

WS2002 – Verteilte Datenbanksysteme

Prof. K. Dittrich

Inhalt

1	vDBS - Einführung.....	2
2	Begriffe	5
3	Kommunikation und Rechnernetze	5
4	Transparenz	6
5	DBMS-Standardisierung.....	7
6	Architekturmodelle für vDBMS	7
7	Distribution.....	11
8	Fragmentierung	12
9	Allokation	15
10	Bearbeitung verteilter Anfragen.....	15
11	Transaktionen.....	20
12	Weitere Entwicklungen.....	26

Dieses Dokument ist eine Reinschrift meiner Vorlesungsnotizen der Vorlesung „Verteilte Datenbanksysteme“ des Wintersemesters 2002/03 bei Prof. K. Dittrich an der UniZH. Die Notizen wurden vervollständigt durch die betroffenen Kapitel des Buches *Principles of Distributed Database Systems, 2nd Edition* von M. Tamer Özsu / Patrick Vaslduriez.

Seitenangaben beziehen sich auf die 2nd Edition 1999, Prentice-Hall. Diese Vorlesungsnotizen sind jedoch nicht als Zusammenfassung des Buches gedacht. Einige Kapitel wurden nicht behandelt.

1 vDBS - Einführung

DBS	anwendungsspezifische Daten (Dateien) → zentral definierte/verwaltete Daten
Datenunabhängigkeit	<ul style="list-style-type: none"> • Integrierte Datenbestände • Redundanzarmut (nicht automatisch!) • überwachte Verwendung
Rechnernetze	
Integration / Zentralisation	<p>muss nicht physische Zentralisation sein!</p> <p>Integration (log.) impliziert nicht Zentralisation (phys.) Zentralisation (phys.) impliziert nicht Integration (log.)</p>
vDBS	<p>a) per Konstruktion von Grund auf verteilt → viel einfacher</p> <p>b) per Tradition mehrere (bestehende) zentrale DBS zu einem einzigen zusammenfügen → meist viel komplexer, da bereits bestehende Datenbestände/Schemen, die zusammengefügt werden müssen.</p>
„verteilt“	<p>Was ist „verteilt“?</p> <ul style="list-style-type: none"> ✓ Verarbeitungslogik (Prozessoren) ✓ Funktionen (→Dedizierung) ✓ Daten ✓ Steuerung (control)

Klassifizierungskriterien

- a) **Kopplungsgrad** klein/eng(stark)
ausgetauschter Daten / # lokaler Verarbeitung je Auftrag
- b) **Verbindungsstruktur**
Punkt-zu-Punkt
gemeinsamer Verbindungskanal („bus“)
- c) **Interdependenz** schwach/stark
(wechselseitige Abhängigkeit von Verarbeitungselementen bei der Durchführung von Aufgaben)

schwach: z.B. unabhängige Arbeit, Resultat liefern ohne abhängig von anderen zu sein.
- d) **Synchronisation** synchron / asynchron
(zw. Verarbeitungselementen)

Warum vDBS ?

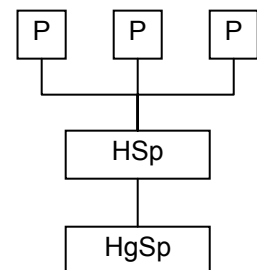
- ✓ **Organisatorische Gegebenheiten**
- ✓ **Höhere Zuverlässigkeit** (mehrere Knoten), Ausfalltoleranz
- ✓ **Effizienz** (Parallelität; Problem: Koordinationsaufwand)
- ✓ **Dateneingabe/-speicherung am Entstehungsort**
→ spart Kommunikationskosten, evtl. Sicherheitsprobleme
- ✓ **Preis-/Leistungsvorteile**
- ✓ **Gemeinsame Datennutzung**

globaler: „Teile und Herrsche“

Multiprozessor-DBS

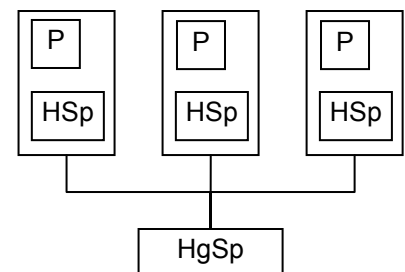
eng gekoppelt:

shared everything

**lose gekoppelt:**

(teilen sich HSp nicht, jedoch HgSp)

shared something



Vor/Nachteile vDBS

p. 19

+	<ul style="list-style-type: none"> ✓ lokale Autonomie ✓ bessere Leistungsfähigkeit ✓ höhere Zuverlässigkeit/Verfügbarkeit ✓ Wirtschaftlichkeit ✓ Erweiterbarkeit ✓ gemeinsame Datennutzung ✓ Grosh: $\Delta \text{Leistung} \approx (\Delta \text{Preis})^2 \rightarrow$ veraltet, heute eher umgekehrt
-	<ul style="list-style-type: none"> ✓ Mangelnde Erfahrung ✓ Komplexität ✓ Kosten ✓ verteilte Kosten ✓ Sicherheit (mehrere Zugangspunkte) ✓ Konversion (v.a. bei „per Tradition“) <p><i>Zusatzfaktoren für Komplikationen:</i></p> <ul style="list-style-type: none"> → Datenreplikation → Knotenausfall/Netzausfall → Synchronisation

Problembereiche

p. 20

- **Entwurf**
Partitionierung / Replikation
Fragmentierung / Allokation (welches Fragment an welchen Knoten?)
- Bearbeitung **verteilter Anfragen**
- Verwaltung **verteilter Kataloge**
- verteilte **Konkurrenzsteuerung**
- Behandlung **verteilter Verklemmungen (Deadlocks)**
- **Zuverlässigkeit**
(graceful degradation: gnädiges Heruntergehen)
- **BS-Unterstützung**
- **heterogene DBMS**

2 Begriffe

DB

- DB ::= Menge von Relationen
- Relation
- Schlüssel (→ Eindeutigkeit gewährleisten)
- DB-Entwurf
- Normalisierung
- Integritätsregeln
- Datenmanipulation: relationale Algebra, SQL

Relation

$$r(R) = \{t_1, \dots, t_m\}$$

m: Kardinalität

$$\text{mit } t_i = \langle v_1, \dots, v_n \rangle$$

n: Grad (Anz. Attribute)

$$v_j \in \text{dom}(A_j) \cup \{NULL\} \forall j$$

zu Relationenschema $R(A_1, \dots, A_n)$

In Domänen steht mehr Semantik als nur in Wertebereichen.

3 Kommunikation und Rechnernetze

Def. Rechnernetze

Kollektion miteinander **verbundener, autonomer** Rechner (Knoten, Hosts, Sites,...), die miteinander **Informationen austauschen** können.

Datenkommunikation

Information → Daten → Zeichen → Signale

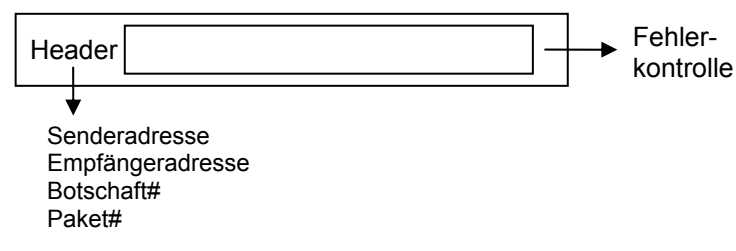
Kanäle, Verbindungen (Kanten)

- ✓ Übertragung: analog / digital
- ✓ Kapazität (Bandbreite): bps (Band)

Faktoren für die Übertragungszeiten

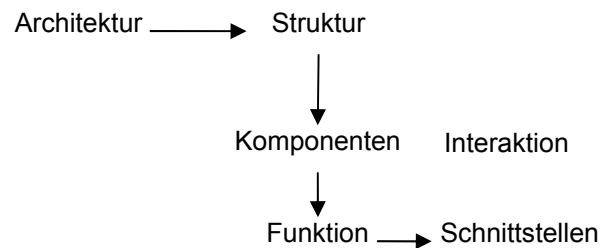
- **Kapazität**
- **Betriebsart**
- **Software** („overhead“)
 - Redundanz in Botschaften zur Fehlererkennung resp. -korrektur
 - Zusatzinformationen

Datenübertragungsrate (Nutzdaten) < Bandbreite des Kanals



Netzwerkarten
Kriterien

- **Verbindungsstruktur** (Topologie): Stern / Ring
- **Übertragungsmodus**: Punkt-zu-Punkt / Mehrpunkt
- **geograph. Verteilung** (LANs, MANs, WANs)



4 Transparenz

Def. Transparenz

Trennung „höherer“ semantischer Aspekte von „niederen“ Implementierungseigenheiten

Transparenz in Informatik =

„(hin)durchschauen“, nicht im Sinne von „hineinschauen“!!

