9. Replikationsverfahren

9.1 Motivation

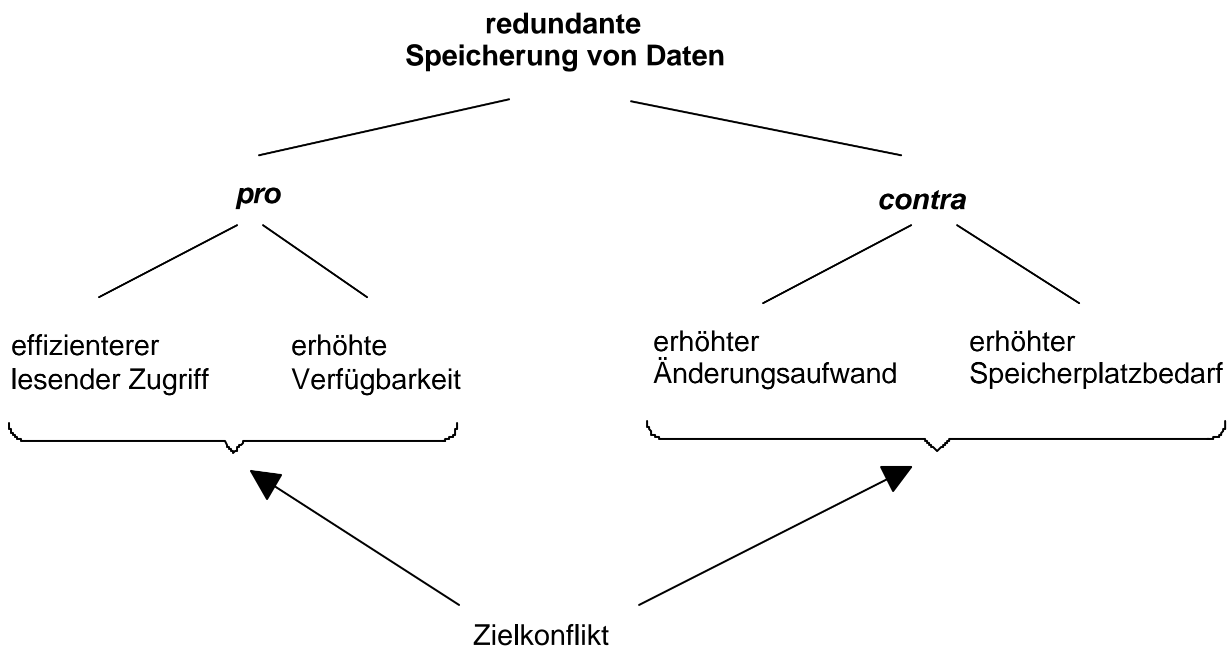


Abb. 9-1: Zielkonflikte bei redundant gespeicherten Daten

- **Ziel/Forderung:**
  - Aktualisierung durch das vDBMS, transparent für AP!
  - Gleiches Verhalten wie nicht-replizierte DB

    ⇒ ***1-Kopie-Äquivalenz***

    dazu notwendig: wechselseitige Konsistenz der Kopien

- Realisierung der Ziele/Forderungen durch ein ***Replikationsverfahren***

9.2 Grundsätzliche Problemstellungen und Vorgehensweisen

9.2.1 Read-One-Write-All-Verfahren (ROWA-Verfahren)

❐ **Merkmale:**

- Eine logische Leseoperation wird jeweils auf eine physische Leseoperation auf einer beliebigen Kopie abgebildet ("read one")
- Eine logische Schreiboperation führt zu physischen Schreiboperationen auf allen Kopien ("write all"; synchrones Update aller Kopien)

❐ **Bewertung:**

⊕ Einfach zu implementieren

⊕ Alle Kopien stets auf dem gleichen Stand

⊕ Problemloses lokales Lesen

⊖ Starke Abhängigkeit einer Änderungs-TA von der Verfügbarkeit *aller* kopienhaltenden lokalen DBMSe (✱)

⊖ Hierdurch relativ geringe Verfügbarkeit des Gesamtsystems

⊖ Längere Laufzeiten für Änderungs-TAs

❐ Anmerkung:

- Naheliegendes Verfahren, wegen (✱) aber nur von geringem praktischen Wert

- Deshalb: Viele weitere Vorschläge für Replikationsverfahren

Zu unterscheidende Aspekte bei der Diskussion von Replikationsverfahren

1. **Kopien-Update-Strategie**:
   Vorgehensweise im Normalfall, d. h. Anzahl und Auswahl der Kopien, die zur Durchführung eines Updates benötigt werden

2. **Fehlerbehandlung:**
   Vorgehensweise im Fehlerfall, insbesondere bei Netzpartitionierungen

3. **Synchronisation konkurrierender Zugriffe:**
   Eingesetztes Synchronisationsverfahren und Gewährleistung der globalen Serialisierbarkeit von Transaktionen

4. **Behandlung von Lesetransaktionen**

9.2.2 Kopien-Update-Strategien

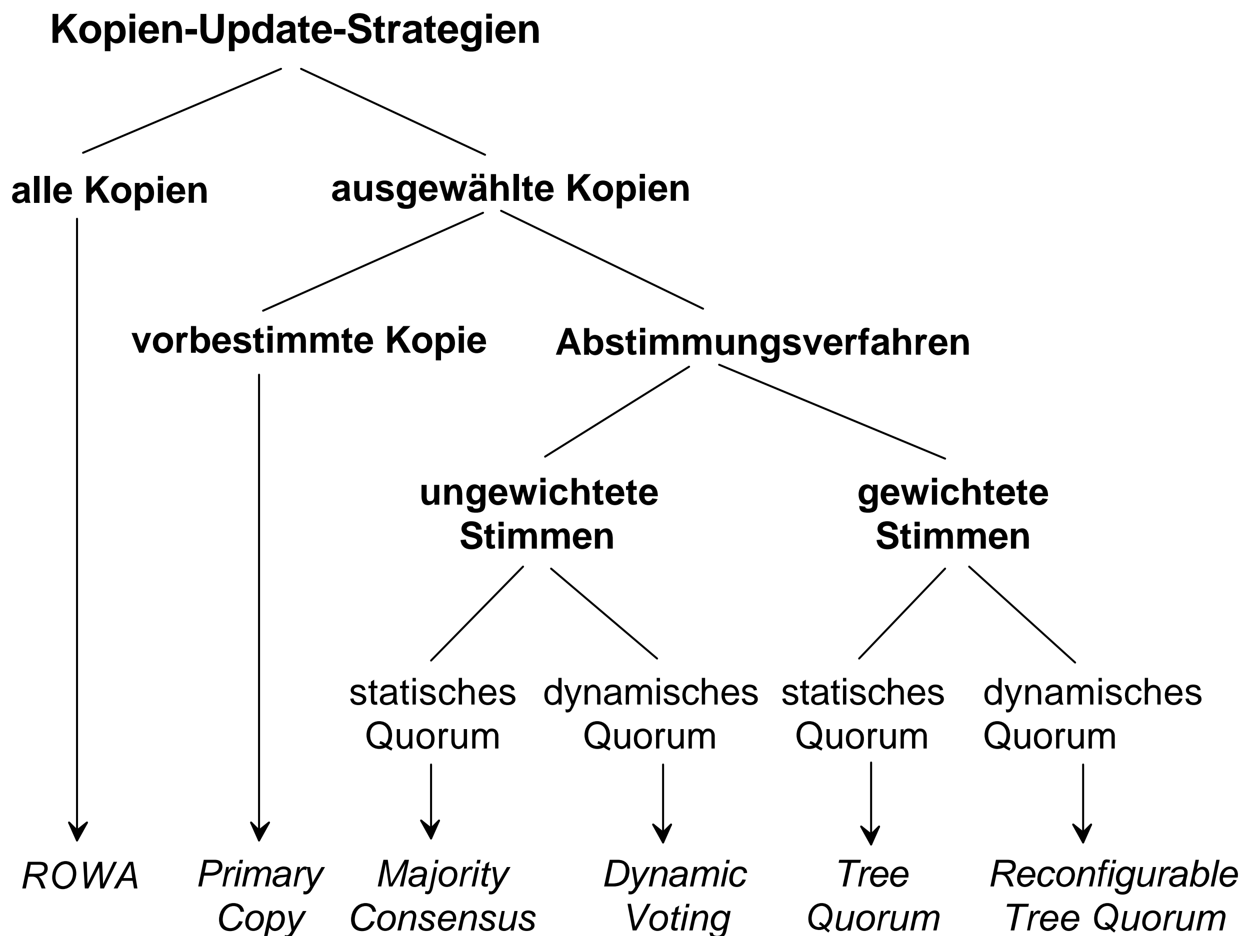


Abb. 9-2: Kopien-Update-Strategien im Überblick

❒ Verfahren mit vorbestimmter Kopie

- Festlegung einer Primärkopie (primary copy, master copy), über die alle Updates laufen

- Diese ist die "Originalversion", alle anderen sind von dieser abgeleitete Kopien

- Zum Update ist berechtigt, wer die Primärkopie sperren kann.

- Zweistufiger Ansatz (in der reinen Form):
  - Transaktion sperrt und ändert nur die Primärkopie
  - Primärkopie sorgt für Propagation der Änderungen zu den anderen Kopien

- Bekanntestes Verfahren in der Literatur:

  Primary Copy Verfahren [1] (siehe später)

- Varianten dieses Verfahren in vielen kommerziellen Produkten implementiert.

---

[1] Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans. on Software Engineering, May 1979, SE-5(3):188-194

❒ Abstimmungsverfahren

- Prinzip:
  Ein Update auf einer Kopie wird nur dann durchgeführt, wenn die entsprechende Transaktion in der Lage ist, eine Mehrheit von Kopien dafür zu gewinnen (z.B. geeignet zu sperren)

- Die Verfahren unterscheiden sich darin, ob
  - alle Kopien bzgl. dieser Abstimmung
    - gleich behandelt werden (= *ungewichtete Stimmen*)
    - oder nicht (= *gewichtete Stimmen*)
  - Anzahl von Stimmen, die für das Erreichen der Mehrheit erforderlich sind
    - fest vorgegeben ist (= *statisches Quorum*) oder
    - sich erst dynamisch zur Laufzeit ergibt
      (= *dynamisches Quorum*)

  Für Lesezugriffe (***Lesequorum***; Q_L) und für Schreibzugriffe (***Schreibquorum***; Q_U) können unterschiedliche Anzahl von erforderliche Stimmen festgelegt werden.

