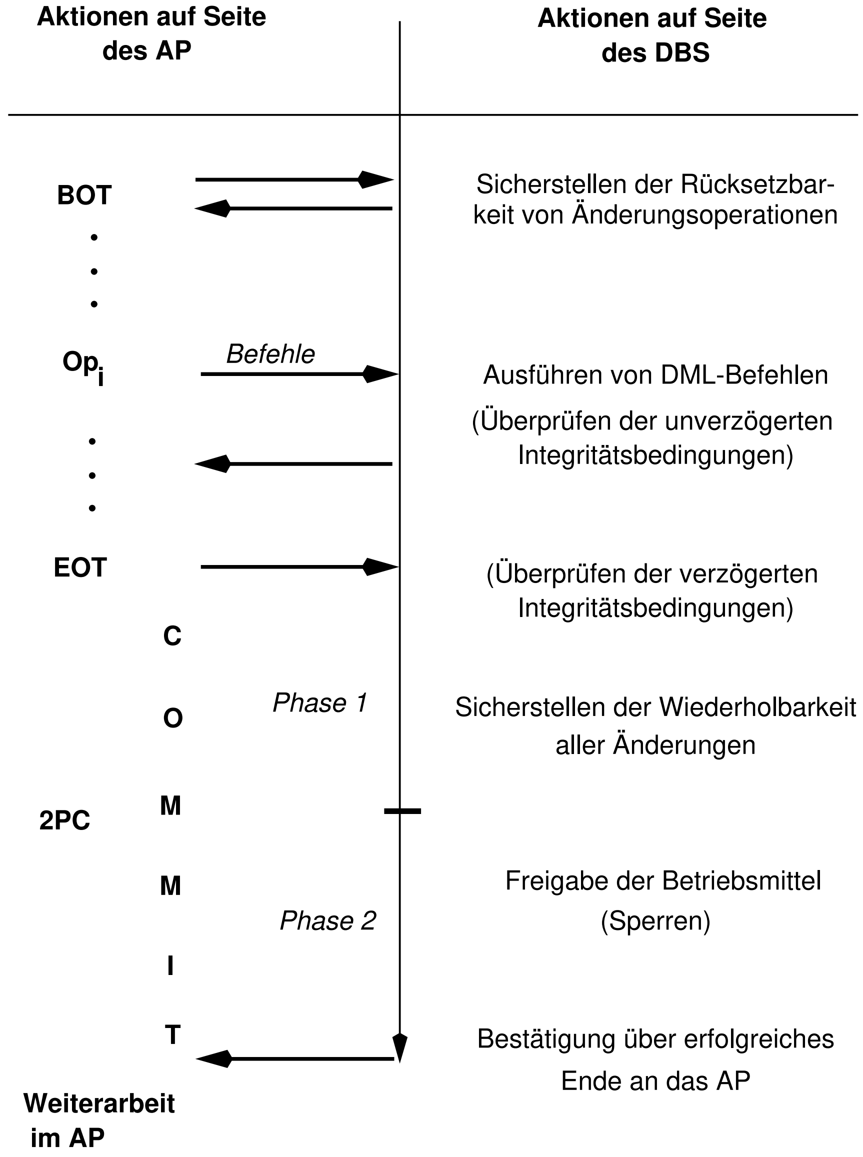
6. Verteilte Transaktionsverwaltung

- **Einführung**
- **Commit-Protokolle**
  - Anforderungen
  - Basis-Protokoll (2-Phasen-Commit)
  - Variationen des 2PC
  - 1-Phasen-Commit
  - 3-Phasen-Commit
- **Integritätssicherung**
- **Synchronisation**
  - Verfahrensüberblick
    - Sperrverfahren
    - semantische Synchronisation
    - Optimistische Verfahren
    - Zeitmarkenverfahren
  - Deadlock-Vermeidung
  - Deadlock-Auflösung
- **Logging und Recovery**
  - Welche Techniken eignen sich für den verteilten Fall?
    - Logging
    - Sicherungspunkte
    - Ersetzungs- und Einbringstrategien
  - Geräte-Recovery
    - Erstellung von Archivkopien
    - Systemweite Synchronisation

