbinden



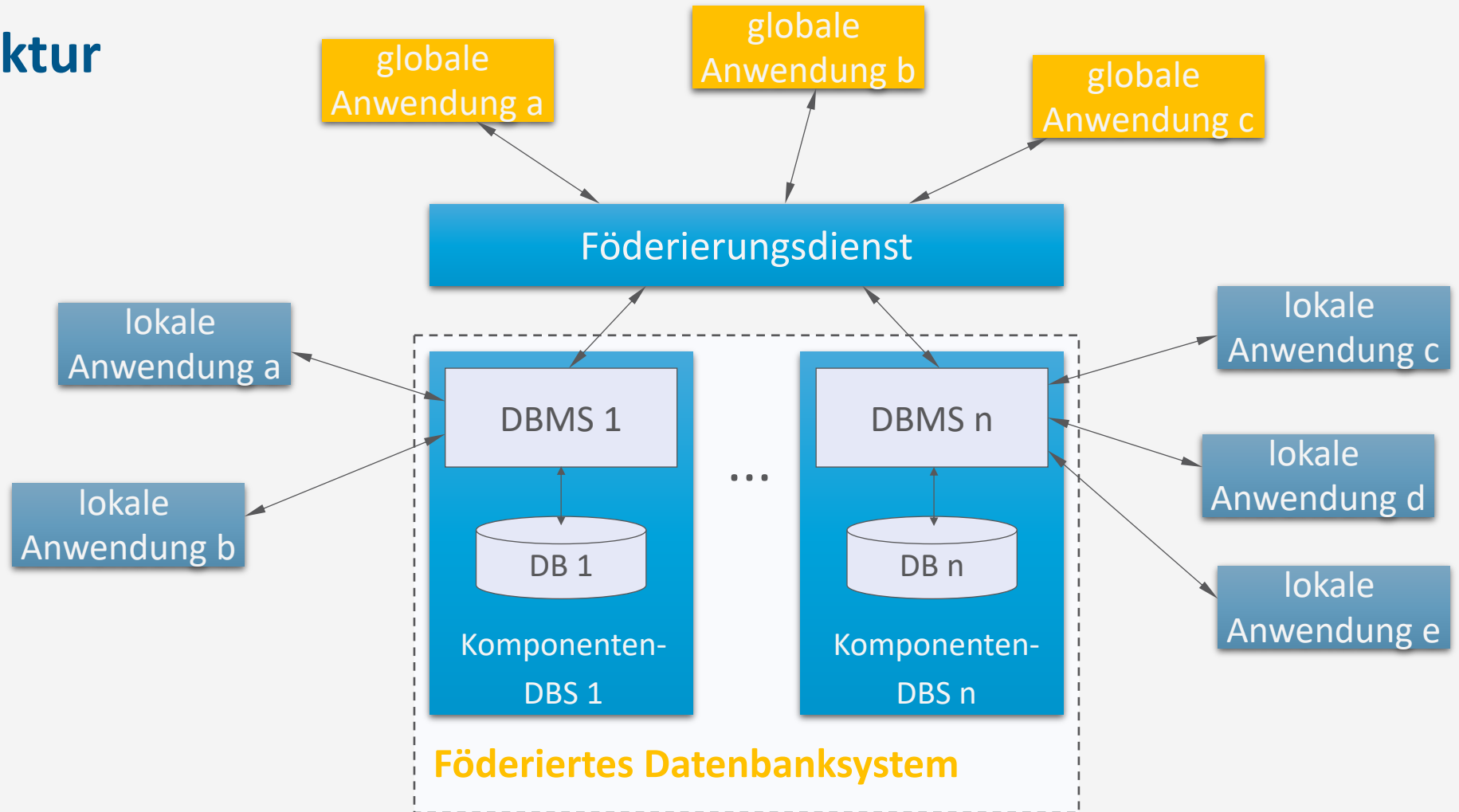
# Anwendungsbeispiel



# Anwendungsbeispiel



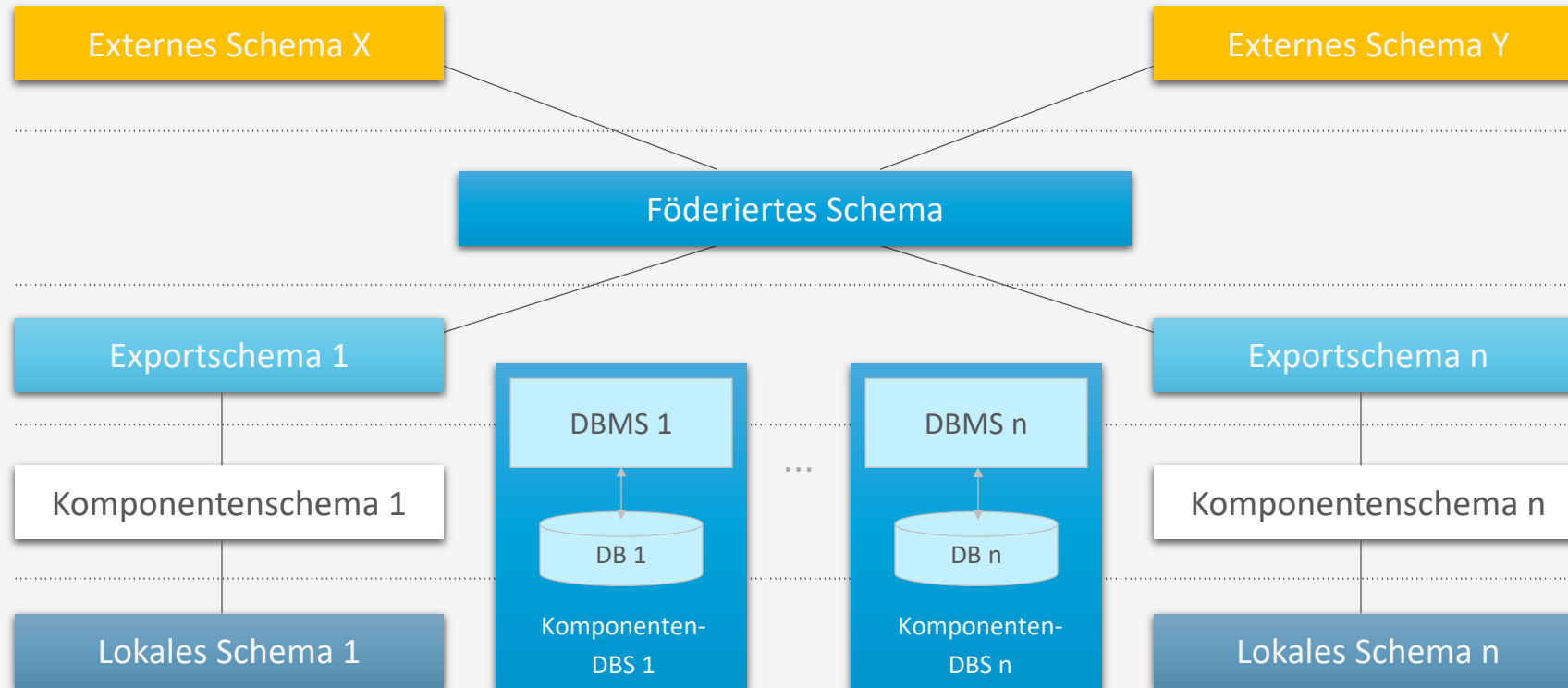
# Föderierte DBS: Allgemeine Architektur