Transparenzarten

- (1) **Datenunabhängigkeit**
- (2) **Netztransparenz** (Verteilungstransparenz)
- (3) **Replikationstransparenz**
- (4) **Fragmentierungstransparenz**

1 – Datenunabhängigkeit

logische & physische Datenunabhängigkeit
Nur möglich, da externe Sichten nicht von Änderungen in der DB-Struktur profitieren wollen.

2 – Netztransparenz

Aus Benutzersicht: als ob es ein zentrales System wäre
Sichtweisen: → Funktionen
→ Daten

Gegenbeispiel UNIX:

local copy: cp <source> <dest>

global copy: rcp <MName:source> <MName:dest>

Netztransparenz:

- ✓ **Ortstransparenz**
- ✓ **Namenstransparenz** (eindeutige Bezeichnung jedes DB-Obj.)

3 – Replikationstransparenz

Replikationstransparenz: Existenz von Replikation. Sagt jedoch nicht aus, wo platziert.

Lesen: von welcher physischen Kopie?

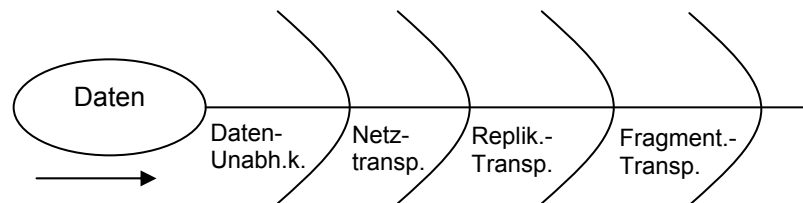
Ändern: an allen vorhandenen physischen Kopien

→ System selber kümmert sich darum. Benutzer soll nichts davon erfahren.

4 – Fragmentierungstransparenz

Gründe: Performance, Zuverlässigkeit, Verfügbarkeit

Zwiebel-Modell



5 DBMS-Standardisierung

Probleme Standardisierung

- zu frühe Standards
→ Hinderung des Fortschritts
- zu späte Standards
→ kein Hersteller hält sich daran, Eigenimplementationen

Referenzarchitektur

A conceptual framework whose purpose is to divide standardization work into manageable pieces and to show at a general level how these pieces are related to one another.

Standardisierung Ansätze

p. 76

a) **aufbauend auf (Software-)Komponenten**

[CCA / NBS]

Computer Corporation of America / National Bureau of Standards

b) **aufbauend auf Funktionen**

[ISO / OSI]

Schichtenarchitektur, Schnittstellen

bottom-up / top-down

c) **aufbauend auf Daten**

[ANSI / SPARC]

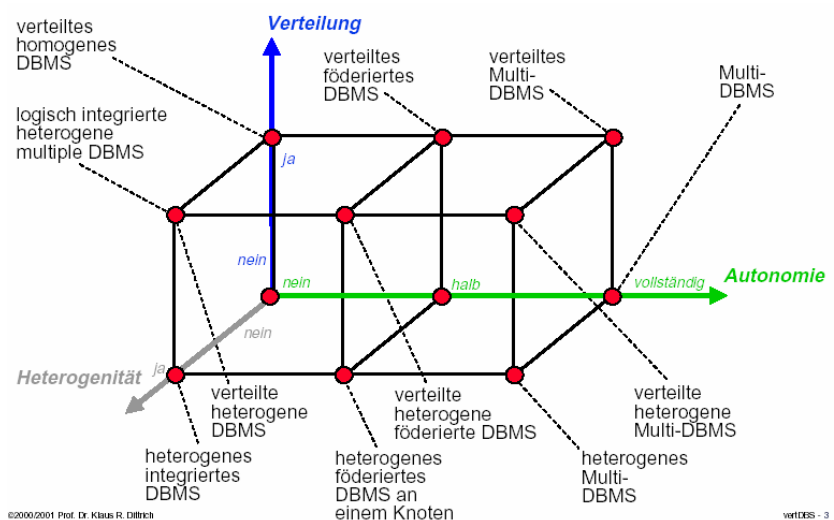
→ physische Speicherung → konzeptuell → externe Sicht

Alle 3 Ansätze werden für verschiedene Teilaspekte verwendet.

6 Architekturmodelle für vDBMS

Architekturmodelle

p. 82



©2000/2001 Prof. Dr. Klaus R. Dittich

vertDES - 3

- **Verteilung** → data
Daten physisch über mehrere Knoten verteilt
- **Heterogenität**
Hardware, z.B. Protokolle Netzwerkkommunik.
- **Autonomie** → control
Verteilung der Steuerung und Überwachung
Fähigkeit, unabhängig voneinander zu arbeiten.
Können Komponenten alleinstehend Transaktionen ausführen?
black boxes?
→ lokale Ausführung/Anfragen dürfen die globale Ausführung nicht beeinträchtigen
→ Systemkonsistenz-/Betriebsfähigkeit darf nicht beeinträchtigt werden bei Hinzufügen von ...

Autonomie-Arten

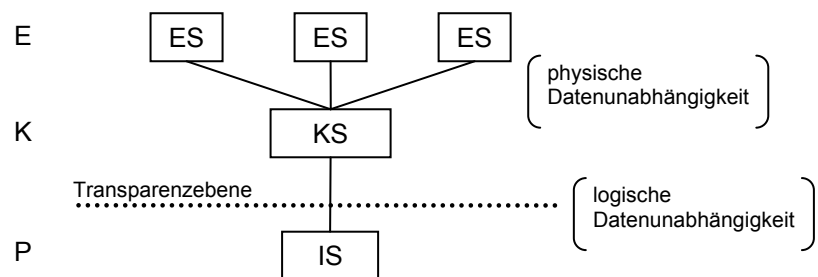
- ✓ **Entwurfsautonomie** (*design autonomy*)
- ✓ **Kommunikationsautonomie** (*communication autonomy*)
- ✓ **Ausführungsautonomie** (*execution autonomy*)

Architektur logisch integrierter vDBS

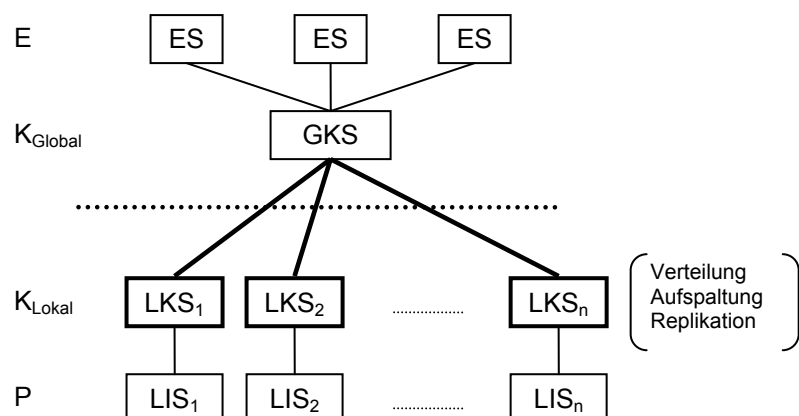
p. 90

herkömmlich (nicht verteilt):