- **Definition 9-1: *Quorum-Überschneidungsregel***

> Sei Q die Gesamtstimmenzahl, dann muß stets die folgende ***Quorum-Überschneidungsregel*** beachtet werden, um die 1-Kopie-Serialisierbarkeit zu gewährleisten:
>
> 1. $Q_L + Q_U > Q$
> 2. $Q_U + Q_U > Q$

❍ Ungewichtete vs. gewichtete Stimmen

- **ungewichtet:**
  - jede Kopie zählt einfach
  - Anfrage an die verschiedenen Kopien in beliebiger Reihenfolge

- **gewichtet:**
  - Ziel: Reduktion des Kommunikationsaufwandes
  - Kopien erhalten (in gewisser Weise) unterschiedliche Stimmengewichte
  - Abfrage der Kopien in festgelegter Reihenfolge (beginnend mit den "hochwertigsten" Kopien
  - ist eine "höherwertige" Kopie nicht erreichbar, so kann ihr Stimmengewicht durch eine entsprechende Anzahl "niederwertigerer" Kopien kompensiert werden

❍ Statisches vs. dynamisches Quorum

- **statisch:**
  - Quoren werden (z.B. bei Systemstart) festgelegt
  - Problem: bei zahlreichen Ausfällen kann erforderliches Quorum u.U. nicht mehr erreicht werden

- **dynamisch**:
  - Quoren werden entsprechend der Verfügbarkeit von Knotenrechnern immer wieder neu angepaßt

9.2.3 Strategien für den Fehlerfall

- Zu unterscheiden
  - Ausfall einzelner Knoten
  - Netzpartitionierung (Zerfall in Teilnetze)

- **Ausfall einzelner Knoten**
  - **vorbestimmte Kopie**
    - das Teilnetz, das im Besitz dieser Kopie ist, darf noch Updates machen
    - die anderen dürfen (maximal) noch Lesezugriffe auf ihre Kopien erlauben
  - **Abstimmungsverfahren**
    - keine spezielle Behandlung dieses Fehlerfalles
    - erreicht eine Transaktion das erforderliche Quorum, dann darf sie die entsprechende Operation (Lesen oder Schreiben) durchführen

❐ Netzpartitionierung

❍ Zwei Gruppen von Verfahren

- konservative ("vorsichtige", "pessimistische") Verfahren
- progressive ("optimistische") Verfahren

❍ **Konservative Verfahren**

- Erhaltung der DB-Konsistenz (zu jedem Zeitpunkt) hat hier oberste Priorität
- im Partitionierungsfall darf maximal ein Teilnetz ("Hauptpartition") noch Updates durchführen
- Bestimmung der Hauptpartition auf verschiedene Weise möglich
  - Primärkopie
  - Abstimmungsverfahren

❍ **Progressive Verfahren**

- Der jederzeitigen Verfügbarkeit des Systems wird höchste Priorität eingeräumt
- Updates werden in allen Netz-Paritionen zugelassen
- temporäre Inkonsistenzen werden toleriert bzw. müssen hingenommen werden
- (verschiedene) "Konsoldierungsstrategien" bei "Wiedervereinigung" der Teilnetze
- Interessanter Vertreter dieser Kategorie:

  Data Patches [2] (siehe später)

---

[2] Garcia-Molina, H.: Data-patch: Integrating Inconsistent Copies of a Database After a Partition. Proc. 3th IEEE Symposium on Reliable Distributed Systems, New York, October 1983, pp. 38-48

9.2.4 Synchronisation von Updatetransaktionen

❒ Ziel:

Das vDBS soll sich bzgl. Korrektheit nicht anders verhalten als ein vDBS ohne Kopien.

❒ **Definition 9-2**: ***1-Kopie-Serialisierbarkeit***

> Eine Schedule S von (abgeschlossenen) Transaktionen, die auf einer repliziert gespeicherten verteilten Datenbank ausgeführt wurden, heißt dann und nur dann *1-Kopie-serialisierbar*, wenn es mindestens eine serielle Ausführung der Transaktionen aus S auf einer verteilten Datenbank ohne Replikate gibt, welche, angewandt auf denselben Ausgangszustand, die gleiche Ausgabe sowie denselben Endzustand erzeugt.

❒ Verschiedene Verfahren zur Sychronisation der Updates

- Konventionelle Verfahren
  - Sperrverfahren
  - optimistische Verfahren
- Zeitstempelbasierte Synchronisationsverfahren
- Semantische Synchronisationsverfahren

9.2.5 Behandlung von Lesetransaktionen

- Im Prinzip drei Fälle/Vorgehensweisen bzgl. des Inputs der Lesetransaktion unterscheidbar

  - der Input muß stets aktuell und konsistent sein

  - der Input muß konsistent sein, darf aber "etwas" veraltet sein

  - "etwas inkonsistenter" Input ist tolerabel

❐ **Fall 1: Der Input muß stets aktuell und konsistent sein**

⇒ gleiche Konsistenzanforderungen wie an Update-Transaktionen

⇒ bis auf unterschiedliche Sperrmodi / unterschiedliche Quoren keine Unterscheidung zwischen Lese- und Update-Transaktionen

⇒ geringer Systemdurchsatz

❐ **Fall 2: Der Input muß konsistent sein, darf aber "etwas" veraltet sein**

⇒ ***Versionskonzept*e**
Lesetransaktionen werden auf alte bzw. eine der alten Versionen umgeleitet

⇒ einfachster Fall: (einfaches) ***Snapshot-Konzept***
(in vielen kommerziellen Replikationsprodukten realisiert)[3]

⇒ Lesetransaktionen können ohne oder mit geringem Synchronsiations-Overhead ausgeführt werden[4]

---

[3] Allerdings nicht ausschließlich. ROWA bzw. Varianten davon kann meist auch gewählt werden.

[4] siehe hierzu z.B. Mohan, C.; Pirahesh, H.; Lorie, R.: Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. Proc. ACM-SIGMOD '92, Int'l Conf. on Management of Data, San Diego, Calif., June 1992, pp. 124-133

❐ <u>Fall 3</u>: "Etwas inkonsistenter" Input ist tolerabel

⇒ Kommt ohne Versionen aus

⇒ Angabe einer ***maximalen Differenz*** ε, um die der Input maximal vom aktuellen konsistenten Datenbankzustand abweichen darf

⇒ eine Lesetransaktion gilt dann noch (in diesem Sinne) gegenüber Updatetransaktionen als serialisierbar, wenn ihr Input innerhalb der vorgegebenen ε-Abweichung bleibt (⇒ **ε-Serialisierbarkeit**) [5]

Für Wahl der konkreten ε-Abweichung gibt es verschiedene Möglichkeiten:

- ❍ Anzahl der noch nicht durchgeführten Updates ("missed updates")
- ❍ maximale zeitliche Distanz zum letzten Update (Beispiel: Börsenkurse)

<u>Aber</u>: Da kein konsistenter Input garantiert werden kann, nur für spezielle Anwendungen einsetzbar.

---

[5] siehe hierzu Pu, C.; Leff, A.: Replica Control in Distributed Systems: An Asynchronous Approach. Proc. ACM SIGMOD ‘91, Int’l Conf. on Management of Data, Denver, May 1991, pp. 377-386

9.3 Ausgewählte Verfahren

9.1.1 Primary Copy [6]

- Idee: Zweistufige Durchführung von Änderungen

- Lese-Transaktionen greifen auf die lokale Kopie zu (soweit vorhanden)

- Änderungs-Transaktionen ändern die Primärkopie

- Primärkopie propagiert Änderungen an die Kopien (asynchron)

- Jeder Knoten hat mit jedem anderen Knoten je einen Ein- und Ausgabekanal, der im FIFO-Prinzip verwaltet wird

---

[6] Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans. on Software Engineering, May 1979, SE-5(3):188-194

© P. Dadam, 1999 - Verteilte Datenbanken und Client/Server-Systeme

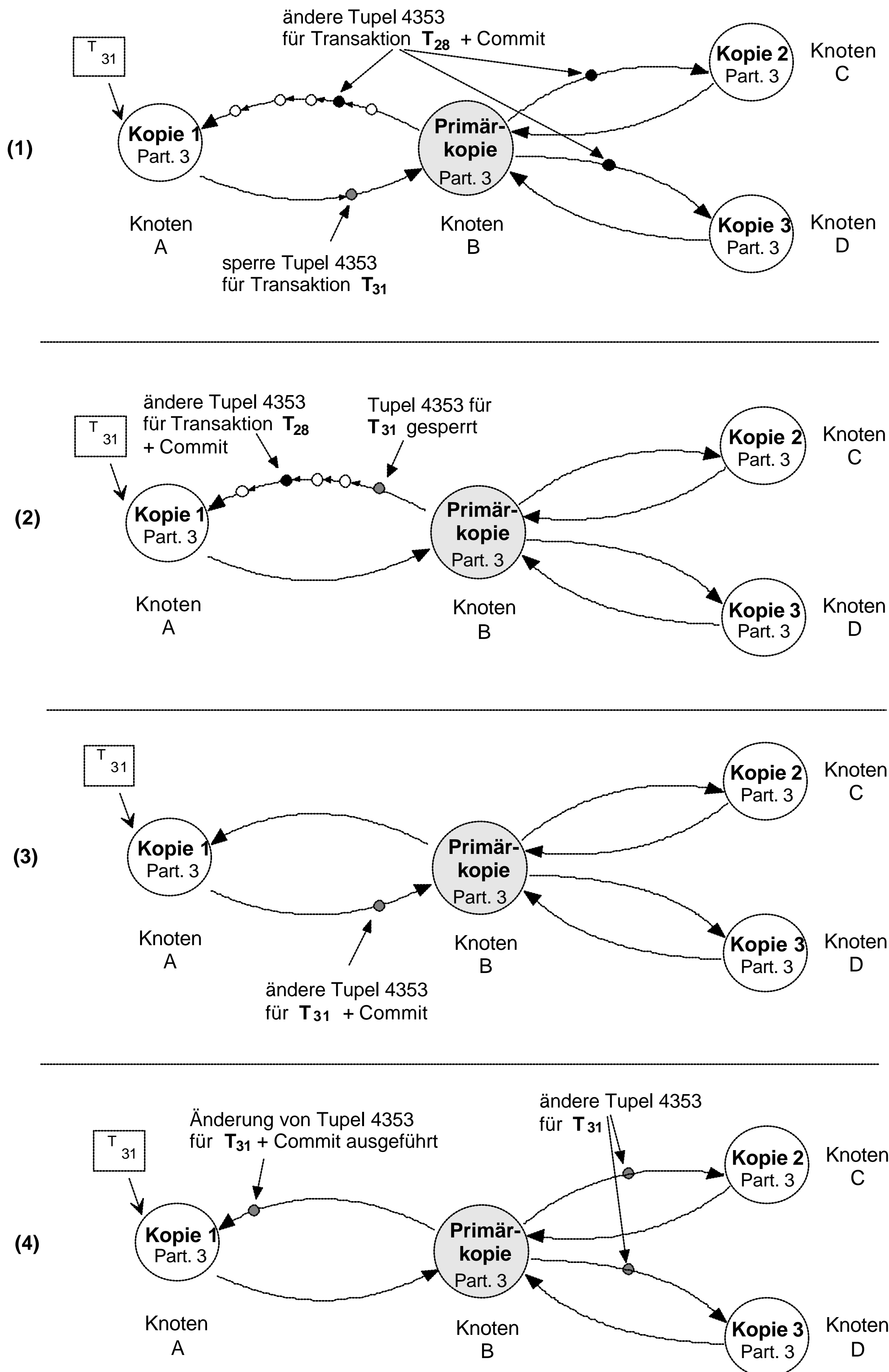


Abb. 9-3: Beispiel für Primary-Copy-Verfahren

❒ (Potentielle) Probleme beim Primärkopie-Ansatz:

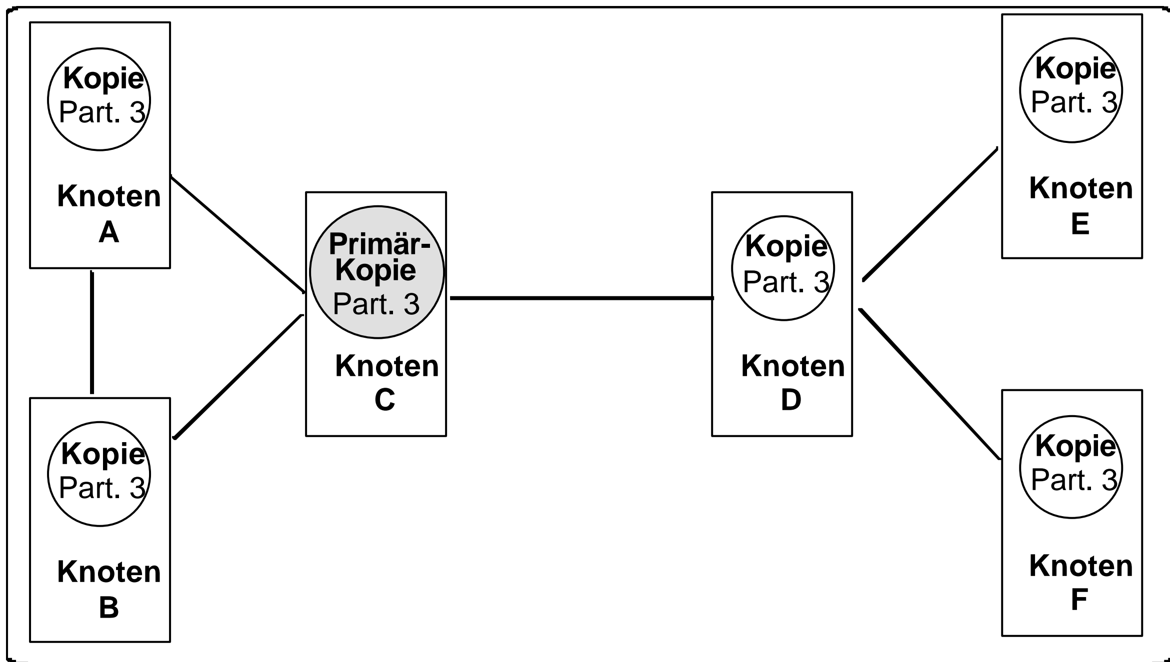
- Gewährleistung von Lese-Konsistenz
- Ausfall der Primär-Kopie