## Was soll ein föderiertes DBS können?

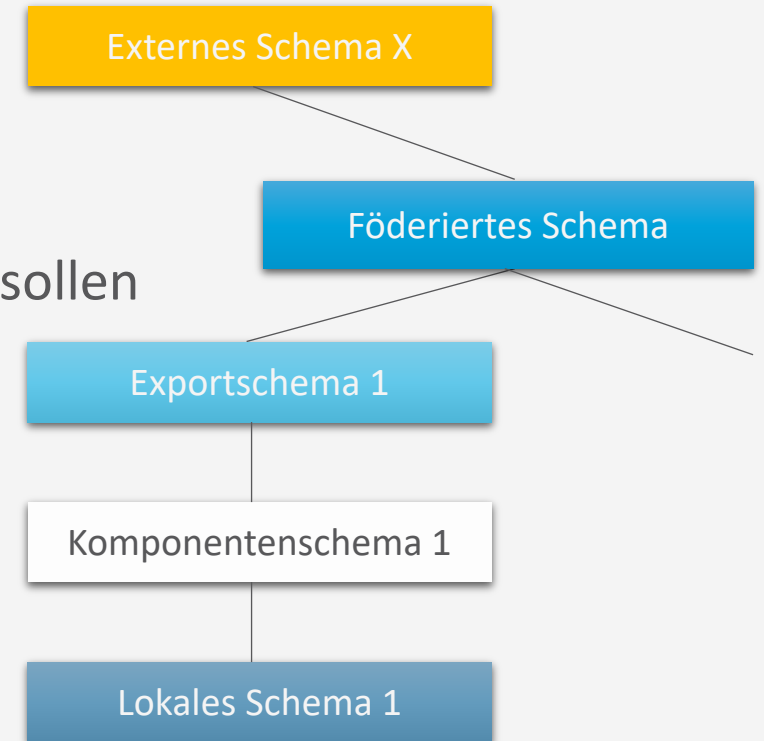
- Im Idealfall volle Funktionalität eines DBS, dabei Verbergen der Verteilung bzgl.
  - Transaktionsverwaltung
  - Integritätskontrolle
  - Anfrageverarbeitung und -optimierung
  - Recoveryauf globaler Ebene (im Föderierungsdienst) → Insgesamt hohen Aufwand
- Pragmatischer Ansatz:
  - Nur solche Funktionalität im föderierten System realisieren, die tatsächlich benötigt wird – eine stark anwendungsgetriebene Diskussion

# 5-Schichten-Architektur



# Schemaintegration

- Lokales Schema
  - Datenschema des lokalen DBS
- Komponentenschema
  - Schema des lokalen DBS im globalen Datenmodell
- Exportschema
  - Teile des lokalen Schemas, die in Föderation eingebracht werden sollen
- Föderiertes Schema
- Externes Schema
  - Schema für einzelne Anwendungen bzw. Benutzer





# Schemaintegration

- Vorgehen
  - Schematransformation:
    - Übersetzung der lokalen Schemata in Komponentenschemata (semi-automatisch oder interaktiv anhand von Transformationsregeln)
  - Verhandlung (unter Domänenexperten und Datenbankdesignern):
    - Anwendungsorientierte Diskussion der angestrebten Exportschemata
  - Eigentliche Schemaintegration:
    - Vereinigung der Exportschemata zu föderiertem Schema

- Probleme
  - Synonyme: gleiche Bedeutung bei unterschiedlicher Benennung
  - Homonyme: gleiche Namen bei unterschiedlicher Bedeutung
  - Ähnlichkeit, Spezialisierung, Überschneidung

Forschungsgebiet u.a.: **Ontology-based data access (OBDA)**  
Gemeinsame Ontologie definieren, für Übersetzung von globalen Anfragen in lokale Anfragen / Zusammenstellung der Einzelergebnisse in ein Gesamtergebnis

- Anwendung z.B. in der Zusammenführung von medizinischen Daten aus unterschiedlichen Krankenhausinformationssystemen