Transaktionsverarbeitung in zentralisierten DBS

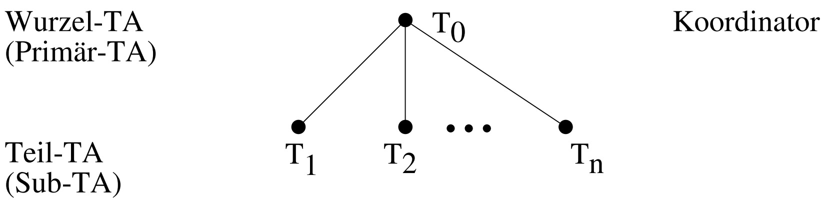


Transaktionsverwaltung in verteilten DBS

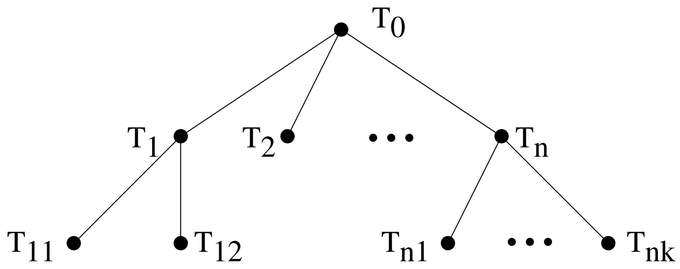
- **ACID-Eigenschaften** sind auch im verteilten Fall zu garantieren
  - **A**tomarität: Alles oder Nichts
  - **C**onsistency ➥ Integritätssicherung
  - **I**solation ➥ Synchronisation (Concurrency Control)
  - **D**auerhaftigkeit
- **Atomarität** durch globales Commit-Protokoll
- **Integritätssicherung**
  - verteilte Überwachung von Integritätsbedingungen, v. a. bei fragmentierten Relationen
  - Ausführung verzögerter Integritätsbedingungen im Rahmen des Commit-Protokolls
- **Synchronisation**
  - Wahrung der globalen Serialisierbarkeit
  - rechnerübergreifende Abhängigkeiten (globale Deadlocks u. ä.)
- **Logging und Recovery**
  - erweitertes Fehlermodell
  - Robustheit gegenüber partiellen Fehlern, insbes. Kommunikationsfehlern (Netzwerkpartitionierungen u. ä.)
- **Replikation**
  - Wahrung von Replikationstransparenz
  - Optimierung von Leistung und Verfügbarkeit

Transaktionsstruktur

- **Grundeinheit der Verarbeitung**
  - TA mit BOT, OP1, OP2, ..., OPn, COMMIT
  - ACID-Eigenschaften werden gewährleistet
- **Kontrollstruktur in einer verteilten Umgebung**



- **Verallgemeinerung**
  - beliebige Schachtelung der Teil-TA
  - *Transaktionsbaum* repräsentiert Aufrufbeziehungen



- **Konzept der geschachtelten Transaktionen** wird meist nicht unterstützt (obwohl wünschenswert) !
  - isoliertes Rücksetzen von Sub-TA
  - TA-interne Synchronisation zwischen Sub-TA
  - Parallelausführung von Sub-TA

Commit-Protokolle

- **Sicherstellen der *Atomarität* verteilter Transaktionen**
  durch rechnerübergreifendes Mehrphasen-Commit-Protokoll

- **Anforderungen an geeignetes Commit-Protokoll:**
  - Korrektheit
  - Geringer Aufwand (#Nachrichten, #Log-Writes)
  - Geringe Antwortzeitverlängerung
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern
  - Jeder an einer verteilten TA-Ausführung beteiligte Rechner soll möglichst lange das Recht des einseitigen TA-Abbruchs (unilateral abort) haben