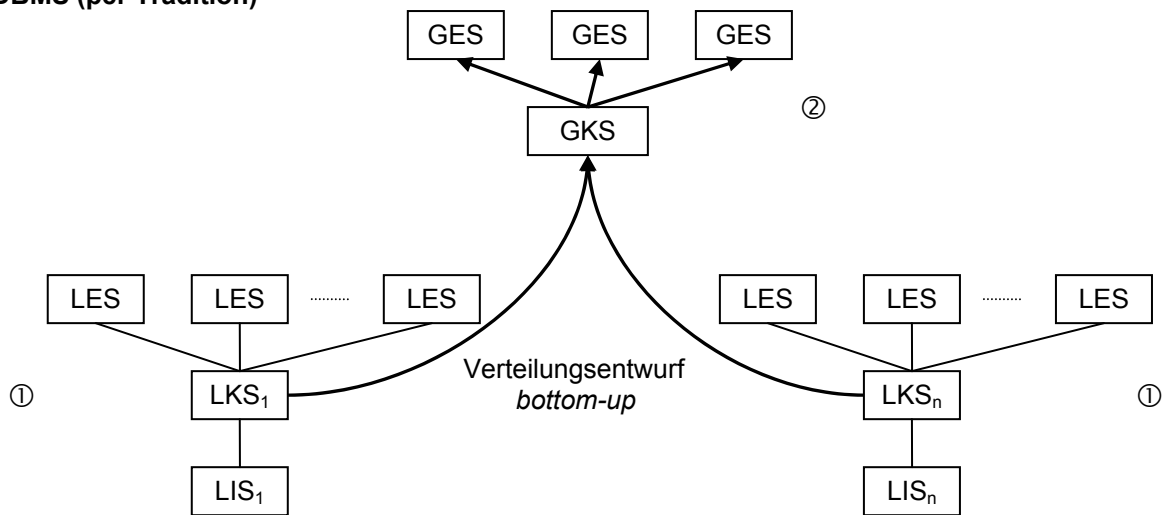
E – extern, P/I – physisch/intern, K - konzeptuell

**verteilt:**

L - logisch

ANSI/SPARC
Sicht

Multi-DBMS (per Tradition)

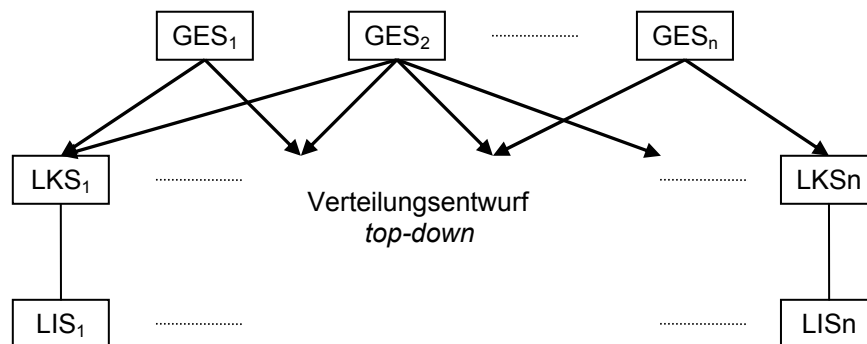


- 1) Schema in ein kanonisches übersetzen
- 2) mehrere Schemata zusammenfügen

Nachteile: • Hinzufügen einer Komponente, Abbildung des neuen LCS auf das GCS → schwierig

Kanonisches Datenmodell (globales):
 herkömmlich: Relational
 → Objektorientiert → heute: Objekt-Relational

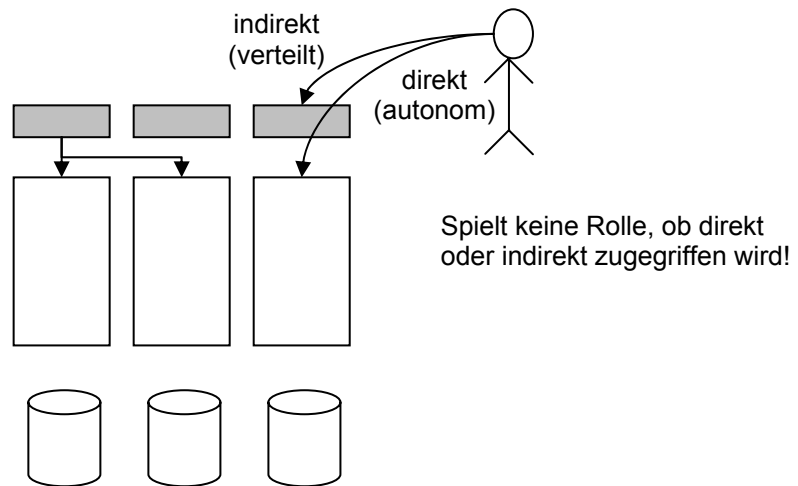
Multi-DBMS (per Konstruktion)



→ kein GCS!

Vorteile: • neues LCS_{n+1}
 einfach integrierbar, da nicht an GCS gekoppelt

Komponenten MDBMS



Übersicht Architekturmodelle:

	ANSI/SPARC	Komponenten Architektur
↪ Autonomie		
Autonomie		

globale Kataloge

→ nur bei Anwesenheit eines GKS

Enthält Meta-Daten über die Daten in der DB, information über Fragmente.
wichtig: effiziente Implementierung

wie bisher + **Fragmentierung + Allokation**

- **global / lokal ?**
- (physisch) **zentral / verteilt ?**
- **Replikation ja / nein ?**
 (an jedem Knoten alle Kataloginformationen)
 Vorteile: Verfügbarkeit,...
 Nachteile: Konsistenz (jedoch wird Katalog nicht oft ändern)

7 Distribution

Distribution der Daten

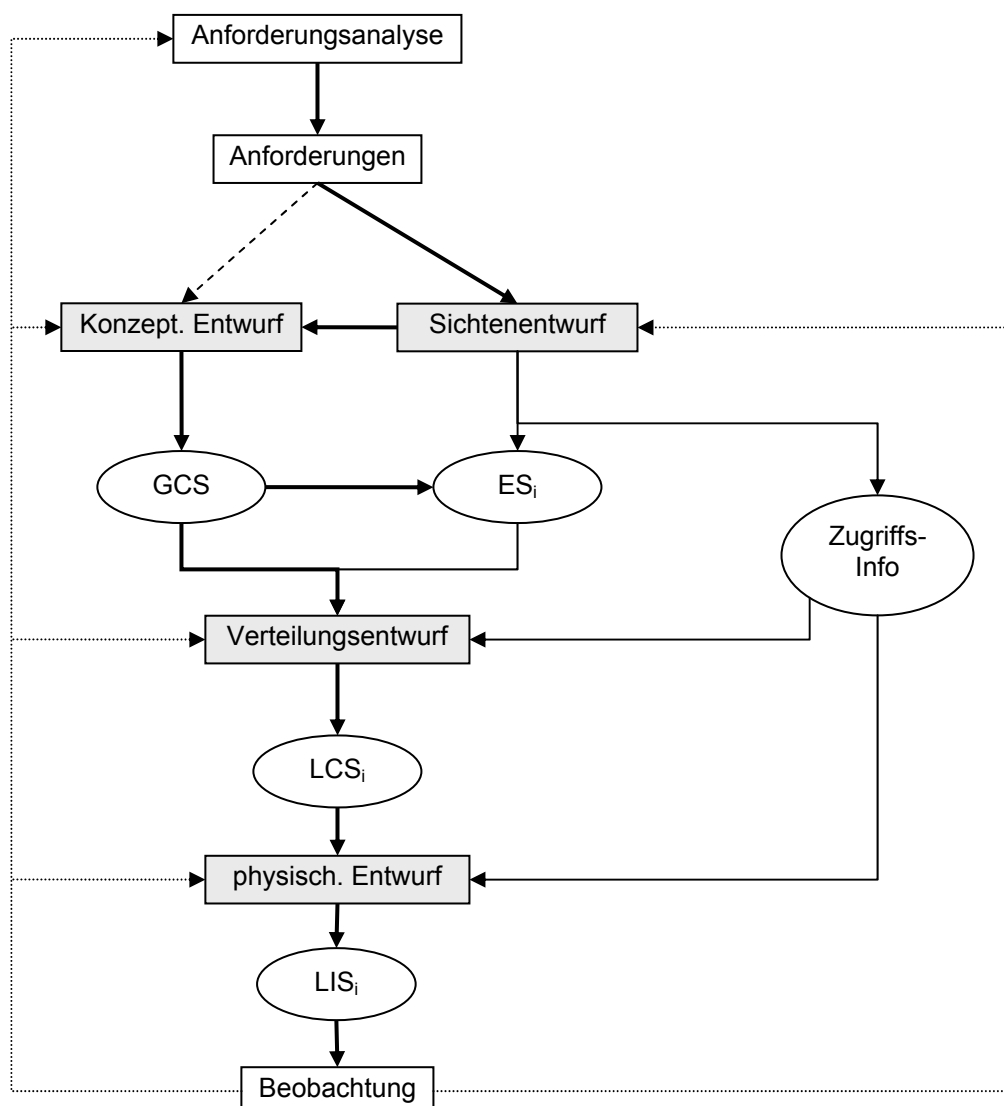
p. 102

- ✓ **Grad gemeinsamer Nutzung**
(no sharing / data sharing / data+program sharing)
- ✓ **Zugriffsmuster**
(static / dynamic)
- ✓ **Kenntnisgrad über Zugriffsmuster**
(no / partial / complete information)