# Schemaintegration → Datenintegration

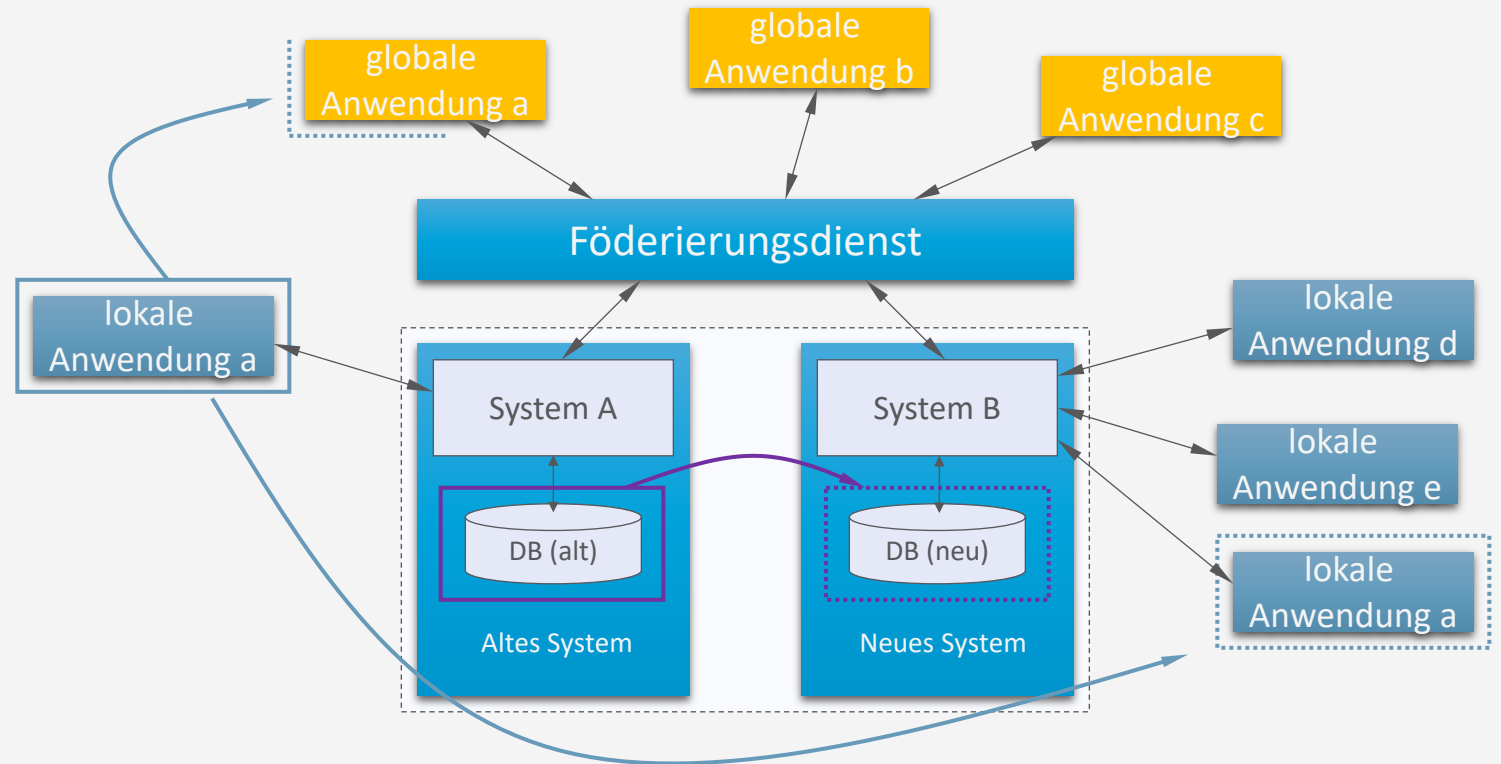
	Semantische Ebene	Zu behebbende Inkonsistenz	Angewandte Methoden
1	Schema	Schematische Heterogenität	Integration Mapping Matching
2	Tupel (Object)	Duplikate	Ähnlichkeitsmaße Partitionierungsstrategien
3	Wert	Datenkonflikte	Datenreinigung Transformationen Fusion

## Beispiele für föderierte DBS

- Ablösung von Altsystemen (legacy systems) – z.B. wegen ...
  - Ansteigender Wartungskosten
  - Nachlassender Unterstützung durch Systemhersteller
  - Überlastung des Altsystems in vorliegender bzw. perspektivischer Anwendungssituation – keine ausreichenden Skalierungsmöglichkeiten
  - Reorganisation betrieblicher Strukturen – z.B. aufgrund einer Fusionen mit anderem Unternehmen, die letztlich die Vereinheitlichung der Systemplattformen notwendig macht
- Wichtige Anforderungen dabei
  - Investitionsschutz
    - Erhalt von Daten
    - Erhalt von Anwendungsprogrammen
  - Fortlaufender Betrieb während Migration

# Migration in kleinen Schritten

- Für fortlaufenden Betrieb
  - System B aufsetzen, Fördererungsdienst definieren
  - Lokale Anwendung a in globale Anwendung a übersetzen
  - Daten aus System A übertragen
  - Lokale Anwendung a an System B überführen, dann alte lokale Anwendung a und globale Anwendung a löschen



## Zwischenzusammenfassung

- Föderierte Datenbanken
  - Einheitliche Sicht auf unterschiedliche DBs
  - Integration
  - Systematische Migration
  - ... von Daten und Anwendungen