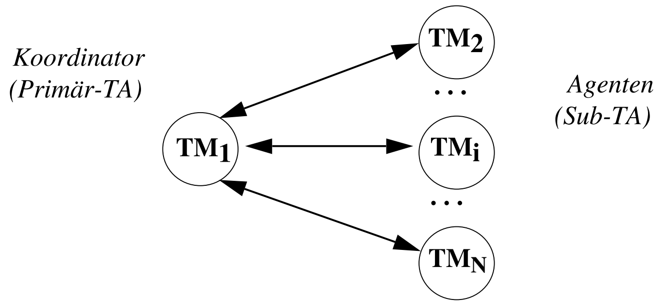
    ➥ Knotenautonomie

"Nicht-Fehler-Fall" ist zu optimieren

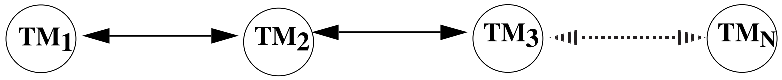
- **Wesentliche Alternativen**
  - 2-Phasen-Commit
    - zentral, linear, hierarchisch
  - 1-Phasen-Commit
  - 3-Phasen-Commit
  -

Kommunikationsstrukturen für verteilte Commit-Protokolle

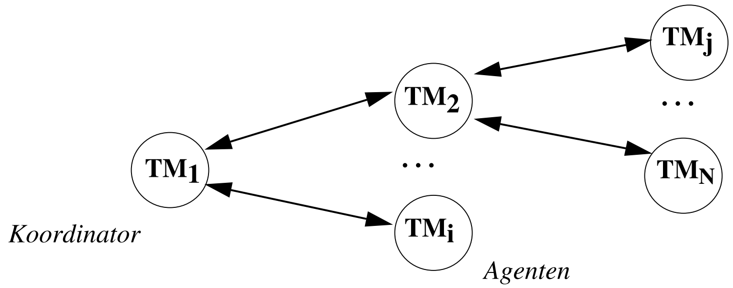
- Ausführung des Commit-Protokolls erfolgt durch ***Transaktions-Manager (TM)*** an jedem Knoten (1 Koordinator + N-1 Agenten)