Entwurfsstrategien:
top down / bottom up

top-down Entwurfsstrategie
(per Konstruktion, from scratch)

p. 105



(*) Zugriffsinfo:
 → Mengengerüste
 → Häufigkeiten Zugriffe
 → Arten der Zugriffe

8 Fragmentierung

Aufteilung von Relationen

p. 107

- Warum** überhaupt fragmentieren?
- Wie** fragmentieren?
- Wieviel** fragmentieren?
- Wie **Korrektheit** der Komposition prüfen?
- Wie **zuordnen**?
- Welche **Informationen** werden für Fragmentierung/Allokation benötigt?

a) Warum fragmentieren?

Fragmentation increases the **level of concurrency**

→ **Zuverlässigkeit**
→ **Effizienz**

b) Wie fragmentieren?

• **horizontal**

P(P#, PName, Budget, Ort)

→ P1 horizontal Budget ≤ 200'000

→ P2 horizontal Budget > 200'000

Prädikate müssen **disjunkt** und **exhaustiv** (alles umfassend) sein!

[→ *kleine Relationen*]

• **vertikal**

P(P#, PName, Budget, Ort)

→ P3 vertikal P3(P#, PName)

→ P4 vertikal P4(P#, Budget, Ort)

Projizieren, JOIN

Zwingende Redundanz: Primärschlüssel muss in allen Knoten sein
Korrektheitskriterium

[→ *schlankere Relationen*]

• **hybrid**

Kombination horizontal/vertikal

c) Wieviel fragmentieren

Extremfälle von Fragmentierung:

Horizontal+vertikal kombiniert

horizontal: nur 1 Tupel (unrealistisch!)

vertikal: Primärschlüssel + 1 Attribut

d) Korrektheit

p. 129

• **Vollständigkeit**

→ „alles da, was vorhin auch da war“
 $R \rightarrow R_1, \dots, R_n$ verlustfreie Zerlegung

• **Rekonstruktion(-smöglichkeit)**

$R \leftarrow R_1, \dots, R_n$
 Operator ∇ so: $R = \nabla R_i$

horizontal: \cup UNION
 vertikal: JOIN

Konsistenzbedingungen müssen beim Dekomprimieren erhalten bleiben.

• **Disjunktheit**

horizontal: disjunkt, keine doppelten Tupel, jedoch exhaustiv (alles umfassend)

vertikal: gleiche Attribute nicht mehrfach in Fragmenten (ausser PK)

e) Zuordnung

Wo Replikation?

→ mind. 1 Knoten für jedes Fragment

keine Replikation: partitioniert
 volle Replikation: Extremfall: alles an jedem Knoten
 partielle Replikation: nicht alles replik. an Teilmengen der Knoten

Replikation vorteilhaft, falls:

$$\frac{|RO - Anfragen|}{|Änderungen|} \geq 1$$

Einflüsse:

	volle Repl.	partielle Repl.	Partitionierung
Anfragebearbeitung	einfach	gleich schwierig	
Kataloge verwalten	einfach	gleich schwierig	
Konkurrenzsteuerung	mittel	schwierig	einfach
Zuverlässigkeit	sehr hoch	hoch	niedrig
Praxis	möglich	realistisch	möglich

f) benötigte Informationen

- **DB-Information**: Schema, Anz. Tupel in Relation
- **Anwendungsinformation**: SQL-Anfragen, Häufigkeiten/load
- **Netzwerkinformation** (quantitativ)
- **Rechnersysteminformation** (quantitativ)

→ Verteilungsproblem

gegeben: Schema

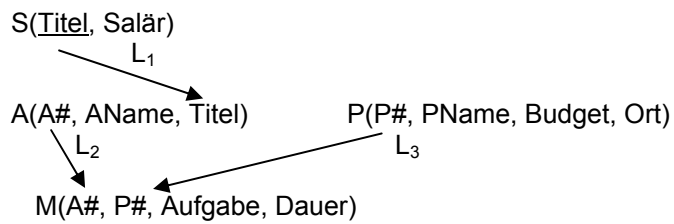
gesucht: Fragmente und Allokation

Fragmentierung

p. 112

- **horizontale**
primär → ∃ Algorithmen? (Schema, wichtigste Anfragen)
abgeleitet (aus anderen Relationen) → ∃ Algorithmen?

Schema:



L₁: 1:N
 owner(L₁) = S
 member(L₁) = A

Anwendungsinformationen:
 Prädikate in Anfragen? 80-20 Regel

- **vertikale**
 komplizierter als horizontale!
 → **Synthetisierende** Angehensweise (alles zerteilen, dann zus.fügen zu Fragmenten)
 → **Aufspaltende** Angehensweise (als Fragment betrachten)

Kriterien der Aufteilung: Affinität der Attribute,...

Bsp. vertikale Fragmentierung

p. 133

Q = {q₁, ..., q_n} Anfragen
 R(A₁, ..., A_n) Relation(Attribute)

$$use(q_i, A_j) = \begin{cases} 1 & \text{wenn } A_j \text{ in } q_i \text{ referiert wird} \\ 0 & \text{sonst} \end{cases}$$

$$aff(A_i, A_j) = \sum_{\substack{\text{alle } q \text{ mit } use(q, A_i) = 1 \\ \text{und } use(q, A_j) = 1}}$$

$$QA = \sum_{\text{alle Knoten}} \text{Ausführungshäufigkeit je } q * \frac{\text{Zugriffe } A_i, A_j}{\text{Ausführung von } q}$$

Bsp:
 R(X, Y, Z)

q₁: SELECT X FROM R where y=7
 q₂: SELECT Z FROM R WHERE Z>10

use(q₁, X) = 1 use(q₂, X) = 0
 use(q₁, Y) = 1 use(q₂, Y) = 0
 use(q₁, Z) = 0 use(q₂, Z) = 1

Affinitätsmatrix →

	X	Y	Z
X	--	2.5	0.7
Y		--	0.1
Z			--

9 Allokation

p. 147

Problem: $F = \{ F_1, \dots, F_n \}$ Fragmente
 $S = \{ S_1, \dots, S_m \}$ Netzknoten (sites), auf denen
 $Q = \{ q_1, \dots, q_k \}$ Anfragen laufen

gesucht: optimale Verteilung der F auf S bzgl. Q (Ressourcenallokation)

i) minimale Kosten:

- um jedes F_i auf s_j zu speichern (*store*)
- um F_i auf S_j zu verwenden (*query*)
- um F_i auf allen betroffenen S_j zu ändern (*update*)
- für Kommunikation

ii) maximale Leistung (Effizienz):

→ Leistungsmass: Antwortzeit, Durchsatz

Annahmen: Dateiallokation im Netzwerk (NP-vollständiges Problem !!)