## Überblick: 8. Verteilte Datenbanken

### A. *Verteilte DBMS*

- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

### B. *Anfragenbeantwortung in verteilten Systemen*

- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

### C. *Föderierte DBS*

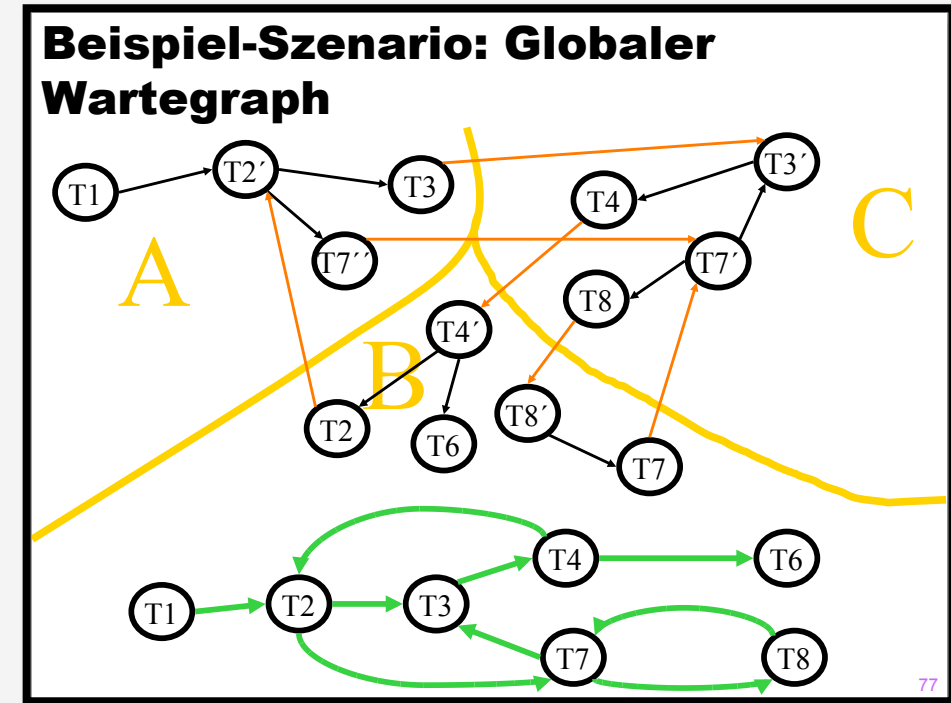
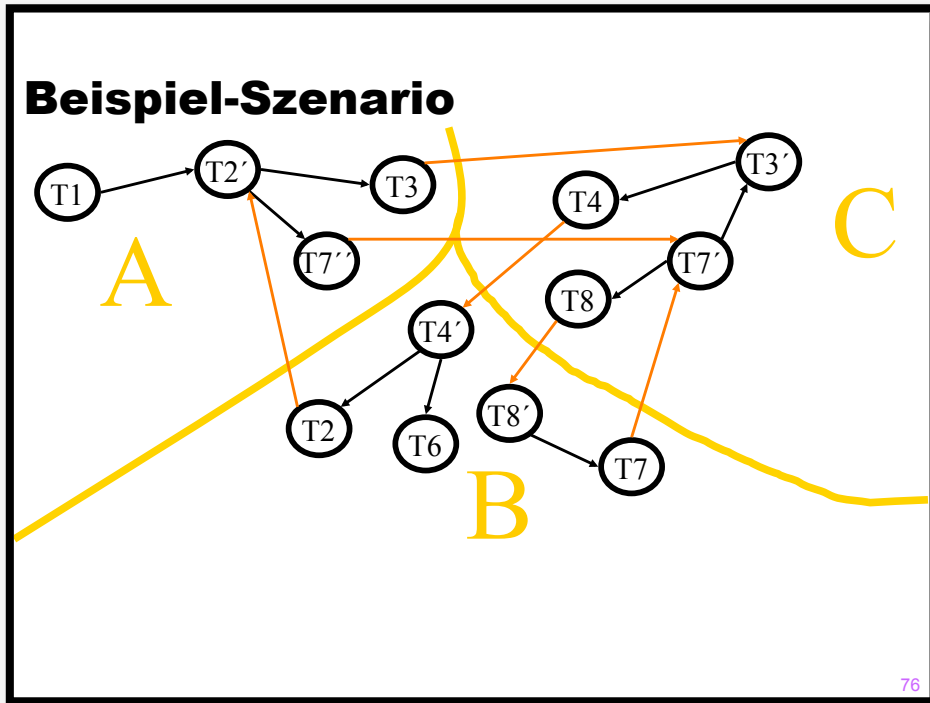
- Integration
- Migration

*Ende*

# Anhang

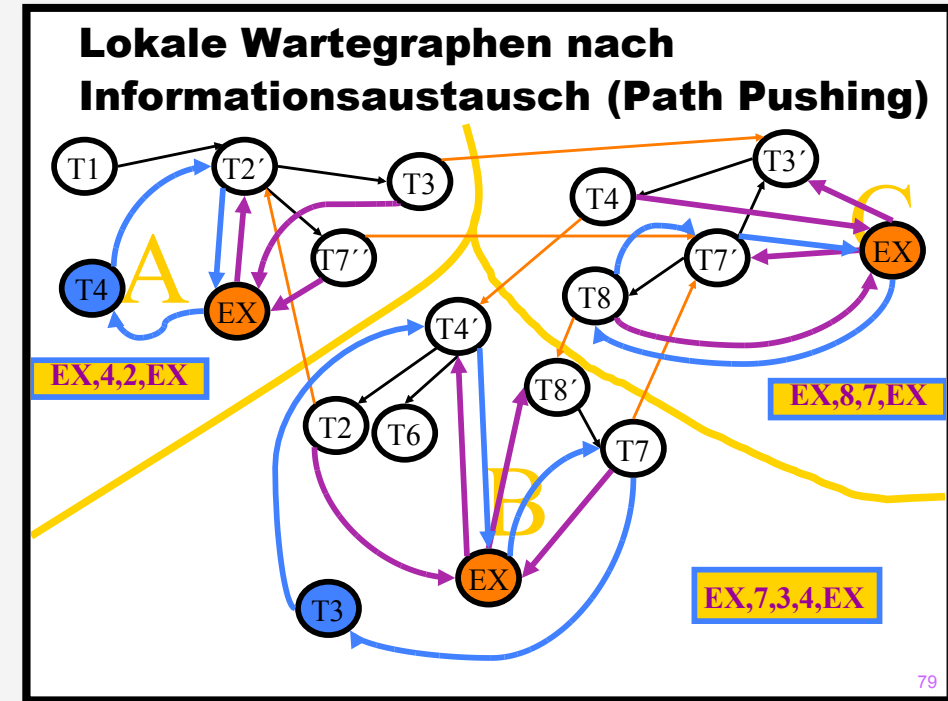
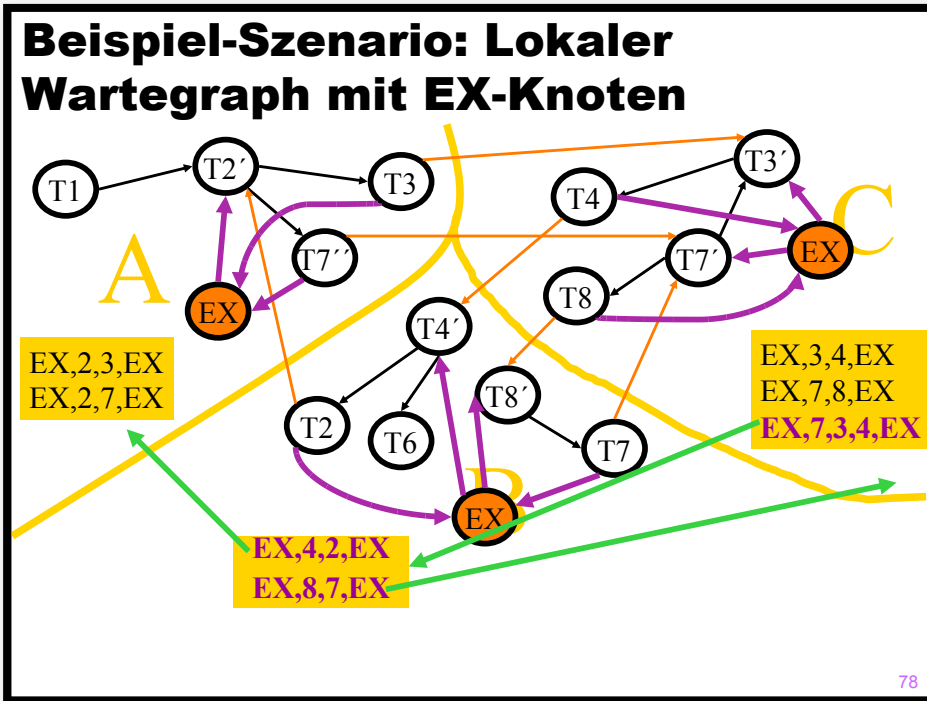
Dezentrale Deadlock-Erkennung: Größeres Beispiel

# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )

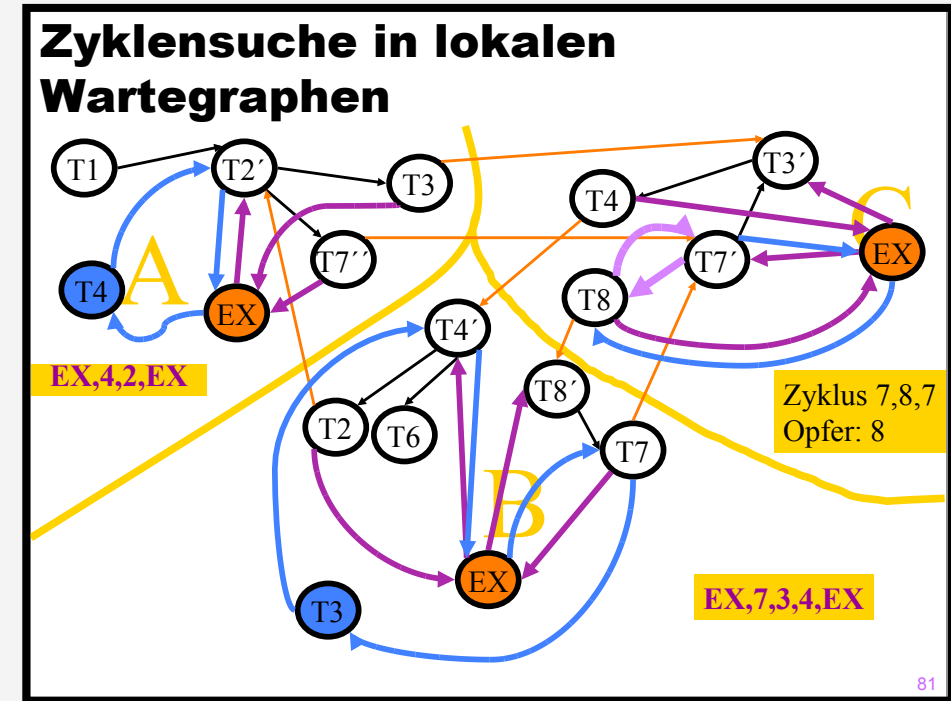
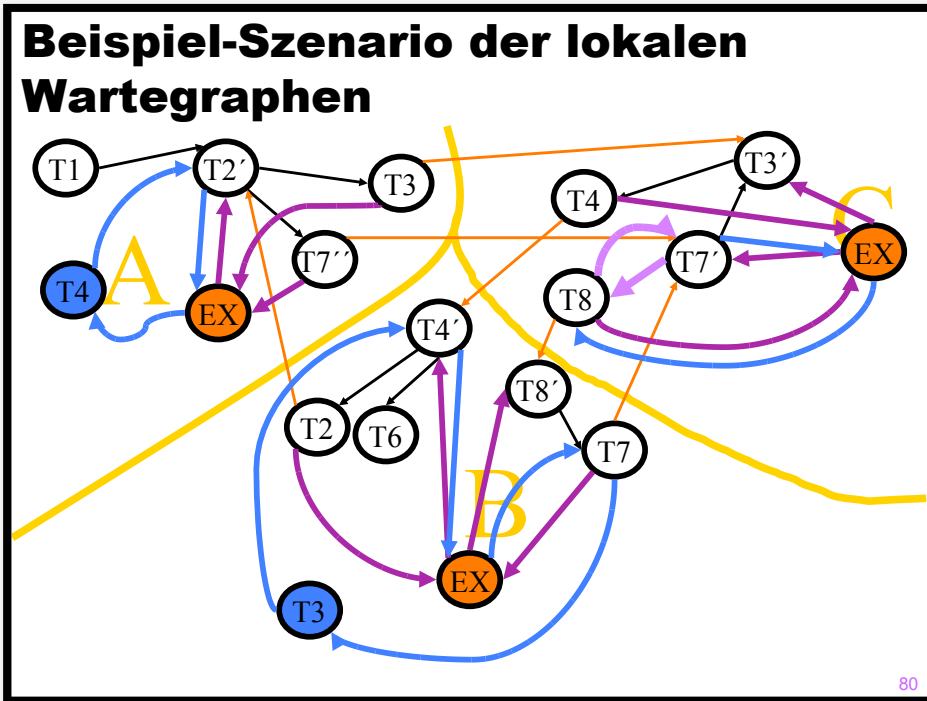