Zentralisierte Commit-Struktur

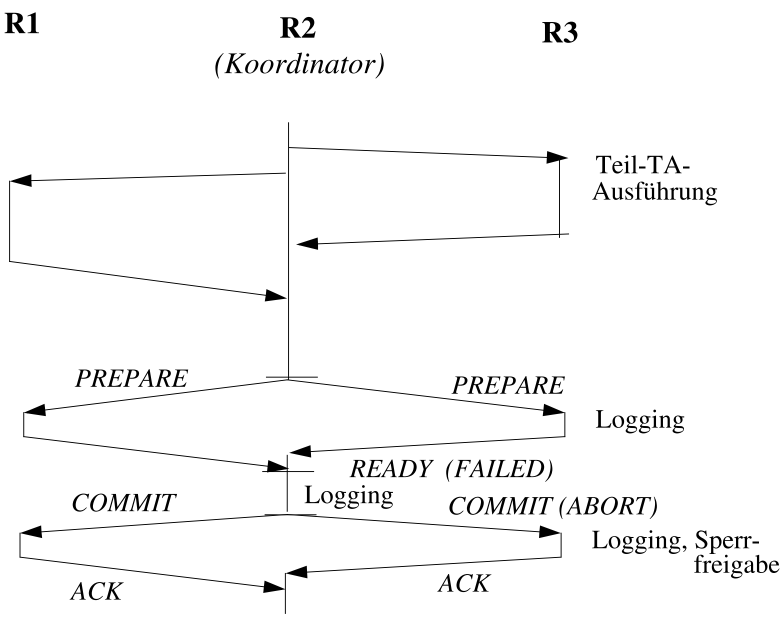


Lineare Commit-Struktur



Hierarchische Commit-Struktur

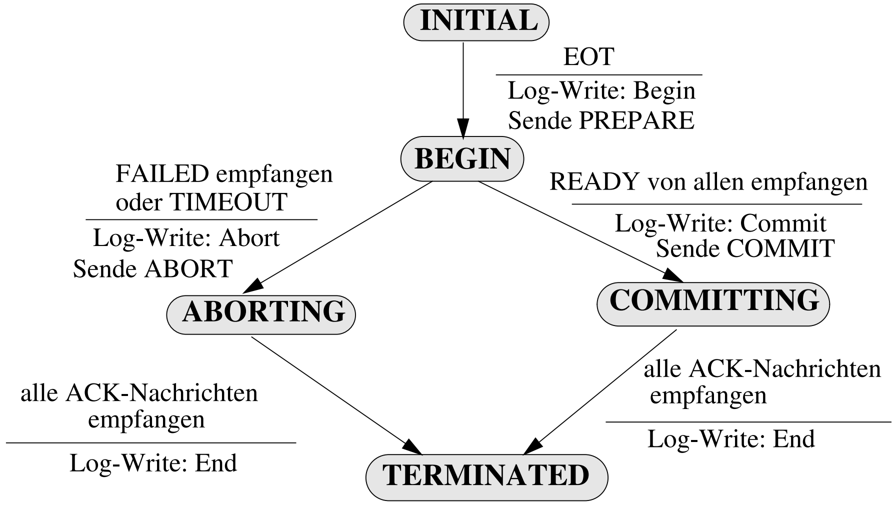
Zentralisiertes 2-Phasen-Commit



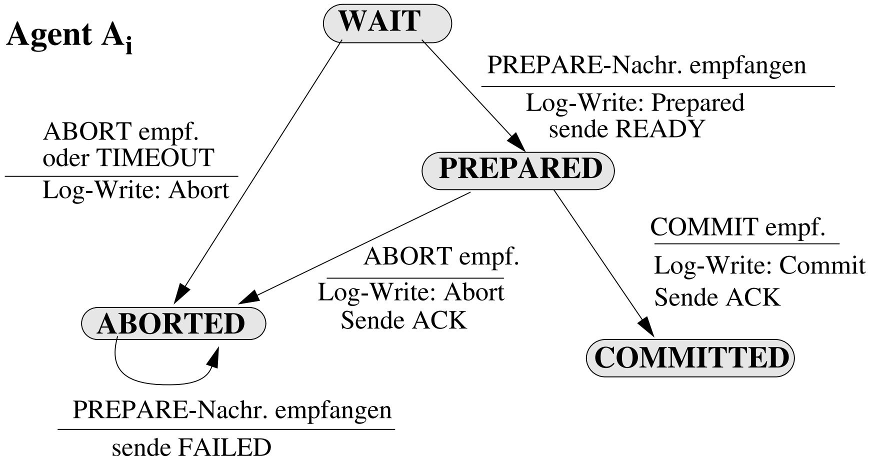
- **ABORT-Nachrichten** gehen nur an Teil-TA, die nicht mit FAILED gestimmt haben
- **Problem bei 2PC:**
  Koordinatorausfall ➥ lange Blockierung möglich
- **Basisverfahren:**
  - 4 (N-1) Nachrichten (N = Anzahl der beteiligten Knoten)
  - 2 + 2 (N-1) Log-Ausgaben (forced log writes)
- **Optimierung für lesende Sub-TA (Anzahl M)**
  - 4 (N-1) - 2M Nachrichten für M < N, 2 (N-1) für M=N-1
  - 2 + 2 (N-1) - M Log-Ausgaben

2PC: Zustandsübergänge

Koordinator C



Agent A_i



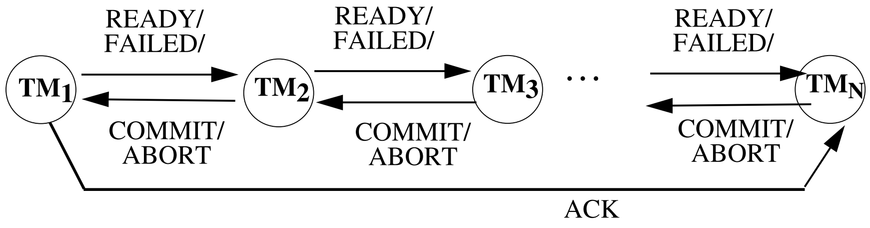
2PC: Fehlerbehandlung

- **Timeout-Bedingungen für Koordinator:**
  - WAIT ➥ setze TA zurück; verschicke ABORT-Nachr.
  - ABORTING, COMMITTING ➥ vermerke Agenten, für die ACK noch aussteht
- **Timeout-Bedingungen für Agenten:**
  - WAIT ➥ setze Teil-TA zurück (unilateral ABORT)
  - PREPARED ➥ erfrage TA-Ausgang bei Koordinator (bzw. anderen Rechnern)
- **Ausfall des Koordinatorknotens:**