Allokation: quantitativ

Fragmentierung: qualitativ

p. 152, *Totalkosten:*

$$TOC = \sum_{\text{alle Anfragen}} \text{Kosten Anfragebearbeitung} + \sum_{\text{alle Knoten}} \sum_{\text{alle Fragmente}} \text{Speicherkosten eines Fragments}$$

10 Bearbeitung verteilter Anfragen

verteilte Anfragen



- globale SQL-Anfrage → Algebra
- Feststellen der betroffenen Fragmente**
- Globale Anfrage** → { lokale Anfragen }
- Kommunikationsop.** hinzufügen, **optimieren**
- Anfrageausführung auf Fragmenten an Knoten
- Ergebnisse zusammensetzen**

[fett: im verteilte Fall neu dazugekommen]

p. 188, Fallbeispiel:

Anfrage

```
select AName
  from A, M
 where A.A# = M.A#
    and Aufgabe = "Manager"
```

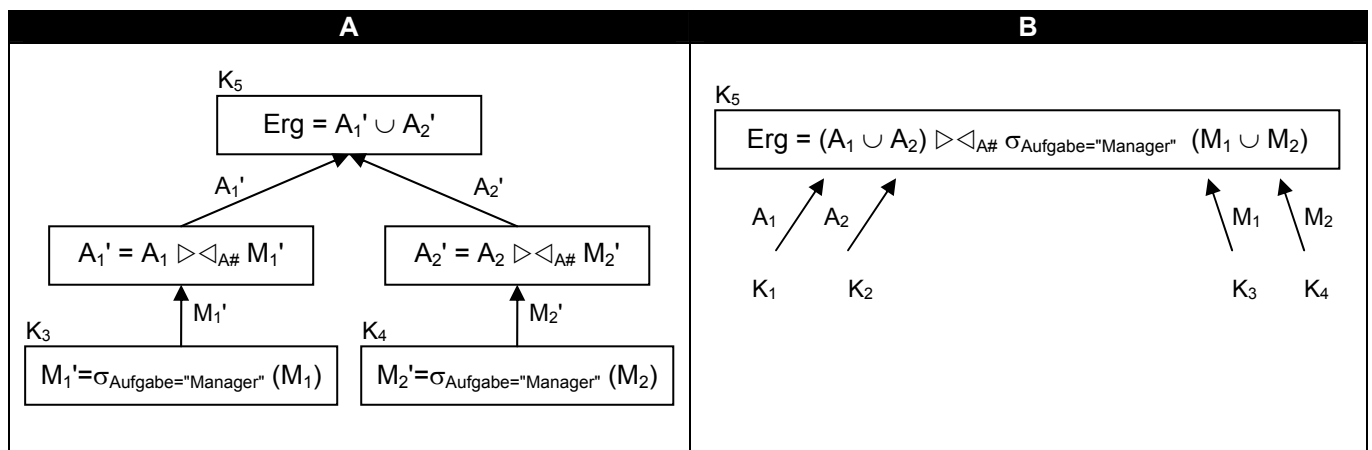
Algebra

- $\pi_{AName}(\sigma_{Aufgabe="Manager" \text{ and } A.A\#=M.A\#} (A \times M))$
- $\pi_{AName}(A \triangleright \triangleleft_{A\#} (\sigma_{Aufgabe="Manager"} (M))) \rightarrow \text{effizienter !!}$

$\triangleright \triangleleft$: Equi-JOIN

Knoten

K ₁	K ₂	K ₃	K ₄
A ₁ =	A ₂ =	M ₁ =	M ₂ =
$\sigma_{A\#\leq"A3"} (A)$	$\sigma_{A\#\gt"A3"} (A)$	$\sigma_{A\#\leq"A3"} (A)$	$\sigma_{A\#\gt"A3"} (A)$



gegeben:

size(A) = 400 [Tupel]
 size(M) = 1000 [Tupel]

tupacc = 1 [Einheit]
 tuptrans = 10 [Einheit]

tupel access: Kosten, auf ein Tupel zuzugreifen
 tupel transport: 1 Tupel von einem Knoten zum anderen transportieren

20 "Manager" in M
 Gleichverteilung der Daten auf die Knoten
 direkter Tupelzugriff (Zugriffspfad: INDEX) via A#(auf A) / Aufgabe (M)

1) M ₁ ', M ₂ ' produzieren	20*tupacc	20	1) A nach K ₅	400*tuptrans	4000
2) M ₁ ', M ₂ ' nach K ₃ , K ₄ (Datenkommunikation)	20*tuptrans	200	2) M nach K ₅	1000*tuptrans	10000
3) A ₁ ', A ₂ ' produzieren	2*10*tupacc*2	40	3) σ produzieren	1000*tupacc	1000
4) A ₁ ', A ₂ ' nach K ₅	20*tuptrans	200	4) $\triangleright \triangleleft$ produzieren	400*20*tupacc	8000
		460			23'000

Vorteil: direkter Tupelzugriff in den einzelnen Fragmenten. Geringe Transportkosten, da nur benötigte Tupel „nach oben“ gesendet werden.

Problem: Es fallen grosse Transportkosten an. Ausserdem sind durch UNION im „zentralen Knoten“ die direkten Zugriffspfade verloren gegangen, d.h. wir können die INDICES nicht mehr nutzen!

A ohne Zugriffspfade:		
1) M ₁ ', M ₂ ' produzieren	1000*tupacc	1000
2) M ₁ ', M ₂ ' nach K ₃ ,K ₄ (Datenkommunikation)	20*tuptrans	200
3) A ₁ ', A ₂ ' produzieren	200*10*tupacc*2	4000
4) A ₁ ', A ₂ ' nach K ₅	20*tuptrans	200
		5400
Fazit: A ist um Grössenordnungen schneller als B, da Transportkosten in der Realität sehr stark ins Gewicht fallen!		

Kostenminimierung, Masse

i) **Totalkosten = I/O-Kosten + CPU-Kosten + Komm.Kosten**

ii) **Antwortzeit** für Anfragen

Antwortzeit < Totalkosten, da die Anfrage parallel verarbeitet werden kann.

Komplexität von Algebraoperationen

p. 194

<u>select</u> σ	}	O(n)
<u>project</u> π (ohne Dupl.elimination)		
<u>project</u> π (mit Dupl.elimination)	}	O(n log n)
<u>group by</u>		
<u>join</u>	}	O(n log n)
<u>semijoin</u>		
Mengenoperationen (ausser X)		
X (Kreuzprodukt)	}	O(n ²)