❒ Gewährleistung von Lese-Konsistenz

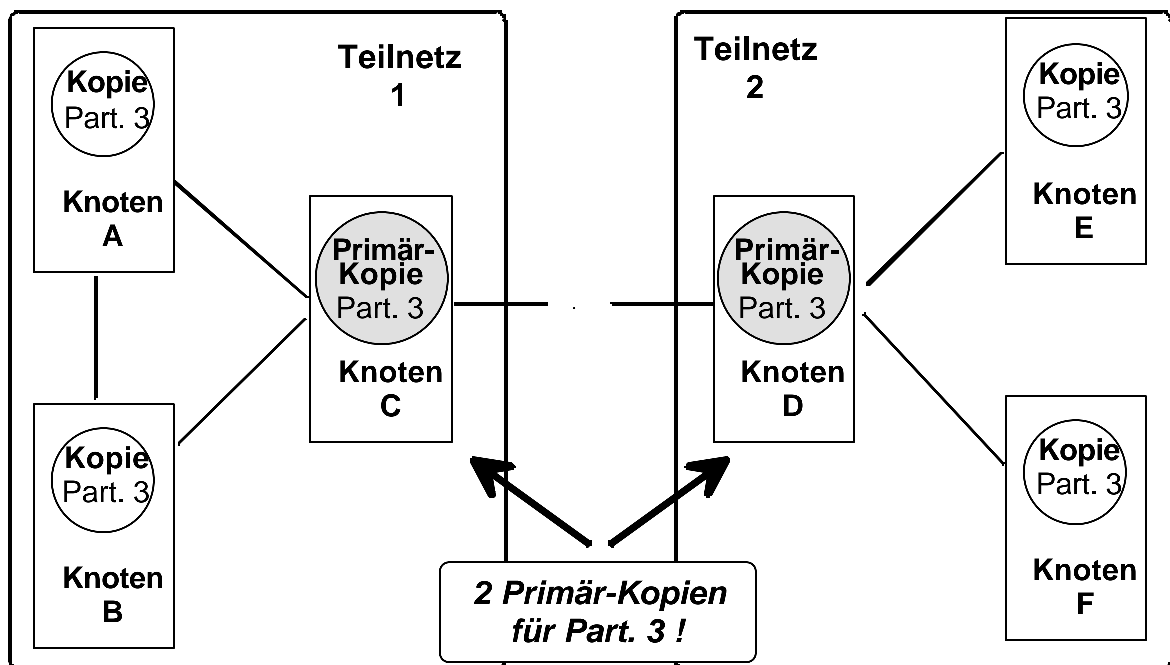
- Bei rein lokalem Lesen nicht sicher, ob alle relevanten Änderungen bereits durchgeführt
- Abhilfe:
  - Entsprechende Lesesperre(n) auf Primärkopie anfordern
  - Eingang Sperr-Bestätigung
    ⇒ "Pending updates" abgearbeitet
- Problem: Gewinn des lokalen Lesens i.w. wieder verloren
- Praktische Lösung heute: ***Snapshot-Refresh***

❒ Ausfall der Primär-Kopie (siehe Abb. 9-4)

- Vorschlag: Die verfügbare Kopie mit der höchsten KnotenID wird neue Primärkopie
- Problem: Primärkopie kann für einen oder mehrere Knoten nicht mehr verfügbar sein, weil
  - der entsprechende "Primär"-Knoten ausgefallen ist
  - eine Netz-Partitionierung auf getreten ist
- **Fazit:** Automatische Erzeugung einer neuen Primärkopie ist problematisch!



a) Situation vor der Netz-Partitionierung



b) Situation nach der Netz-Partitionierung und Bestimmung einer Primär-Kopie in Teilnetz 2

Abb. 9-4: Beispiel für Netzpartitionierung

9.3.1 Majority Consensus [7]

- "Urvater" aller Abstimmungs-Verfahren
- Synchronisation: Zeitstempelverfahren
- ...inspirierte viele Folge-Vorschläge
- Ausgangs-Situation: Kopien vollredundant gespeichert (d.h. an allen Knoten)
- Ansatz / Ziel:
  - Alle Kopien gleichwertig
  - Erhaltung der Arbeitsfähigkeit des vDBMS trotz einzelner Knotenausfälle
  - Möglichst wenig empfindlich gegen Netz-Partitionen
  - Konsistenthaltung der Kopien ohne (explizites) globales Sperren
  - Bei Zugriffskonflikten Mehrheits-Entscheidung wer gewinnt (***majority consensus***)
- Logischer Kommunikationsweg: **Ring**
  $\Rightarrow$ Entscheidung wird von Knoten zu Knoten weitergereicht

---

[7] R. H. Thomas, R. H.: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM Transactions on Database Systems, 4(2):180-209, 1979

❒ **Vorgehensweise**:

- ❍ Jedes DB-Objekt ist mit einem Zeitstempel der letzten Änderung (bzw. der Erzeugung) versehen

- ❍ Jede Änderungs-Transaktion:

  - Führt alle Änderungen zunächst rein lokal durch (macht sie aber noch nicht anderen Transaktionen sichtbar)

  - Erstellt eine Liste aller Ein- und Ausgabe-Objekte mit den jeweiligen Zeitstempeln

  - Schickt diese Liste zusammen mit ihrem Transaktions-Zeitstempel an alle anderen Knoten

  - Darf die Änderungen permanent machen, wenn die Mehrheit der Knoten ($> \frac{n}{2}$) zustimmt.

- Jeder Knoten stimmt über eingehende Änderungs-Anträge wie folgt ab und reicht sein Votum zusammen mit den anderen Voten an den nächsten Knoten weiter:

  - Er stimmt mit **ABGELEHNT**, wenn einer der übermittelten Objekt-Zeitstempel veraltet ist.

  - Er stimmt mit **OK** und markiert den Auftrag als **schwebend** ("pending"), wenn alle übermittelten Objekt-Zeitstempel aktuell sind und der Antrag nicht in Konflikt mit einem anderen Antrag steht.

  - Er stimmt mit **PASSIERE**, wenn alle Objekt-Zeitstempel zwar aktuell sind, der Antrag mit einem anderen schwebenden Antrag mit höherer Priorität (d.h. mit aktuellerem Zeitstempel) in Konflikt steht.

    Falls durch das Votum "PASSIERE" keine Mehrheit mehr zustande kommen kann, stimmt er mit **ABGELEHNT**.

  - Er **verzögert seine Abstimmung** über den Antrag, wenn der Antrag in Konflikt mit einem Antrag niedrigerer Priorität in Konflikt steht oder wenn einer der übermittelten Objekt-Zeitstempel höher als der des korrespondierenden lokalen Objektes ist. [8]

---

[8] Dann muß noch ein bereits beschlossener Update "unterwegs" sein.

- Annahme / Ablehnung:

  - Der Knoten, dessen Zustimmung (OK) dem Antrag die **Mehrheit** verschafft, erzeugt die **globale COMMIT-Meldung** für die gewünschte Änderung.

  - Jeder Knoten, der mit **ABGELEHNT** stimmt, löst ein **globales ABORT** des Änderungs-Auftrages (d.h. der Transaktion) aus.

  - Wird eine **Änderungs-Transaktion akzeptiert**, werden die Änderungen bei Eintreffen eingebracht (Ausnahme siehe unten) und die Objekt-Zeitstempel entsprechend aktualisiert.

  - Wird eine **Änderungs-Transaktion abgelehnt**, so werden die "verzögerten Abstimmungen" je Knoten jeweils überprüft, ob jetzt eine Entscheidung möglich ist.

  - Bei Ablehnung muß die Transaktion komplett wiederholt werden, einschließlich Objekt-Lesen

Anmerkungen:

- Lese-Transaktionen müssen für konsistentes Lesen wie (Pseudo-)Änderungs-Transaktionen behandelt werden

- Das Verfahren läßt zu, daß sich genehmigte Updates unterwegs "überholen". Beim Eintreffen eines Updates wird deshalb zunächst der Objekt-Zeitstempel gelesen und veraltete Updates ggf. ignoriert

- Die Knoten dürfen ein einmal getroffenes Votum nicht mehr ändern

❐ Beispiel 9-1:

Gegeben sei die in Abb. 9-5 dargestellte Ausgangssituation. Wir wollen zur Vereinfachung annehmen, daß es einen globalen Zeittakt gibt und daß an jedem Knoten jeweils innerhalb eines Taktschrittes evtl. eingehende Anforderungen geprüft, eine Entscheidung herbeigeführt und ggf. eine Nachricht an den nachfolgenden Knoten weitergeleitet wird, die dieser dann im nächsten Taktschritt verarbeitet. Commit- und Abort-Nachrichten werden parallel an alle Knoten verschickt und dort ebenfalls im nächsten Taktschritt verarbeitet. Zu Beginn haben alle Objektzeitstempel den Wert 0.