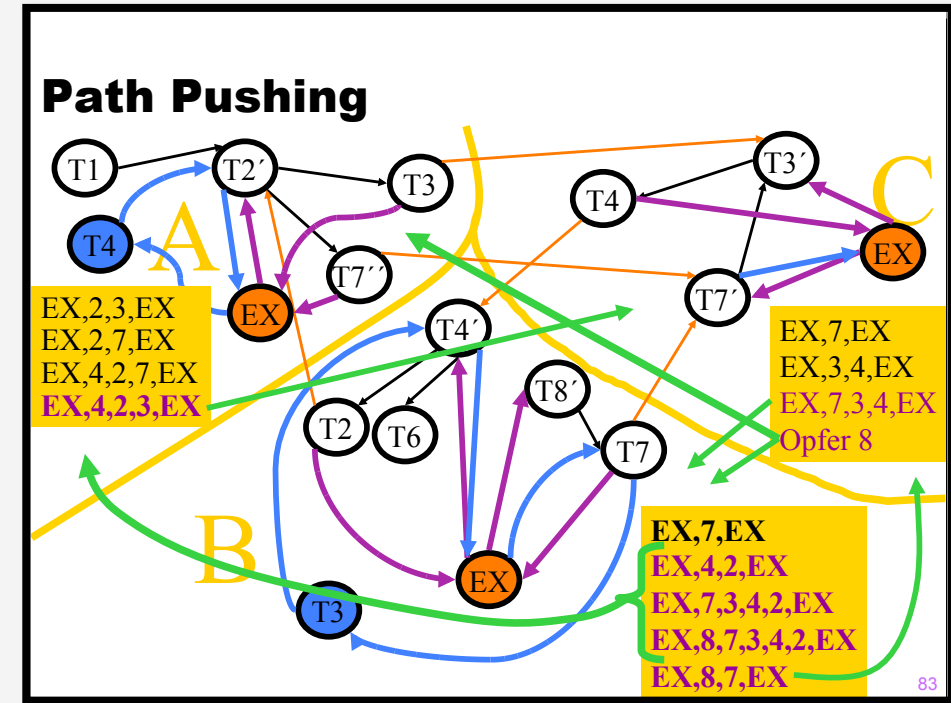
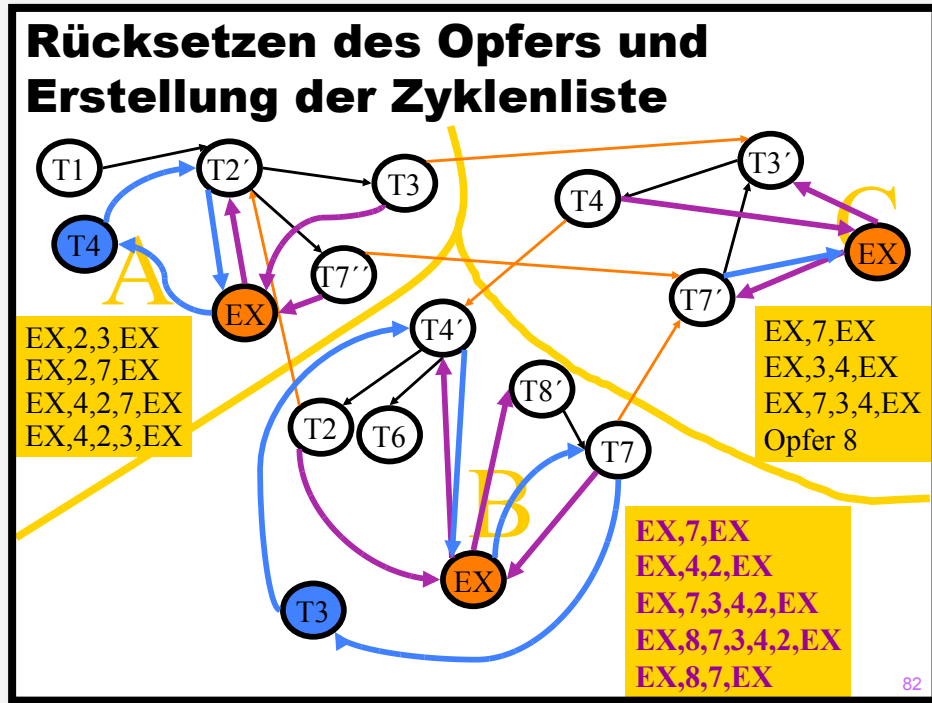
# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