Bsp. Semijoin

R \bowtie_a S

R	a	b
	1	x
	1	y
	2	z
	3	n

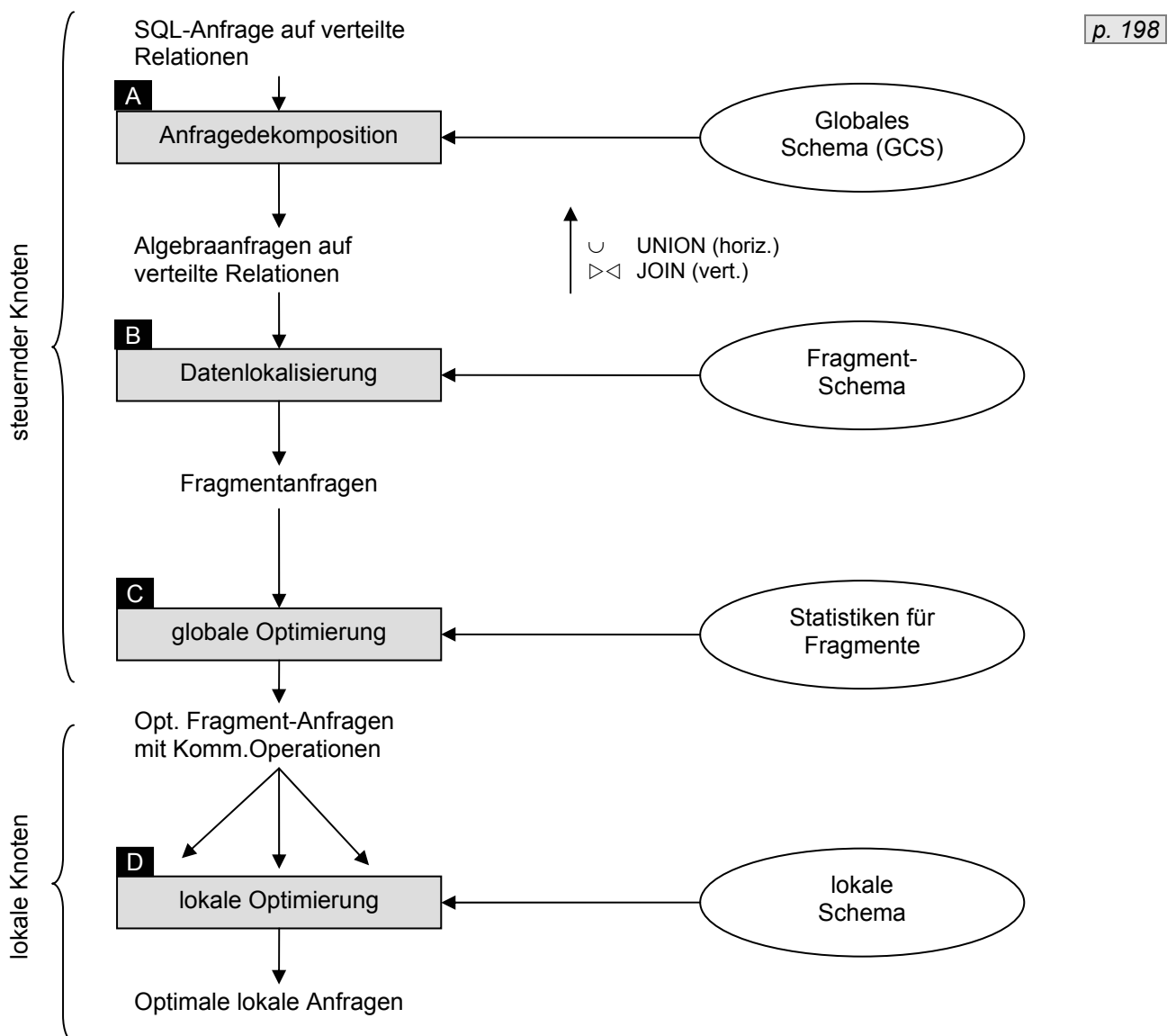
S	a	c
	1	t
	4	v
	5	j
	6	p

Semijoin nur um herauszufinden, welche Attribute beteiligt, ohne z.B. S.c anzuschauen

Charakteristika von
Anfrageprozessoren

p. 195

- **Optimierungstypen**
 - erschöpfende Suche im Lösungsraum (hohe Optimierungskosten!)
 - Zufallsstrategien (Kompromiss, tiefe Optimierungskosten)
 - Heuristiken (Bsp: JOINS durch Kombination von SEMIJOINS)
- **Optimierungsgranularität**
 - eine Anfrage
 - mehrere Anfragen
- **Optimierungszeitpunkt**
 - statisch (vor Ausführung)
 - hybrid
 - dynamisch (während der Ausführung)
- **Statistiken**



Erläuterung:

- Anfrageoptimierung, Redundanzen eliminieren, Vereinfachung
Abbruch bei Syntaxfehler, gesamte Dekomposition
① *normalize* → ② *analyze (semantically)* → ③ *simplify* → ④ *restructure* p. 204
- Restrukturierung intern, "Baumartige Darstellung"
- auf physisch Vorhandenes, unter Berücksichtigung TC = I/O-Kosten + CPU-Kosten + Komm.Kosten
- lokal wie normale SQL-Anfrage in zentralistischem System

Anfragedekomposition & Datenlokalisierung

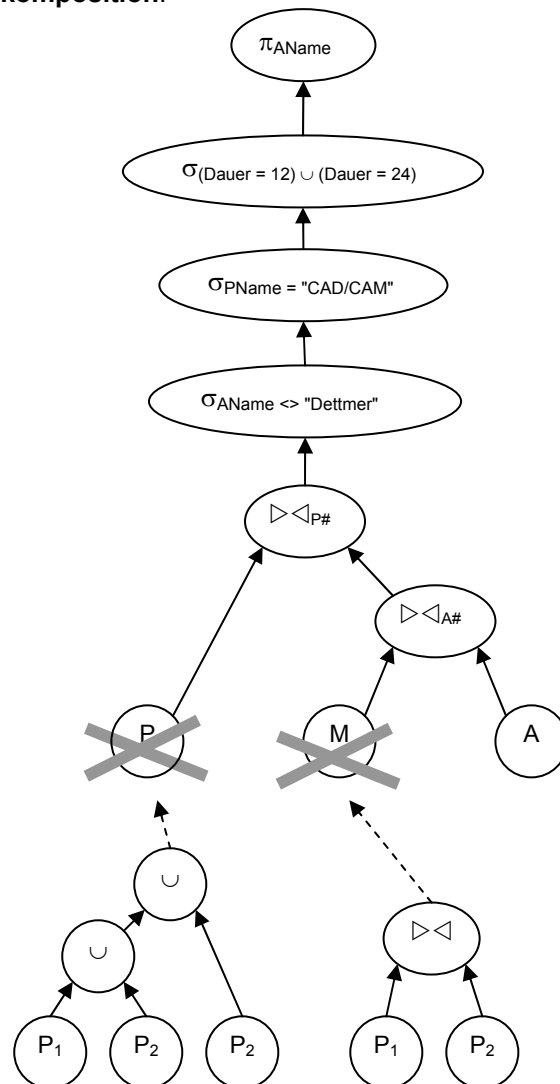
Baumstruktur

p. 211

```
SQL:  select  AName
      from    A, P, M
      where   A.A# = M.A#
            and M.P# = P.P#
            and AName <> "Dettmer"
            and P.PName = "CAD/CAM"
            and ( Dauer = 12 or Dauer = 24 )
```

Fragmentierung:P: P₁, P₂, P₃ horizontalM: M₁, M₂ vertikal

A: horizontal

A) Anfragedekomposition:**B) Datenlokalisierung:**

Umbauen des Baumes, Ersetzen von Blättern, Ausscheiden der unnötigen Fragmente

C) globale Optimierung:

Wissen + Annahmen über die Ausführungskosten

Ausführungskosten vs.
Antwortzeit**• Ausführungskosten:** $C_{CPU} * \#Befehle + C_{IO} * \#IO\text{-Befehle} + C_{MSG} * \#Botschaften + C_{TM} * \#bytes$ **• Antwortzeit:**

CPU-Zeit + IO-Zeit + Komm.Zeit

CPU-Zeit = Befehlsausführungszeit * #sequentieller Befehle

11 Transaktionen

Transaktionen

p. 274

- Paralleles R/W
- Fehlersituationen

↑ Konsistenz, Integrität, Zuverlässigkeit

- **DB-Konsistenz** (einzelne TA)
explizit: Schlüssel, Typenhaltung
implizit
- **TA-Konsistenz** (gesamte TA-Menge)
Replikate: wechselseitige Konsistenz
- Zuverlässigkeit

} TA-
Verwaltung

TA heute "selbstverständlich" – Nicht eine Spezialität Relat. DBS, jedoch heute sogar im SQL-Standard.

Einzelne Befehle, z.B. UPDATE, werden intern als TA betrachtet.

Transaktion Befehle

→ **begin_TA**
→ **end_TA** (commit_TA)
→ **abort_TA**

im SQL: begin_TA implizit

Transaktion (*high-level*) vs. Lock/Unlock (*low-level*)

ACID

p. 283

A atomicity
C consistency
I isolation
D durability

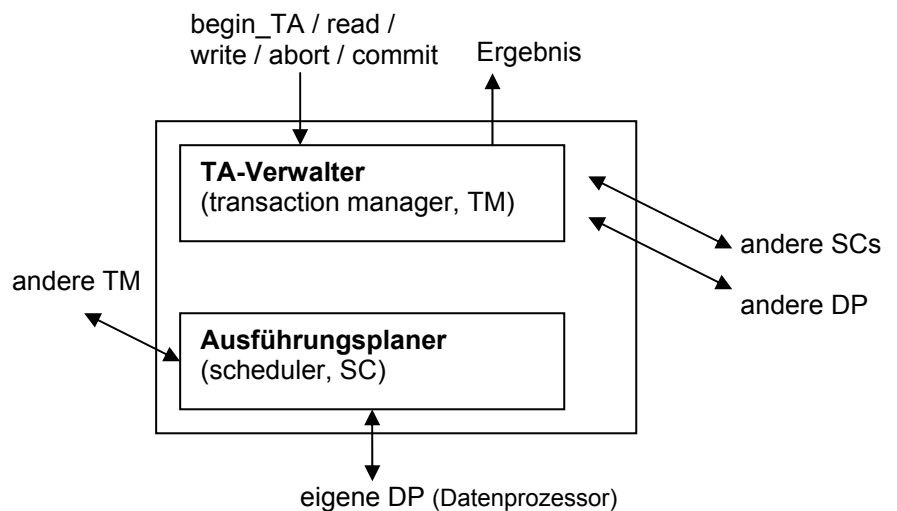
Konsistenz-Ebenen

p. 284

degree 3	(echte ACID) 1) TA überschreibt keine schmutzigen Daten (noch kein commit) anderer Transaktionen 2) TA gibt keine Änderungen frei, bevor alle Änderungen freigegeben wurden (commit) 3) TA liest keine schmutzigen Daten anderer TA's 4) Andere TA's verschmutzen keine von TA gelesenen Daten bevor TA endet
degree 2	1), 2), 3) 4) nicht gewährleistet Falls eh nur 1mal gelesen wird und man weiss, dass man nicht ein weiteres Mal liest.
degree 1	1), 2) 3), 4) nicht gewährleistet
degree 0	1) 2), 3), 4) nicht gewährleistet