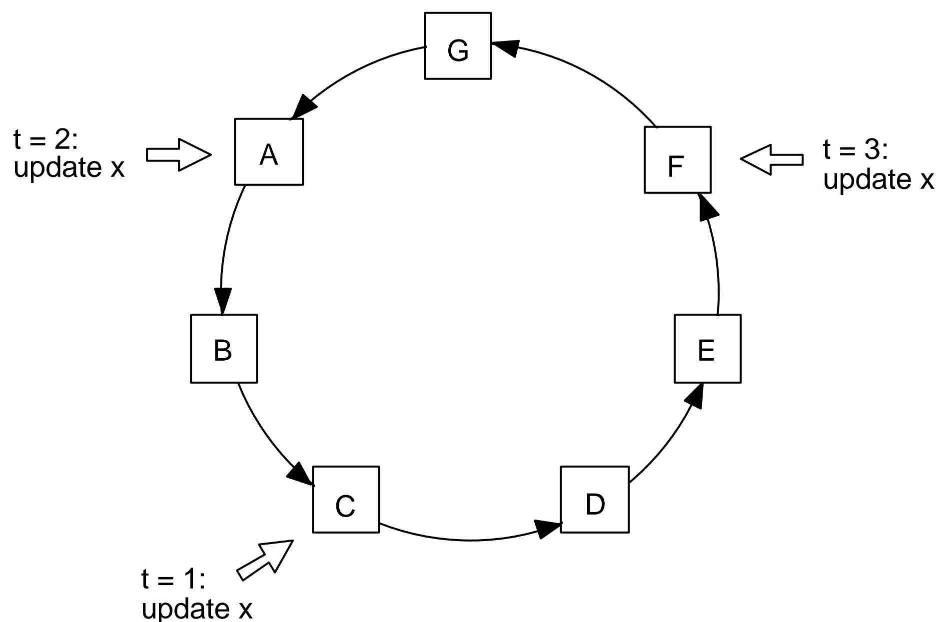


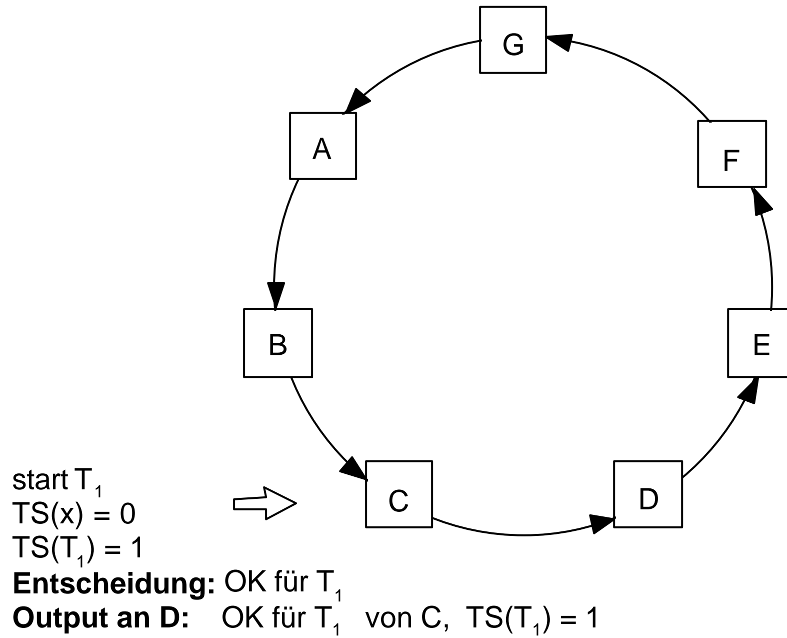
Abb. 9-5: Ausgangssituation

Zum Zeitpunkt t = 1 wird am Knoten C eine Updatetransaktion T_1 gestartet, die Objekt x verändern möchte.

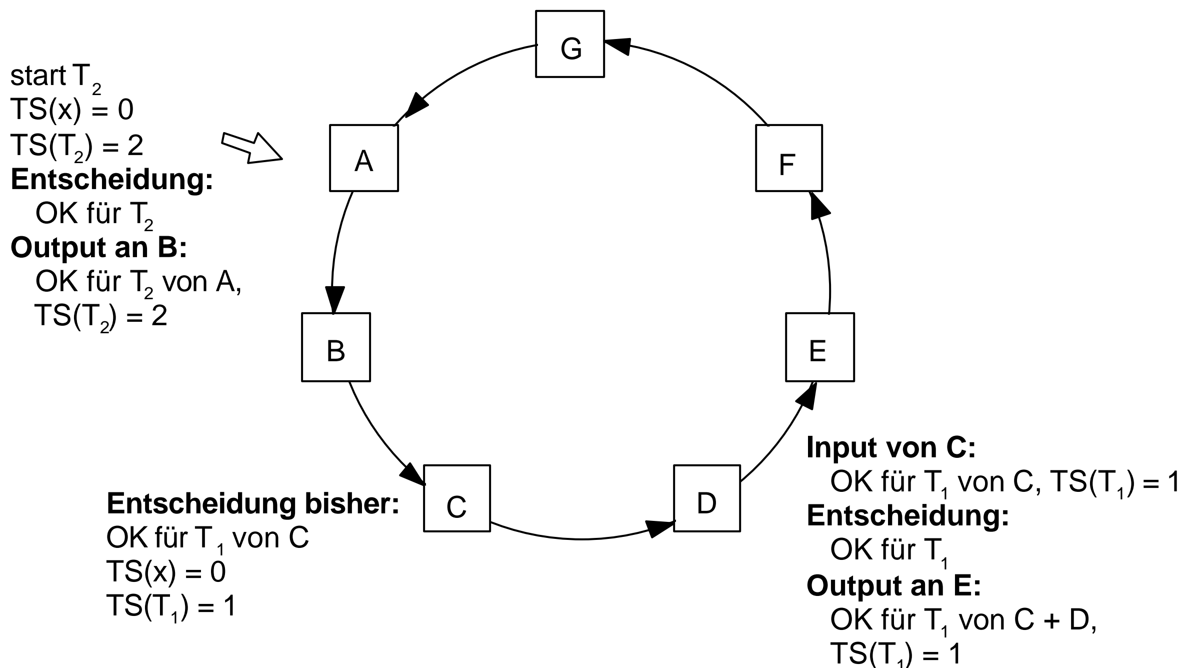
Über diese Anforderung wird dann Knoten D zum Zeitpunkt t = 2 entscheiden.

Zum selben Zeitpunkt (t =2) wird an Knoten A eine Updatetransaktion bzgl. Objekt x und zum Zeitpunkt t = 3 eine entsprechende Transaktion an Knoten F gestartet.

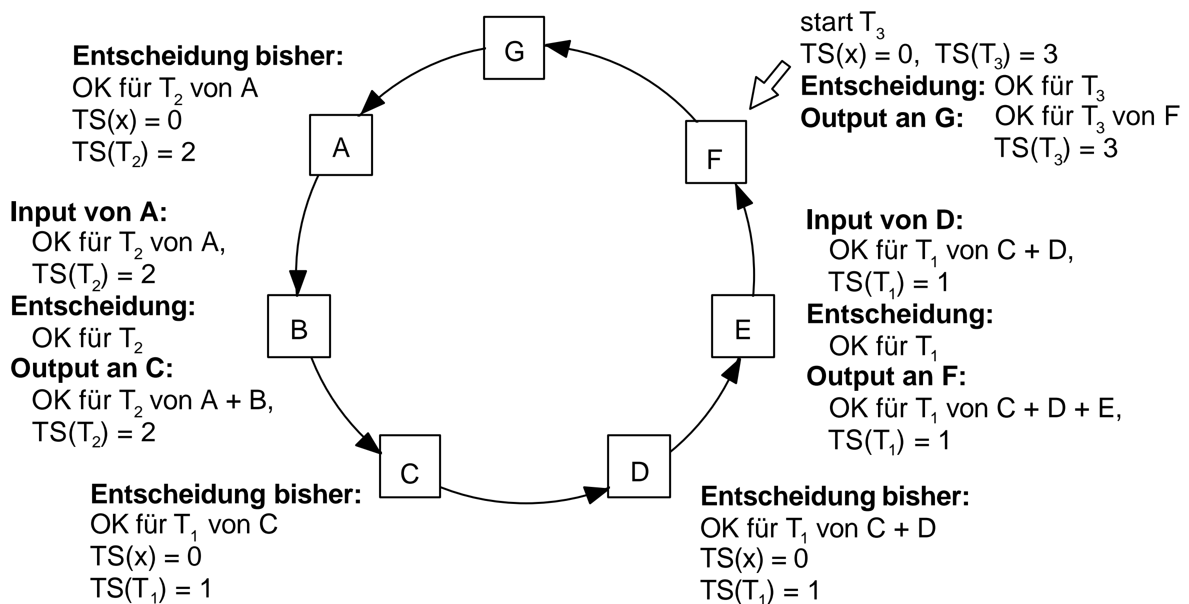
Situation zum Zeitpunkt **t = 1**:



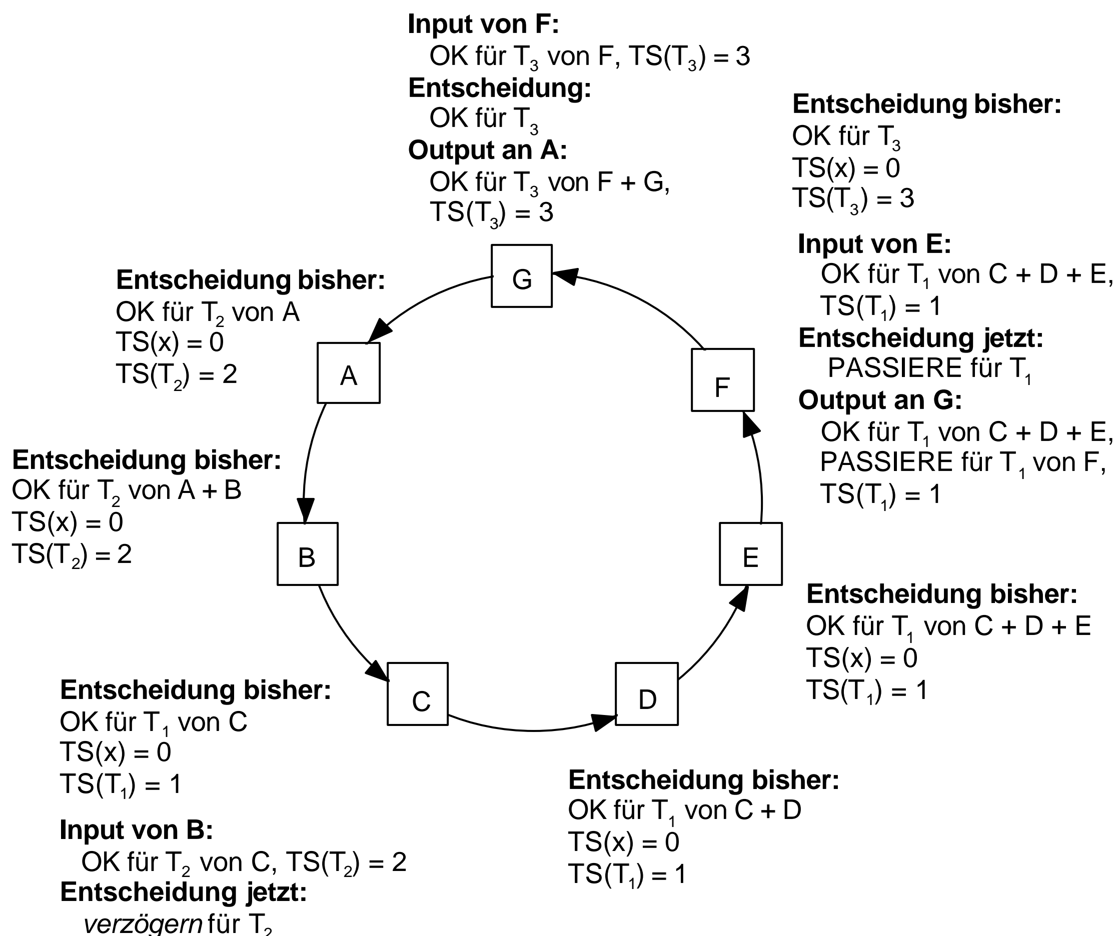
Situation zum Zeitpunkt **t = 2**:



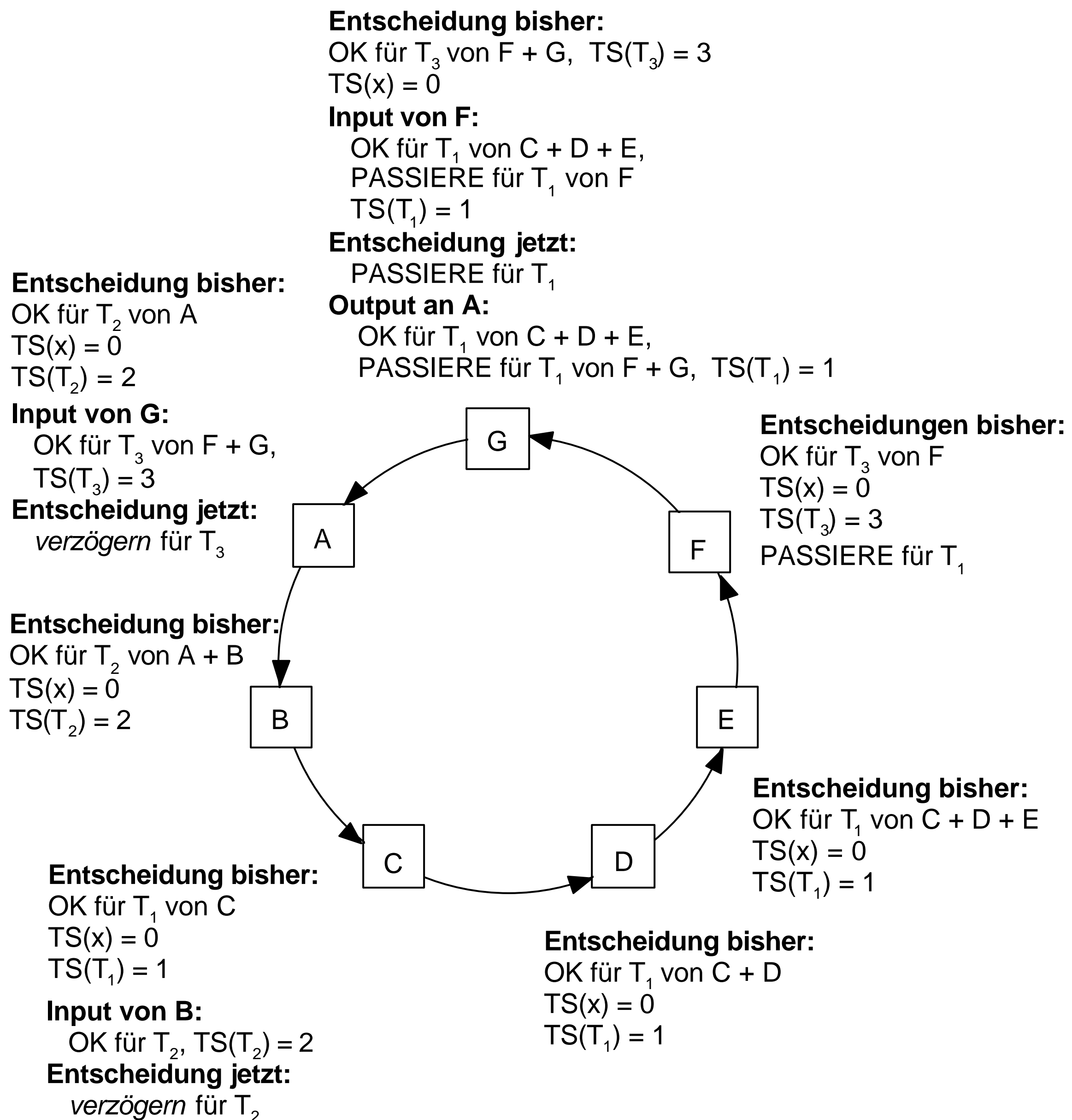
Situation zum Zeitpunkt **t = 3**:



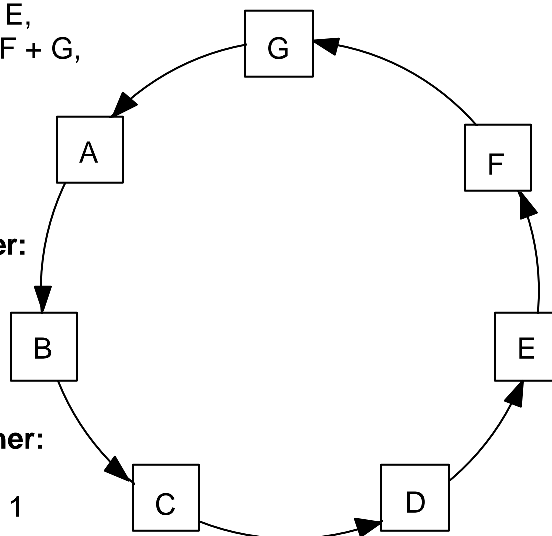
Situation zum Zeitpunkt **t = 4**:



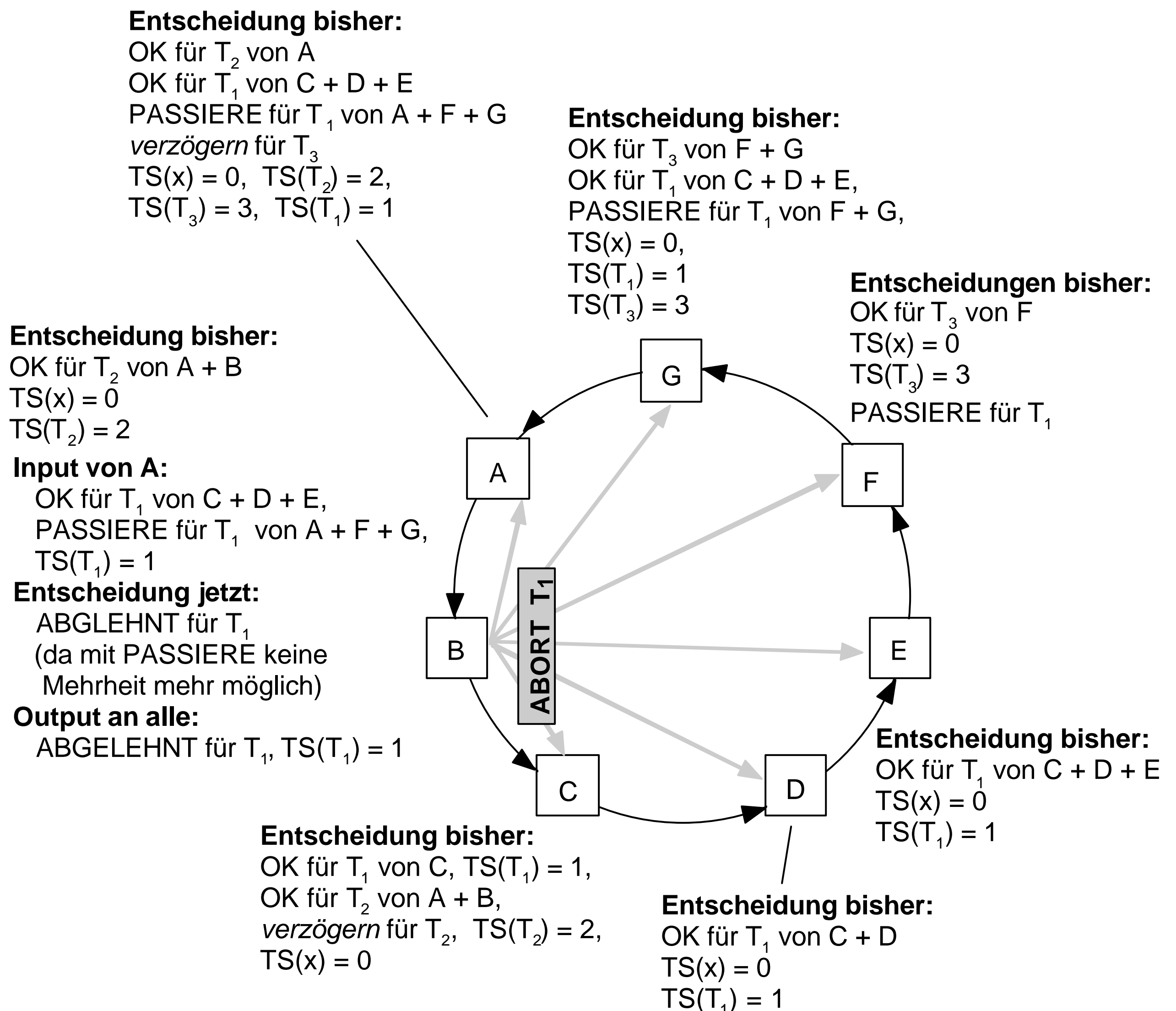
Situation zum Zeitpunkt **t** = **5**:



Situation zum Zeitpunkt **t = 6**:

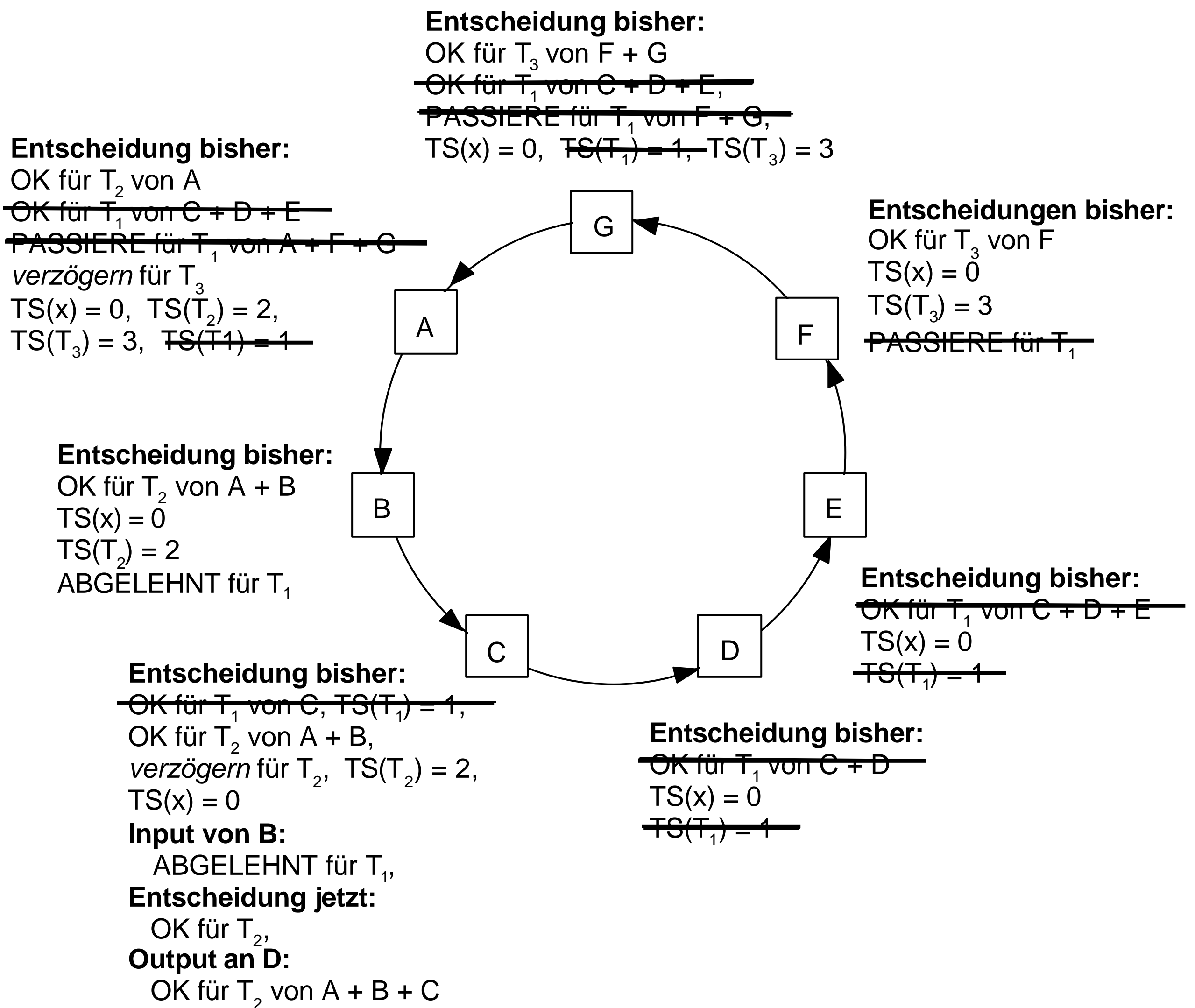


Situation zum Zeitpunkt **t = 7**:

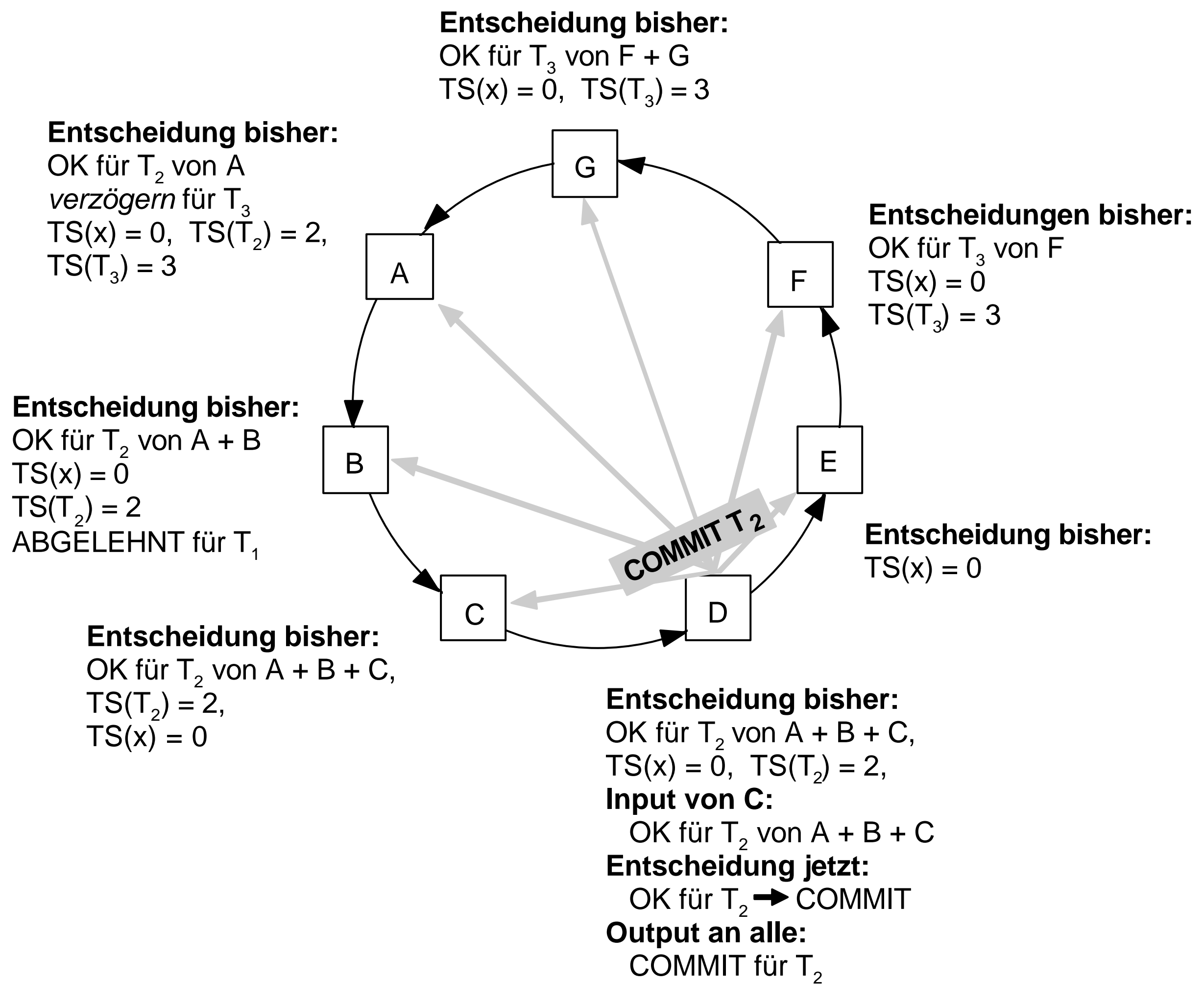


© P. Dadam, 1999 - Verteilte Datenbanken und Client/Server-Systeme

Situation zum Zeitpunkt **t** = **8**:



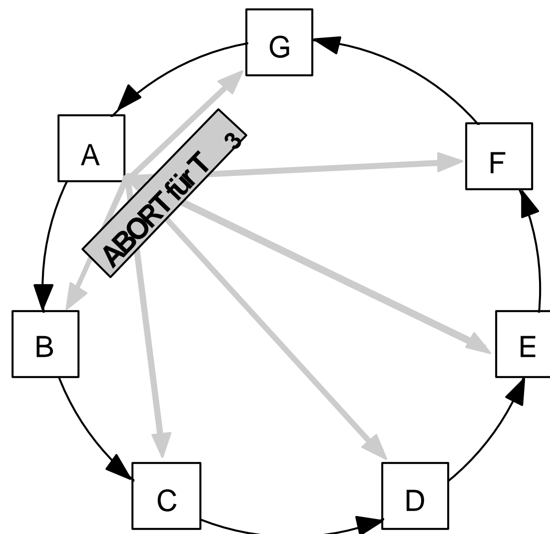
Situation zum Zeitpunkt **t = 9**:



Situation zum Zeitpunkt **t = 10:**

Entscheidung bisher:
OK für T_2 von A
verzögern für T_3
TS(x) = 0, TS(T_2) = 2,
TS(T_3) = 3
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2,
ABGELEHNT für T_3
(da Inputobjekte von
T_3 nun veraltet)
Output an alle:
ABORT für T_3

Entscheidung bisher:
OK für T_3 von F + G
TS(x) = 0, TS(T_3) = 3
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2



Entscheidungen bisher:
OK für T_3 von F
TS(x) = 0
TS(T_3) = 3
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2

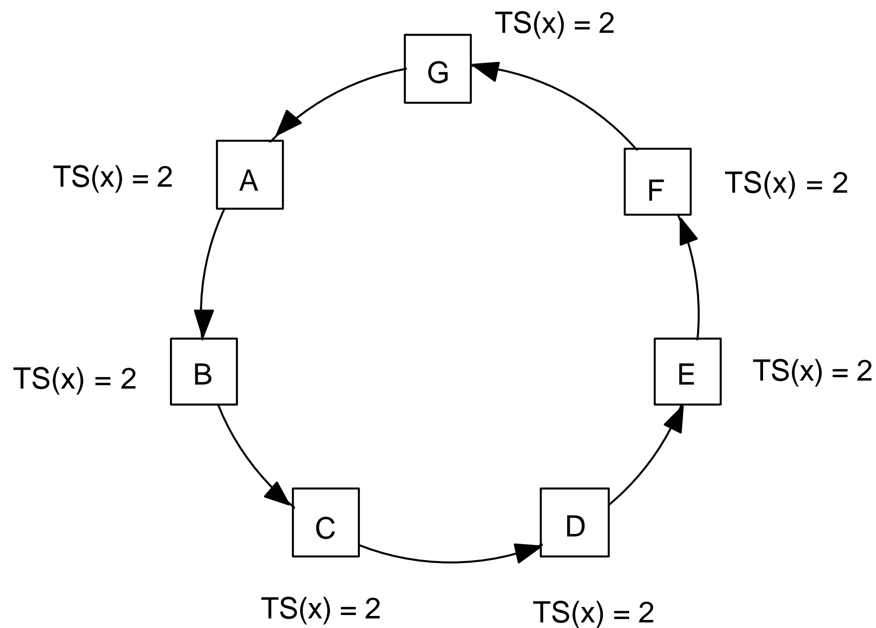
Entscheidung bisher:
OK für T_2 von A + B
TS(x) = 0
TS(T_2) = 2
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2

Entscheidung bisher:
TS(x) = 0
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2

Entscheidung bisher:
OK für T_2 von A + B + C,
TS(T_2) = 2,
TS(x) = 0
Input von D:
COMMIT für T_2
Entscheidung jetzt:
TS(x) = 2

Entscheidung bisher:
TS(x) = 2

Situation zum Zeitpunkt **t = 11:**



Anmerkung:

Die graphische Darstellung wurde in dieser Musterlösung gewählt, um die Dynamik des Abstimmungsprozesses besser illustrieren zu können. Kompakter, und bei vielen Knoten und Iterationen auch übersichtlicher, ist sicherlich eine *tabellarische Darstellung*, die etwa wie folgt aufgebaut sein könnte:

Iterat.	**KnotenID**	**Wert + TS lokaler Objekte**	**bisherige Entscheidungen**	**Input (von)**	**Entscheidung**	**Output (an)**

9.3.2 Dynamic Voting

- <u>Prinzip</u>: Majority Consensus mit dynamischer Quorums-Einteilung

- Problem mit statischem Quorum:

  Bei mehrfacher Netzpartitionierung / mehrfachen Knotenausfällen Quorumbildung u.U. nicht mehr möglich

Beispiel 9-2: Objekt 5fach repliziert, Majority Consensus-Verfahren

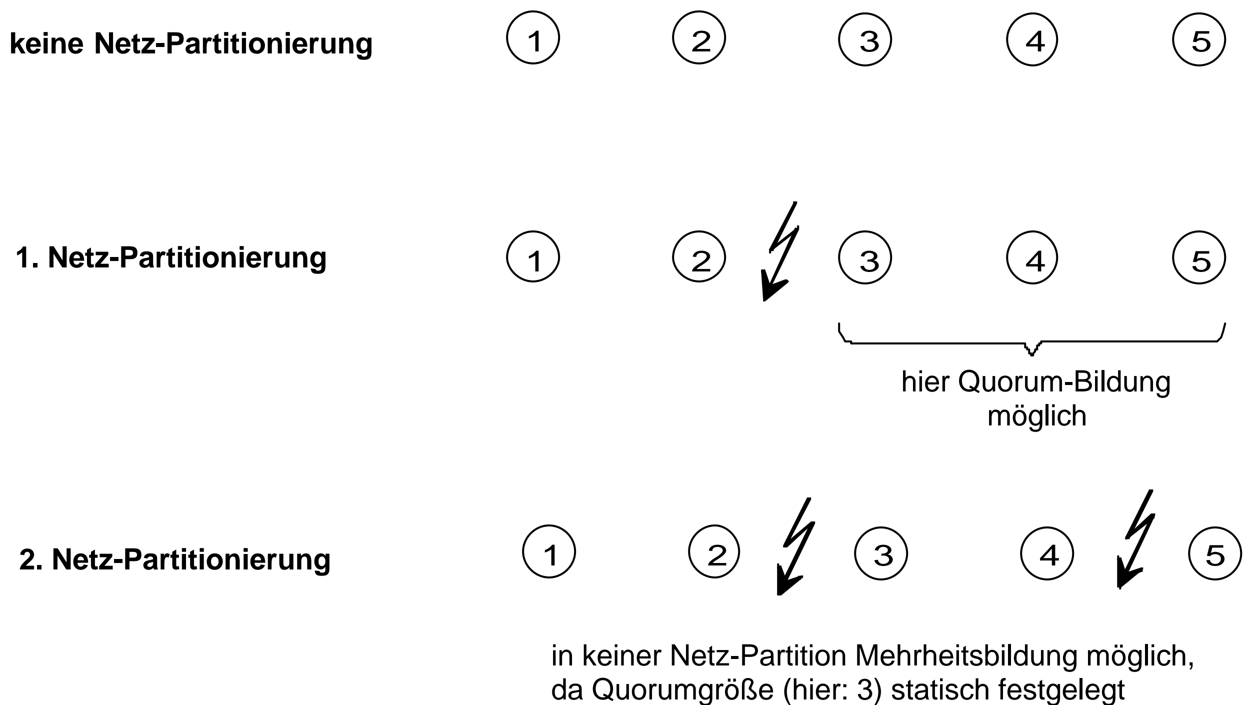


Abb. 9-6: Netzpartitionierung und statisches Voting

- Idee des Dynamic Voting:

  Nur noch diejenigen Knoten, die beim letzten Update beteiligt waren, besitzen Stimmrecht

- dazu notwendig: weitere Verwaltungsinformation je Kopie

  - VN: Versionsnummer der Kopie

  - SK: Anzahl der Knoten, die beim letzten Update der Kopie beteiligt waren (= gegenwärtige Gesamtstimmenzahl)

❒ Beispiel 9-3:

❍ **Ausgangssituation:** 9 Updates auf allen 5 Kopien erfolgreich
⇒ VN = 9, SK = 5