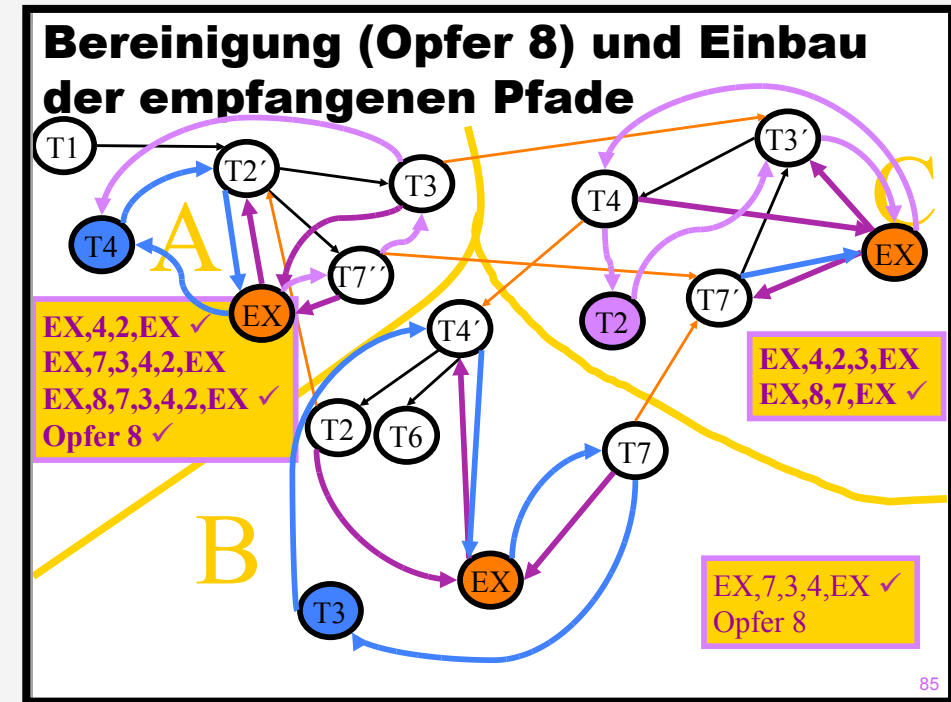
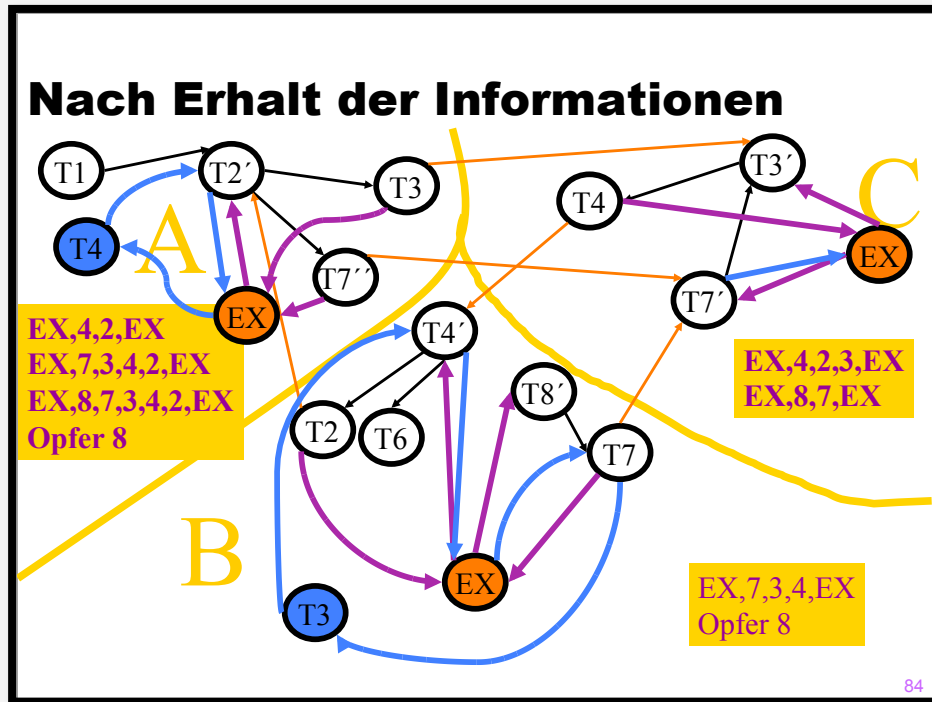
# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



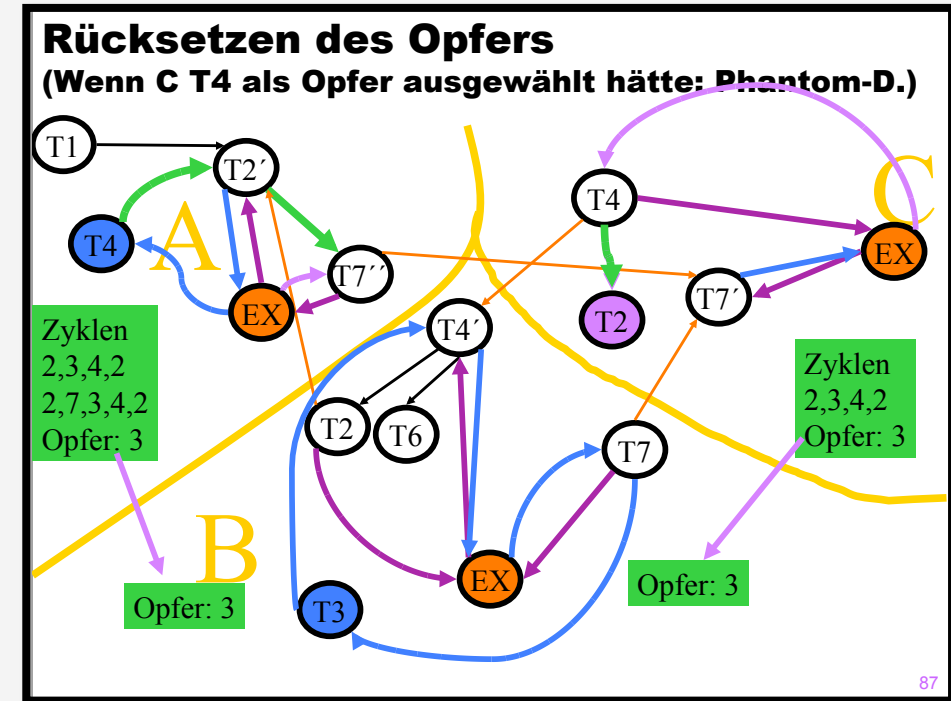
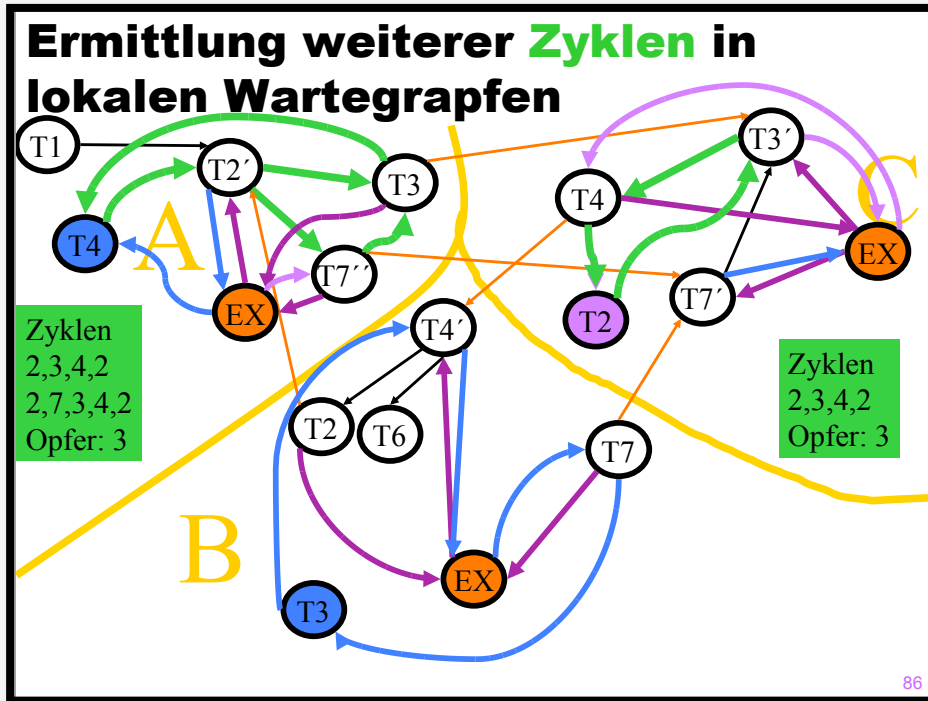
# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