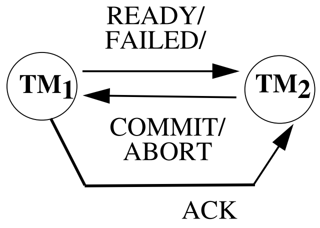
  Log-Zustand
  - TERMINATED:
    - UNDO bzw. REDO-Recovery, je nach TA-Ausgang
    - keine "offene" Teil-TA möglich
  - ABORTING:
    - UNDO-Recovery
    - ABORT-Nachricht an Rechner, von denen ACK noch aussteht
  - COMMITTING:
    - REDO-Recovery
    - COMMIT-Nachricht an Rechner, von denen ACK noch aussteht
  - Sonst: UNDO-Recovery
- **Rechnerausfall für Agenten:**

  Log-Zustand
  - COMMITTED: REDO-Recovery
  - ABORTED bzw. kein 2PC-Log-Satz vorhanden: UNDO-Recovery
  - PREPARED: Anfrage an Koordinator-Knoten, wie TA beendet wurde (Koordinator hält Information, da noch kein ACK erfolgte)

Lineares 2PC

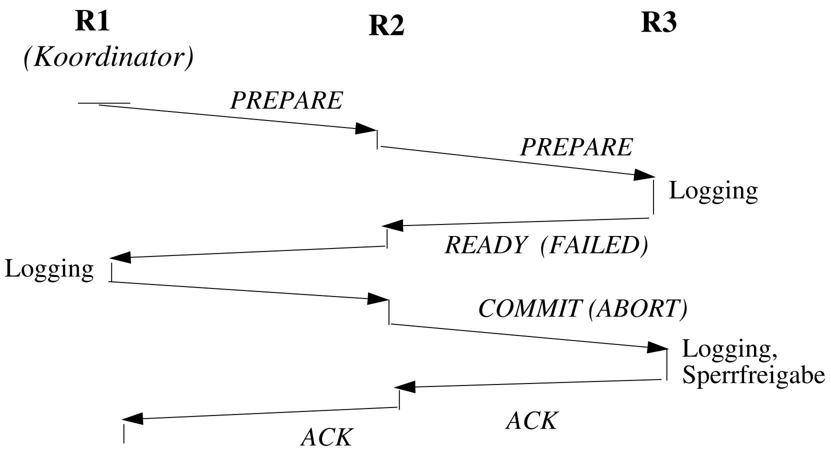


- **Sequentielle Commit-Behandlung,** dafür Halbierung der Nachrichtenanzahl (2N-1)
- **Transfer der Commit-Koordinierung:** Koordinatorrolle geht auf letzten Agenten über (*"Last Agent"-Optimierung*)
- **Besonders vorteilhaft für N=2** (3 Nachrichten)



Hierarchisches 2PC

- **Allgemeineres Ausführungsmodell** mit beliebiger Schachtelungstiefe

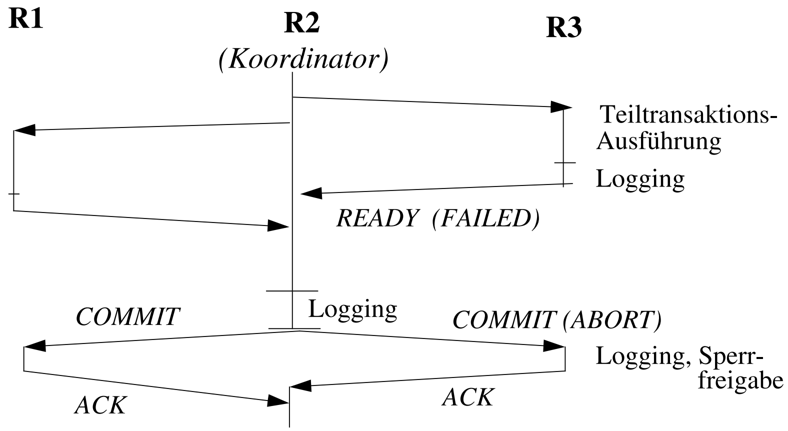


- **Kosten**
  - Anzahl der Nachrichten und Log-Ausgaben wie im zentralisierten Fall
  - **aber:** Antwortzeit steigt mit Schachtelungstiefe
  - Ausfall eines Knotens mit Koordinatorrolle: **Blockierung** möglich
- **Bekanntester Vertreter: *Presumed-Abort-Protokoll*[1]**
  - Optimierung für lesende Teil-TA:
    - kein Logging, Sperrfreigabe in Phase 1
    - Kommunikation für zweite Phase wird umgangen
  - für ABORT:
    - keine ACK-Nachrichten
    - kein synchrones Logging
    - wenn keine Angabe im Log vorhanden, wird per Default ABORT angenommen