	1	2	3	4	5
VN	9	9	9	9	9
SK	5	5	5	5	5

❍ **Erste Netz-Partitionierung, anschl. Update-Anforderung an Knoten 3**

	1	2	⚡	3	4	5
VN	9	9		9	9	9
SK	5	5		5	5	5

Analyse:

Knoten 3 gehört zur Hauptpartition (mit Knoten 3,4,5), da Quorum-Bildung möglich: $\text{round}(\frac{SK}{2}) = \text{round}(\frac{5}{2}) = 3$

⇒ Update zulässig: neue VN = 10

⇒ nur Knoten 3, 4, 5 führen Update durch ⇒ SK = 3 (dort)

Resultat:

	1	2	⚡	3	4	5
VN	9	9		10	10	10
SK	5	5		3	3	3

- **Zweite Netz-Partitionierung, anschl. Update-Anforderung an Knoten 4**

	①	②	⚡	③	④	⚡	⑤
VN	9	9		10	10		10
SK	5	5		3	3		3

Analyse:

Knoten 4 gehört zur Hauptpartition (bestehend aus Knoten 3 und 4), da nur noch 3 Knoten stimmberechtigt (SK = 3) und im geg. Fall round($\frac{SK}{2}$) = round($\frac{3}{2}$) = 2 gilt

⇒ Update zulässig, $VN_{neu} = 11$

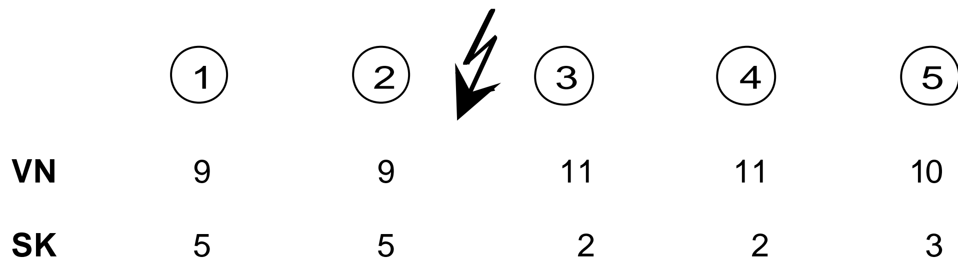
Resultat:

	①	②	⚡	③	④	⚡	⑤
VN	9	9		11	11		10
SK	5	5		2	2		3

Anmerkung:

Das "normale" Majority Consensus-Verfahren würde hier solange blockieren bzw. den Request zurückweisen (da hier als Quorum 3 benötigt wird), bis die (zweite) Netz-Partitionierung wieder beseitigt ist.

- **Zweite Netz-Partitionierung beendet, Update-Anforderung an Knoten 4, Wiederherstellung des Stimmrechts für Knoten 5**

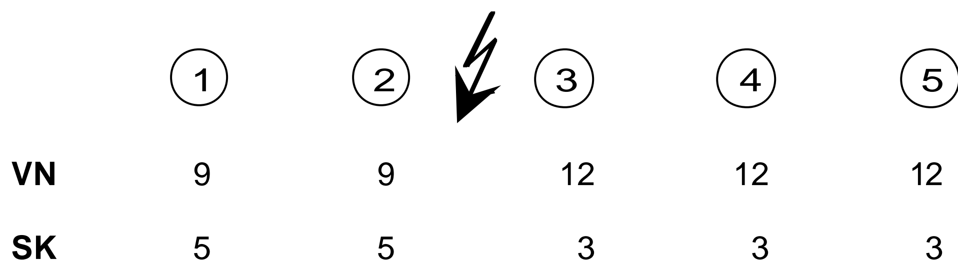


	1	2	3	4	5
VN	9	9	11	11	10
SK	5	5	2	2	3

Sammlung eines Quorums:

- Anfrage von Knoten 4 an alle erreichbaren Knoten
- Knoten 3 und 5 melden sich (jeweils mit VN und SK)
- Knoten 3 besitzt Stimmrecht (VN und SK sind aktuell) und stimmt Update-Anforderung zu
- Update kann durchgeführt werden: $VN_{neu} = 12$, $SK_{neu} = 3$
- Update mit VN_{neu} und SK_{neu} wird an Knoten 3 und 5 verschickt

Resultat:

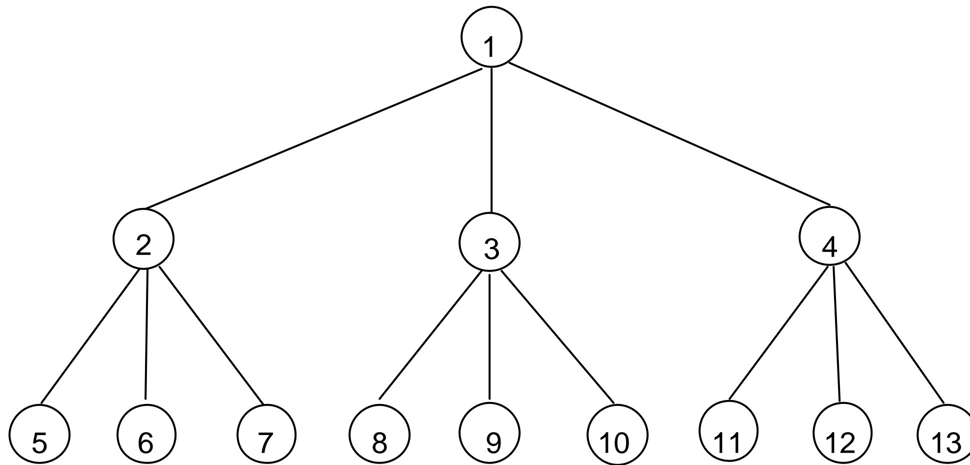


	1	2	3	4	5
VN	9	9	12	12	12
SK	5	5	3	3	3

Anmerkung:

Ein "deaktivierter" Knoten kann durch einen "Null-Update" eine Aktualisierung sowie Erlangung des Stimmrechts (für den nächsten Update) von sich aus initiieren.

9.3.3 Tree Quorum [9]



- Quorum q = < l, w >:

  ⇒ Sammle in l Ebenen jeweils mindestens w Stimmen[10]
  (je Teil-Baum-Ebene)

- **Beispiel 9-4:**

  q = < 2,2 > wäre erfüllt mit

  { 1, 2, 3 } oder { 1,2,4 } oder { 1, 3, 4 } oder

  { 2, 3, 5, 6, 8, 9 } oder { 2, 3, 5, 7, 8, 9 } oder ....

  (siehe Abb. 9-7)

---

[9] Agrawal, D.; El Abbadi, A.: The generalized tree quorum protocol: An efficient approach for managing replicated data. ACM Trans. on Database Systems, 17(4):689-717, December 1992

[10] bzw. alle Stimmen, falls die betrachtete Ebene weniger als w Knoten aufweist

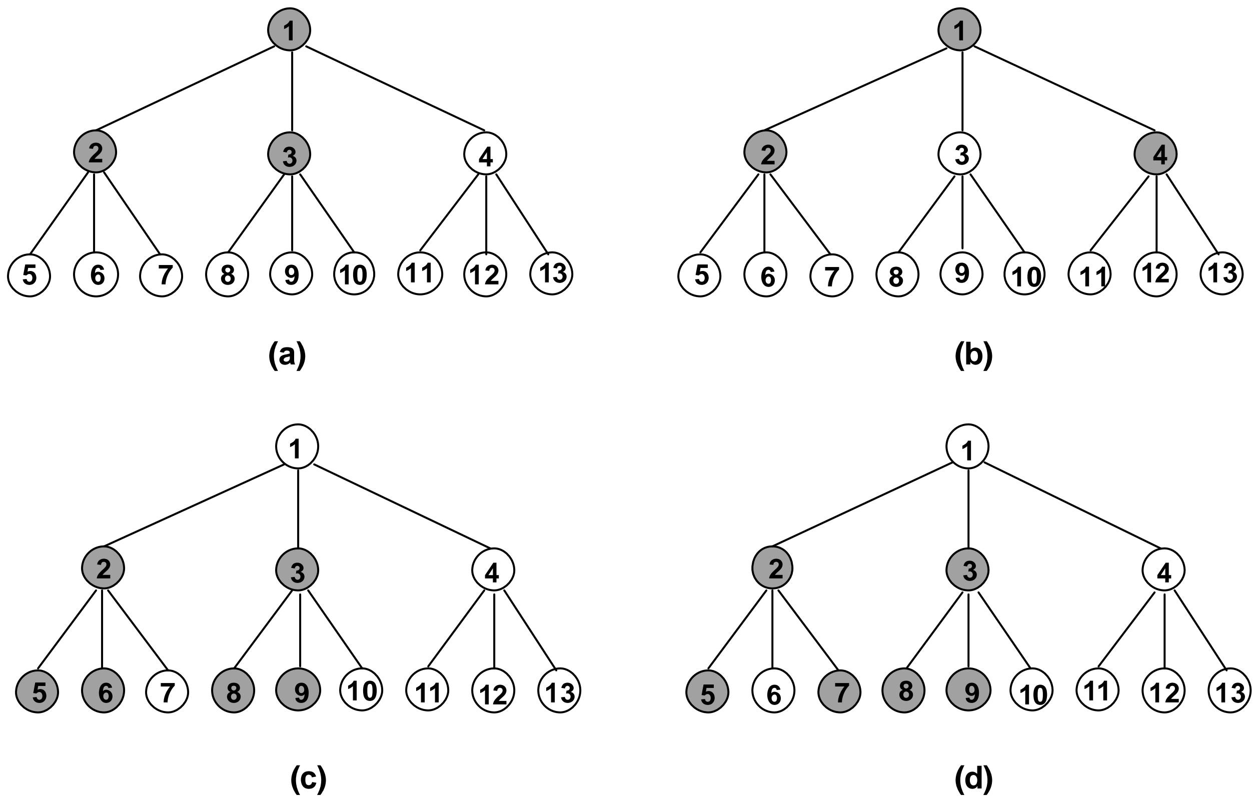


Abb. 9-7: Auswahl möglicher Konstellationen für ein Schreibquorum

❒ Zu beachten:

l und w müssen natürlich so gewählt werden, daß die *Quorum-Überschneidungsregel* (siehe Definition 9-1) eingehalten wird.

❐ Beispiel 9-5:

Gegeben sei der in Abb. 9-8 dargestellte Baum mit $v = 3$ und $h = 5$. Für ein Schreibquorum Q_W müssen in diesem Fall mindestens 3 Ebenen und je Ebene mindestens zwei Zustimmungen erreicht werden. Es gilt also:

$Q_W = Q_W(3,2)$

Ebene 1: Keine Stimme, muß daher ersetzt werden durch 2 Stimmen auf Ebene 2.

Ebene 2: Eine Stimme auf dieser Ebene durch Knoten 2, zweite Stimme auf dieser Ebene wird nicht direkt erreicht, sondern wird ersetzt durch 2 Stimmen auf Ebene 3, und zwar durch die Kinder von Knoten 4.

usw. (siehe schwarz markierte Knoten in Abb. 9-8).