Architektur

p. 295



Die TA wird vom TA-Verwalter des jeweiligen Ursprungsknoten koordiniert.

verteilte Konkurrenzsteuerung

- ✓ einfacher Fall: **sequentielle Abarbeitung**
- ✓ **Schichtenarchitektur**: 1 Schicht = 1 Problem
- ✓ **Serialisierbarkeit**, Ausführungspläne nicht seriell, aber Ergebnis äquivalent zu serieller Ausführung

partitionierte verteilte DB

lokale AP, serialisierbar → globale AP, ebenfalls serialisierbar
sonst (Replikation): gilt nicht!

Replikation
(RKP)

Bsp: x an 2 Knoten repliziert

T ₁ :	r(X) X := X+5 w(X) c	T ₂ :	r(X) X := X*10 w(X) c
lokal:	K ₁ : T ₁ , T ₂ K ₂ : T ₂ , T ₁	X = 1 → X = 60 X = 1 → X = 15	

→ **Replikationsprotokoll (RKP)** benötigt!

RKP: read(X) → read(X_i) für alle i
write(X) → write(X_i) für alle i

ROWA – read once write all

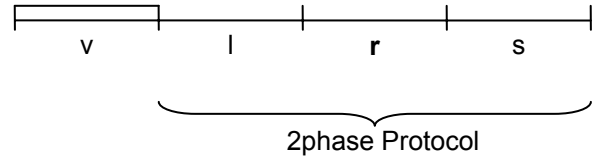
→ map a read on a logical data item to only one copy of the data item, but map a write on a logical data item to a set of writes on all physical data item copies

Konkurrenzsteuerungsverfahren

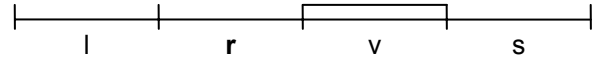
p. 332

- [v] validieren
- [l] lesen
- [r] rechnen
- [s] speichern

pessimistisch

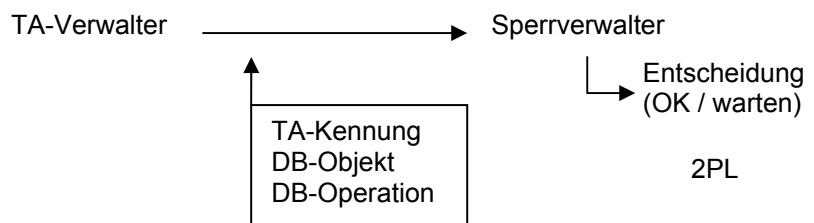


optimistisch



Sperrverfahren
locking

(Ausführungsplaner wird zu Sperrverwalter)

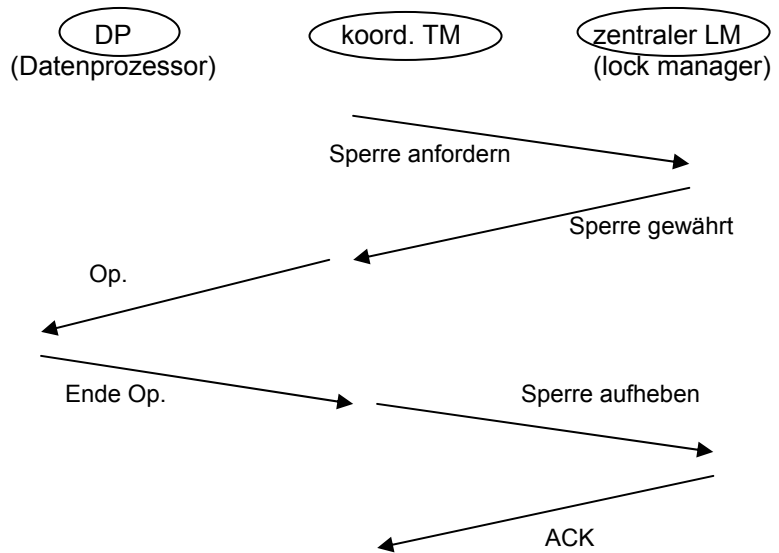


2PL: 2-phase-locking

→ strikt: alle Freigaben am Ende (Vermeiden von kaskadierenden aborts)

p. 318

zentralisiertes 2PL



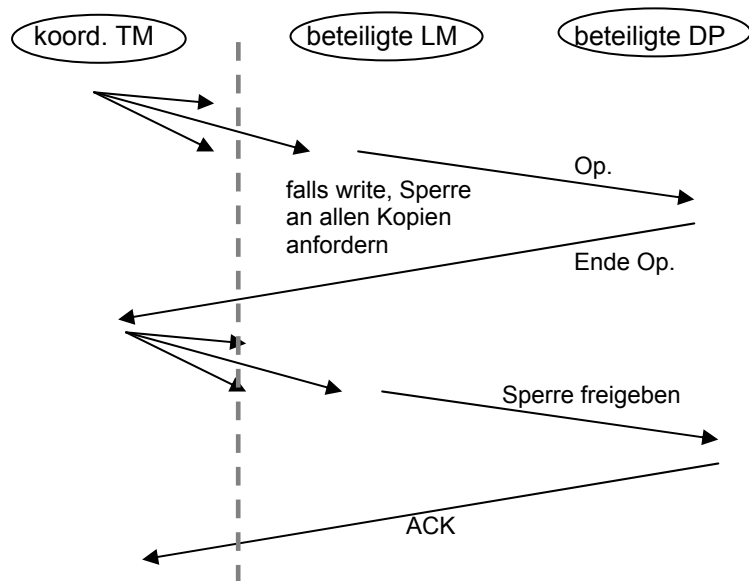
Problem: Zentraler LM als „Flaschenhals“ (Ausfall!!!)

Primärkopien-2PL PCPL

Für jedes DatenObj. wird ein Replikat zur Primärkopie erklärt.
 Dort muss die Sperre angefordert werden.
 → Flaschenhals beseitigt
 → nur 1 Kopie muss gesperrt werden.

verteiltes 2PL

Lock-Manager an allen Knoten. Für alle Replikate Sperren (nicht nur 1 Sperre für alle Replikate)



Zeitstempelverfahren

p. 324

Serialisierreihenfolge (nicht Ausschlussverfahren)

Jede TA erhält bei Start eindeutigen Zeitstempel $ts(T_i)$
 Zeitstempel je TM monoton steigend (nie 2 TA gleicher TS!!)

TO-Regel (time order)

$O_{i,j}$, $O_{k,l}$ zwei in Konflikt stehende Operationen der TA T_i und T_k
 $O_{i,j}$ wird vor $O_{k,l}$ ausgeführt, genau dann, wenn $ts(T_i) < ts(T_k)$

Alternative:

read-ts: jüngste TA, die gelesen hat

write-ts: jüngste TA, die geschrieben hat

Verklemmungen
(dead locks)

p. 337

globale vs. lokale Verklemmungen

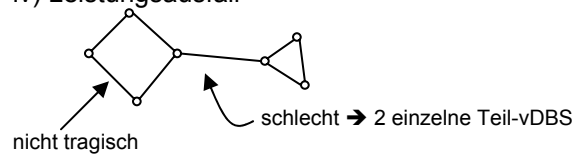
Im verteilten Fall kann lokal OK sein, jedoch globaler Deadlock!