1 Basic 2PC wird auch als Presumed-Nothing-Protokoll bezeichnet. Optimierungen dieses Protokolls sind Presumed-Abort und Presumed-Commit, bei denen eine Log-Ausgabe eingespart werden kann

1-Phasen-Commit

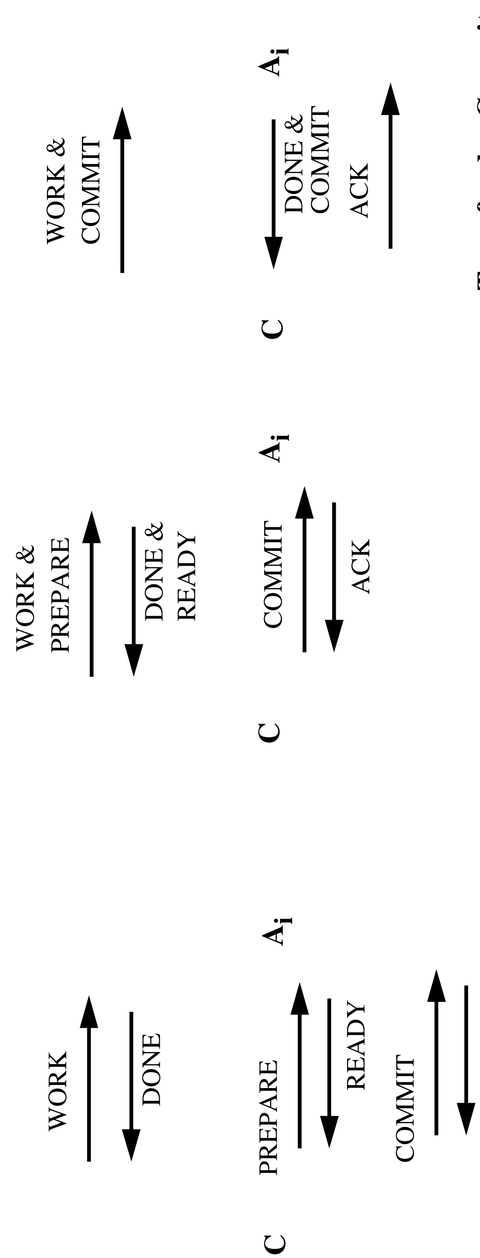
- **Teil-TA sichern ihre Änderungen** bereits vor Rückgabe der Ergebnisse an Primär-TA, d. h., sie gehen in den Prepared-Zustand
- Nach lokalem Commit am Koordinator-Knoten steht Erfolg der TA fest



- **Einsparung** von 2 Nachrichten pro Agent: **2 (N-1) Nachrichten**
- **Besonders vorteilhaft für kurze (verteilte) TA**
- **Nachteile:**
  - Teil-TA geben Recht zum TA-Abbruch frühzeitig auf (➥ lange Phase der Unsicherheit über TA-Ausgang)
  - starke Abhängigkeit vom Koordinator-Knoten
  - mehrfaches Logging (PREPARED) pro TA und Knoten möglich

1-Phasen-Commit (2)

Für N=2 kann weitere Nachricht durch Transfer der Commit-Koordinierung eingespart werden