Frage: **Könnte es bei dieser Konstellation einer zweiten Transaktion gelingen, ebenfalls ein Schreibquorum zu erhalten?**

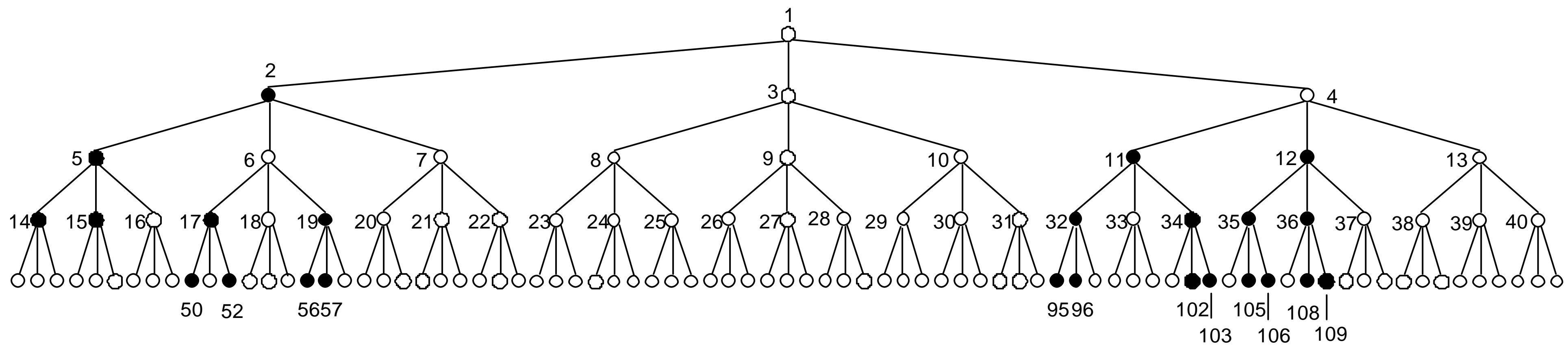


Abb. 9-8: Tree Quorum (Beispiel 9-7)

Anmerkungen:

- Majority Consensus würde die Zustimmung von mind. 61 Knoten benötigen, im dargestellten Fall genügen 24 Stimmen.
- Wäre die Wurzel mit dabei, so hätten sogar 7 Stimmen genügt.

❐ **Bewertung**:

- ⊕ Quorum-Bildung mit weniger als $\frac{n}{2}+1$ Stimmen möglich
- ⊕ billiges Lesen ohne hohe Kosten beim Schreiben
- ⊖ Quorumsbildung nur entlang der logischen Struktur möglich
- ⊖ höherer Verwaltungsaufwand
- ⊖ nur bei hohem Replikationsgrad interessant

9.3.4 Data Patches [11]

- Idee: Konfliktauflösungsstrategie beim Datenbankentwurf festlegen
  ⇒ individuelle Regeln für jede einzelne Relation

- **Tupel-Einfügungsregeln**

  ... anzuwenden, wenn ein Tupel z.B. nur in Netz-Partition P1 und nicht in P2, ... eingefügt wurde

  - Keep-Regel: Tupel einfügen in P2, ...
  - Remove-Regel: Tupel löschen in P1
  - Programm-Regel: Aufruf eines Programms mit Tupel als Parameter
  - Notify-Regel: Systemadministrator benachrichtigen
    ⇒ manuelle Konfliktauflösung

---

[11] Garcia-Molina, H.: Data-patch: Integrating inconsistent copies of a database after a partition. Proc. 3th IEEE Symposium on Reliable Distributed Systems, New York, October 1983, pp. 38-48

❒ Tupel-Integrationsregeln

... anzuwenden, wenn Tupel mit gleichem Schlüssel sowohl in P1 als auch in P2 eingefügt worden ist

- ❍ Latest-Regel: zuletzt eingefügtes Tupel gilt
- ❍ Primary-Regel: Tupel an Knoten k gilt
- ❍ Arithmetik-Regel: neuer Wert = Wert1 + Wert2 - alter Wert
- ❍ Programm-Regel: siehe oben
- ❍ Notify-Regel: siehe oben

❒ Vorgehen bei Wiedervereinigung

1. Bestimme alle Tupel, die nur in einer Netz-Partition eingefügt wurden

   ➠ wende Tupel-Einfügeregel an

2. Bestimme alle Tupel, die in beiden Partitionen vorkommen

   ➠ wende Tupel-Integrationsregel an

❒ **Beispiel 9-6:** Bank-DB (wie oben)

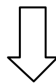
Konto-Relation

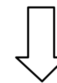
Konto#	Name	Kontostand	Einfügungsregel	Integrationsregel
			KEEP	ARITHMETIC

Ausgangssituation:

Knoten 1		
Konto#	Name	Kontostand
1723	Maier	1.000 DM

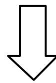
Knoten 2		
Konto#	Name	Kontostand
1723	Maier	1.000 DM

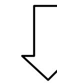




Abhebung: 200 DM

Abhebung: 300 DM





Knoten 1		
Konto#	Name	Kontostand
1723	Maier	800 DM

Knoten 2		
Konto#	Name	Kontostand
1723	Maier	700 DM

Wiedervereinigung: Integrationsregel = ARITHMETIC

$$\Rightarrow \text{Kontostand}_{neu} := \text{Kontostand}_1 + \text{Kontostand}_2 - \text{Kontostand}_{alt}$$
$$= 800\text{ DM} + 700\text{ DM} - 1.000\text{ DM}$$
$$= 500\text{ DM}$$

Nach Konfliktauflösung:

Knoten 1		
Konto#	Name	Kontostand
1723	Maier	500 DM

Knoten 2		
Konto#	Name	Kontostand
1723	Maier	500 DM

9.3.5 Semantikbasiertes Replikationsmanagement

- Verfahren "***Multi Copy Compatible***" [12]
  ⇒ Kommutatitivät der TA zusätzlich zur Serialisierbarkeit

- Idee/Ansatz:

  - Einteilung der TAs in kommutative (C-TA) und nicht-kommutative TAs (NC-TA)

  - Ausführung von C-TAs auf verschiedenen Kopien in unterschiedlicher Reihenfolge möglich

Kopie 1	Kopie 2
$C\text{-}TA_1$, $C\text{-}TA_2$	$C\text{-}TA_2$, $C\text{-}TA_1$

  ⇒ Update einer C-TA zuerst lokal durchführen, erst nach COMMIT andere Kopien asynchron akualisieren.

  - Ersetzen von NC-TA durch "äquivalente" C-TAs und Ausführung von diesen als normale C-TAs aus

---

[12] Kumar, A.; Stonebraker, M.: Semantics based transaction management techniques for replicated data. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Chicago, Illinois, June 1988, pp. 117-125

❐ **Verfahren**:

Für jeden Knoten wird ein Zustands-Vektor (nc_i ,c_i) verwaltet:

c_i = # von C-TAs, die auf Knoten i beendet wurden

nc_i = # von NC-TAs, die auf Knoten i beendet wurden

Ausführung von C-TAs:

1. Führe den Update auf der lokalen Kopie incl. Commit durch

2. Aktualisiere c_i

3. Übergebe nach dem Commit die TA an ein Spool-Programm, das diese an die anderen Kopien weiterleitet[13]

   ⇒ asynchrones Update

   ⇒ verschiedene Versionen eines Objektes [14]

---

[13] dort TA-Ausführung wiederum durch ein Spool-Programm (siehe unten)

[14] D.h. Kopien mit aktuellen und veralteten Werten existieren gleichzeitig

Ausführung von NC-TAs:

1. Sammle ein Quorum (von Knoten) durch Sperren einer entsprechenden Anzahl (Mehrheit) an Kopien

2. Wähle daraus diejenige Kopie mit dem höchsten nc-Wert aus (= aktuellste Kopie bzgl. NC-TAs)

3. Führe dort den NC-TA-Update durch

4. Bestimme eine "äquivalente" C-TA für den Update auf den anderen Kopien

5. Aktualisiere synchron die Quorum-Kopien mittels C-TA und aktualisiere dort auch den nc-Wert

6. Gib die Sperren auf den Quorum-Kopien frei (Commit) und übergebe die C-TA an das Spool-Programm
   (für den asynchronen Update der übrigen Kopien)

Aufgaben des Spool-Programms:

- Annahme der Update-Nachricht (C-TA-Update) des lokalen Knotens und Weiterleitung/Verteilung an die anderen Knoten
- Empfang von Update-Nachrichten anderen Knoten und Durchführung des entsprechenden lokalen Updates
- Aktualisierung des lokalen Zustands-Vektors

Problem:

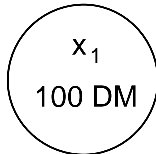
- Asynchrones Update führt zu unterschiedlich aktuellen Kopien
- Es ist nicht sichergestellt, daß im Quorum einer NC-TA eine Kopie enthalten ist, die alle C-Updates widerspiegelt
- DAMIT MUSS DIE ANWENDUNG BEI DIESEM VERFAHREN LEBEN!

Mögliche Abschwächung:

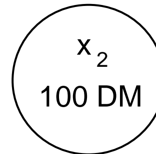
- Durchführung von NC-TA-Updates (durch die Spool-Programme) mit hoher Priorität
- Periodische Synchronisation der Kopien (damit aber wieder Parallelitätsverlust!)

❐ Beispiel 9-7:

Bank-DB, Konto x auf Knoten 1 und 2 repliziert



Knoten 1



Knoten 2

Einzahlungs-TA:	$T_E(e,x) = x + e$	:	Zahlt e DM auf Konto x ein
Auszahlungs-TA:	$T_A(e,x) = x - e$	:	Hebt e DM von Konto x ab
Zinszahlungs-TA:	$T_Z(x) = x + 10\%$	:	Erhöht Konto x um 10%

C-TA: T_E, T_A

NC-TA: T_Z

Ausführung der NC-TA T_Z:

1. Berechnung der Zinsen (= 10 DM) auf Kopie x_1 und Aktualisierung von Kopie x_1 (= COMMIT)

2. Übermittlung der Zins-Gutschrift als Einzahlungs-TA $T_{E_Z}(s,x_2) = T_E(10\ DM,x_2)$ an Kopie x_2

- **Problem:**

  Durch asynchrones Ändern treten Kopien mit unterschiedlichem Aktualitätsgrad auf $\Rightarrow$ keine 1-Kopie-Äquivalenz!

- Deshalb gelegentliche Synchronisationsphasen notwendig
  - dann keine Annahme weiterer TA-Aufträge
  - Ende der Synchronisationsphase, wenn alle asynchronen Änderungen beendet sind
  - Häufigkeit der Synchronisationsphasen abhängig von den Anforderungen der Anwendung

- **Bewertung semantische Verfahren** (allgemein)
  - (+) potentiell höherer Durchsatz
  - (+) potentiell höhere Verfügbarkeit
  - (-) nicht allgemein einsetzbar
  - (-) semantisches (Anwendungs-)Wissen notwendig

9.4 Abschließende Bemerkungen

- Vorgestellte Verfahren sollten Einblick in das Spektrum an verschiedenen Möglichkeiten aufzeigen

- nur einige wenige Verfahren hier vorgestellt, große Anzahl von Vorschlägen in der Literatur[15]

- Replikationsverfahren sind auch in den Kontexten

  - Mobile Computing
  - Disconnected Clients
  - Groupware

  von großer Bedeutung.

---

[15] siehe Abschnitt 9.5 "Ergänzende Literatur". In Dadam: "Verteilte Datenbanken und Client/Server-Systeme ...." sowie in dem Artikel von T. Beuter und P. Dadam findet sich ein umfangreiches und relativ aktuelles Literaturverzeichnis.

© P. Dadam, 1999 - Verteilte Datenbanken und Client/Server-Systeme

9.5 Ergänzende Literatur

- ❐ P. Dadam: Verteilte Datenbanken und Client/Server-Systeme, ...
- ❐ spezielle Artikel, wie angegeben

Überblicksartikel:

- ❐ Beuter, T. Dadam, P.: Prinzipien der Replikationskontrolle in verteilten Datenbanksystemen, Informatik Forschung und Entwicklung, Band 11, Heft 4, November 1996, S. 203-212
- ❐ U. M. Borghoff: Fehlertoleranz in verteilten Dateisystemen - Eine Übersicht über den heutigen Entwicklungsstand bei den Votierungsverfahren. Informatik Spektrum, Februar 1991, 14(1):15-27
- ❐ S. B. Davidson, H. Garcia-Molina, and D. Skeen: Consistency in partitioned networks. ACM Computing Surveys,September 1985, 17(3):341-370
- ❐ S. Hyuk Son: Replicated data management in distributed database systems. ACM SIGMOD Record, December 1988, 17(4):62-69