- ✓ **Deadlock Prevention (verhindern → statisch)**
alle Locks am Anfang, restriktiv, nicht brauchbar für TA's
- ✓ **Deadlock Avoidance (vermeiden → dynamisch)**
Konkurrenzsteuerungs-Algorithmus (Scheduler) passt auf Deadlocks auf.
Betriebsmittel ordnen

Konsistenz bei vDBS Ausfalltypen

p. 363

- a) TA-Ausfall**
TA bricht ab oder wird abgebrochen (ca. 3% der TA's)
- b) Knotenausfall (Systemausfall)**
zentral: nichts geht mehr / verteilt: 1 Knoten von mehreren HW od. SW funktioniert nicht mehr. Problem: Was im Hauptspeicher/Puffer ist, ist verloren. Daten im Hintergrundspeicher sind noch vorhanden. Knoten ist über Netz nicht mehr erreichbar
Ausfall: partiell (1) / paar / alle Knoten
- c) Speicherausfall (HG-Speicher)**
entweder Probleme mit Betriebssystem od. Festplatte selbst (kann auch 1-2x Jahr passieren) → Duplexkopien, Archivkopien
Meist nur an 1 Knoten ein Plattenausfall
Lösung: lokale Recovery-Mechanismen. Können gleich behandelt werden wie bei einem zentralen System
- d) Kommunikationsausfall**
- Fehler in Botschaften
 - falsch geordnete Botschaften
 - verlorene Botschaften als Folge von iv) oder Knotenausfall dazw.
 - Leistungsausfall



Replikate können helfen. Müssen bei erneutem Verbindungsaufbau nachgeführt werden. Einziges Problem: Leute arbeiten in der Zwischenzeit evtl. auf veralteten Daten, kann man aber akzeptieren → Replikationen helfen, verkomplizieren aber auch

Recovery Vorgehen

- ✓ An jedem Knoten sollten **Wiederanlaufmechanismen für den lokalen Fall** (vergl. Zentrale DB) existieren
- ✓ **Recovery-Verwaltung** für vernünftige Koordination des Wiederanlaufs
- ✓ Knoten, von dem wir ausgehen: **Koordinator** für diese TA. Jeder Knoten kann mal Koordinator sein. Pro TA gibt es genau 1 Koordinator!
- ✓ **einfachster Fall:**
→ Koordinator kann alles lokal machen
→ 1 Koordinator, 1 Beteiligter Knoten
- ✓ Das Ganze kann **parallel** oder **seriell** laufen

Recovery
Protokolle

pro TA wird benötigt:

- **Commit-Protokoll**

wird von aussen angestossen über "commit" oder "abort", dann geht's los (zeitlich) → am wichtigsten! (durch Koordinator)

- **Terminierungs-Protokoll**

Wenn ein Knoten ausfällt während TA läuft → Was sollen die anderen Knoten machen?

Wird als Teil des Commit-Protokolls geführt

- **Wiederanlauf-Protokoll**

Wie reagiert der ausgefallene Knoten, wenn er wieder „zum Leben erwacht“ → Wird über Befehl „recover“ gestartet, der vom Betriebssystem kein Wiederstart kommt. Muss zuerst ausgeführt werden.

2PC

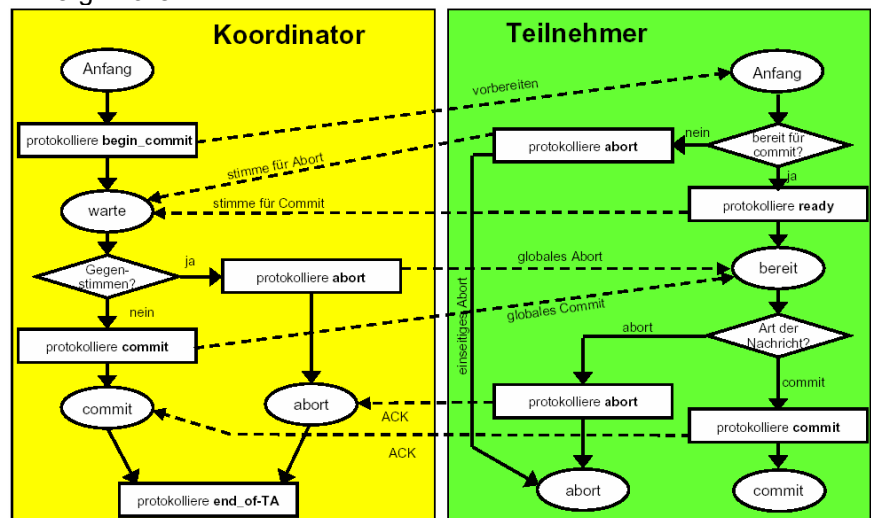
(2-phase-commit)

[bisher: 2PL → 2-phase-locking]

2PC: 2-phase-commit

commit von aussen geht an Koordinator. Dieser stimmt ab: geht's/geht's nicht (über ganzes Netz, nicht nur lokal)

→ vergl. Folie 7

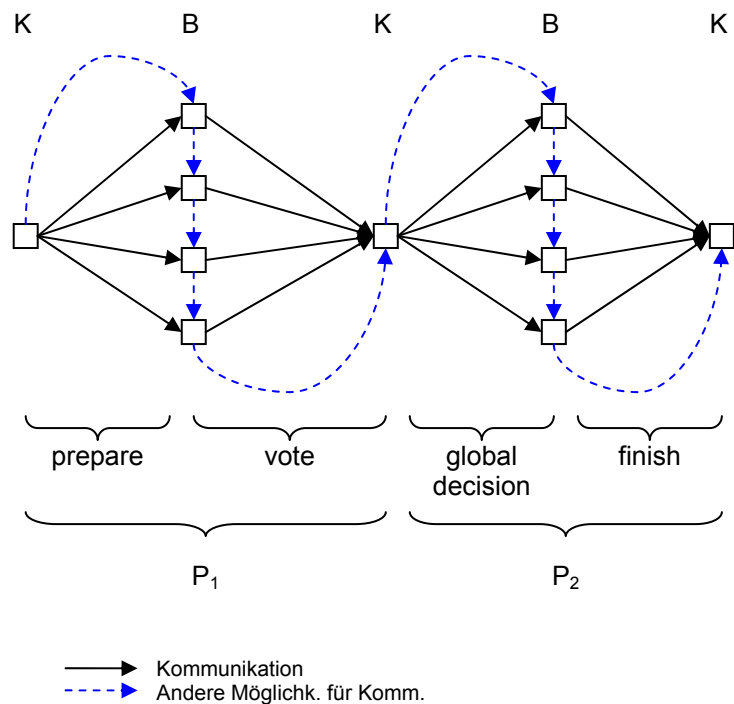


3PC: 3-phase-commit

→ nonblocking protocol

2PC

2PC alternative Aufzeichnung (zeitlicher Ablauf):
 [K] Koordinator, [B] Beteiligte, [P] phase



Problem: Wenn Knoten nicht antwortet → andere Knoten warten
 → Terminierungsprotokolle sollten nicht blockiert sein! (die anderen sollen nicht warten müssen)
 → Wiederanlaufprotokolle sollten unabhängig sein (ohne Kommunikation)

Diese Protokolle gibt es für Ausfall von genau 1 Knoten. Wenn mehrere Knoten: viel komplizierter → 2PC genügt nicht mehr!

Möglichkeit: Benutzer vergisst commit zu verschicken → Timeouts bevor überhaupt 2PC läuft (bei Koordinator)

12 Weitere Entwicklungen

- ✓ **SDD1**: war früher ein System des US-Militärs
 → hat TA voranalysiert (ist es read/write/...)
 → Zeitstempelverfahren in SDD1 erfunden
- ✓ **Replikationsserver**
 repliziert gezielt Daten, ist aber keine vDB. Rest von vDB ist nicht vorhanden
- ✓ **Objektorientierte DB**: sind von Konstruktion her verteilt
- ✓ **www ≠ vDB**
 ist nur ein verteiltes System. Heterogenität sehr gross in www → Versuch das Ganze in eine vDB zu bringen: Semantic Web