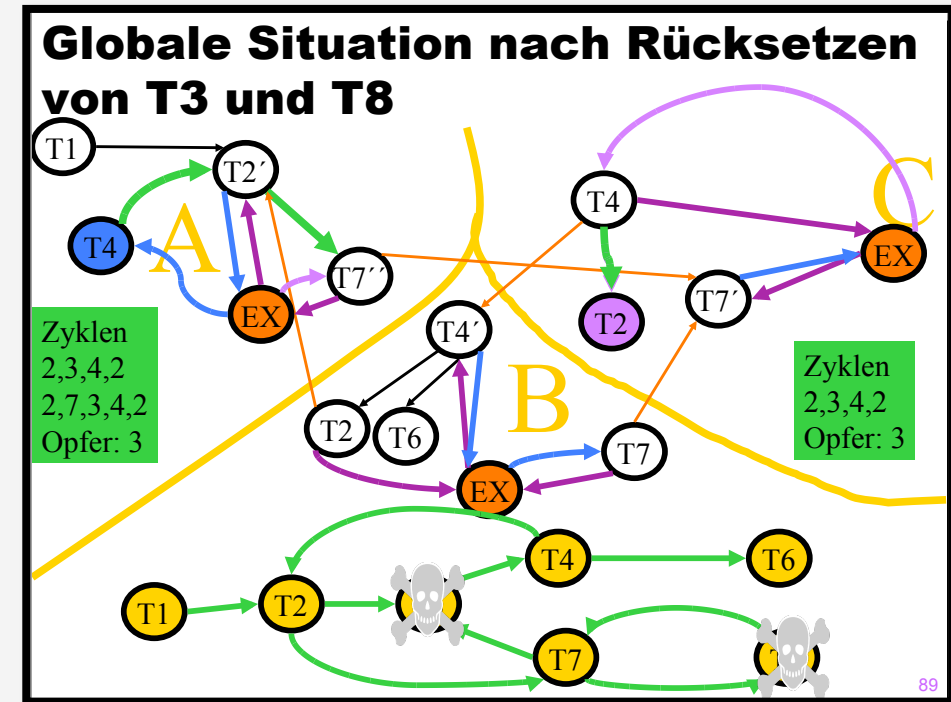
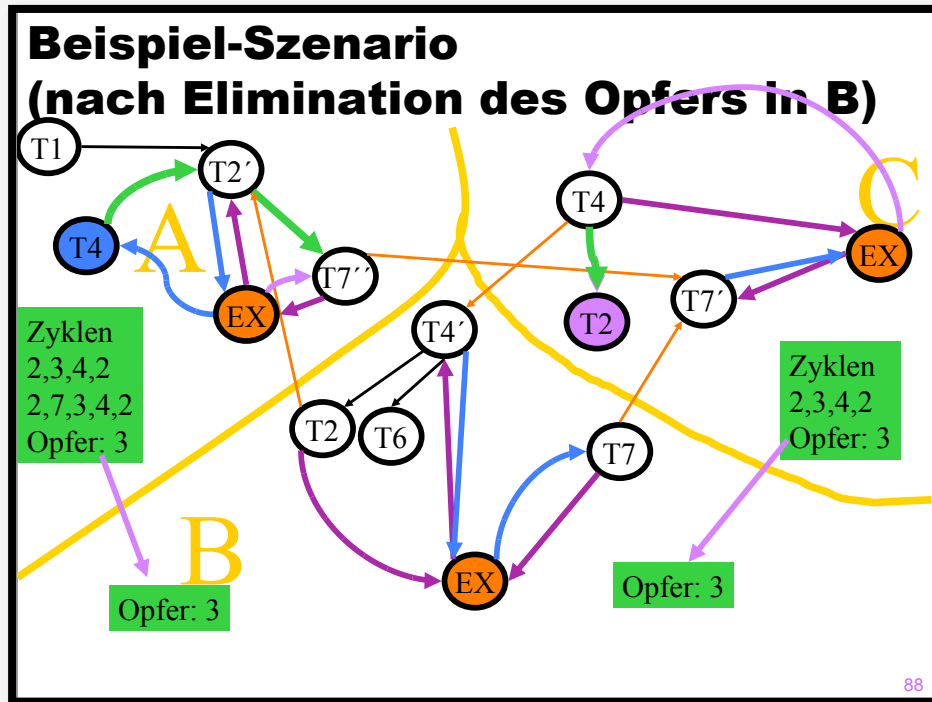
## Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )



# Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$ )