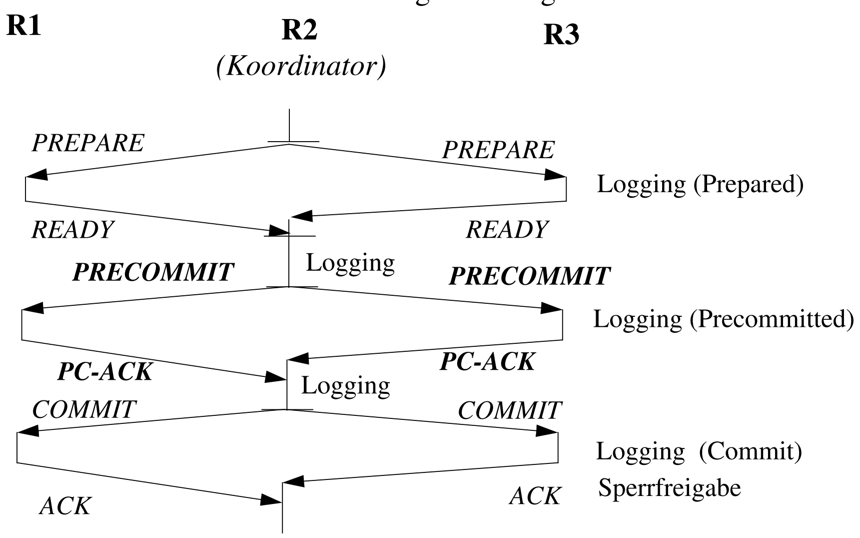
Zwei-Phasen-Commit

Ein-Phasen-Commit

Transfer der Commit-Koordinierung + Ein-Phasen-Commit

3-Phasen-Commit

- **Nicht-blockierendes Verfahren** unter den Annahmen
  - keine Netzwerkpartitionierung
  - höchstens K < N Rechner fallen gleichzeitig aus



- **Erste Phase wie in 2PC**
- **Neue Zwischenphase** (alle haben mit READY gestimmt)
  - Koordinator geht zunächst in Zustand PRECOMMIT und teilt diese Entscheidung allen Teil-TA mit
  - nach Eingang von K Quittungen (PC-ACK) erfolgt Commit-Entscheidung
- **Koordinatorausfall**
  - Wahl eines neuen Koordinators
  - Falls Zustand PRECOMMITTED erkannt wird, wird Commit-Protokoll von neuem Koordinator fortgeführt (mit Verschicken von Precommit-Nachrichten), sonst Abbruch
- **ABORT-Behandlung wie im 2PC**

Nachrichtenbedarf verteilter Commit-Protokolle

■ **N: Anzahl beteiligter Knoten, M: Anzahl lesender Sub-TA**

	allgemein	Beispiel 1 (N=2, M=0)	Beispiel 2 (N=10, M=5)
1-Phasen-Commit	2*(N-1)	2	18
lineares 2PC	2*N-1	3	19
zentralisiertes/ hierarchisches 2PC	4*(N-1)-2M	4	26
3-Phasen-Commit	6*(N-1)-4M	6	34

Commit: Kostenbetrachtungen

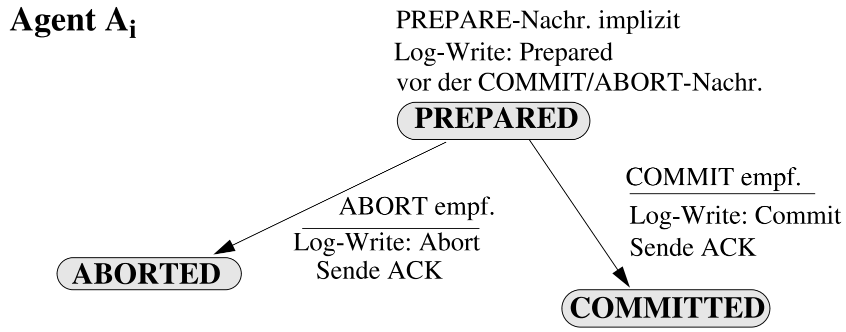
■ **vollständiges 2PC-Protokoll**

- 4 (N-1) Nachrichten (N = Anzahl der beteiligten Knoten)
- 2 + 2 (N-1) Log-Ausgaben

■ **Weglassen der expliziten Ack-Nachricht**

- A_i fragt ggf. nach; "unendlich langes Gedächtnis" von C
- 3 (N-1) Nachrichten
- 2 + 2 (N-1) Log-Ausgaben

■ **A_i geht nach jedem Auftrag in den PREPARED-Zustand**
(oder nach dem letzten Auftrag)



- 2 (N-1) Nachrichten
- 1 + (N-1)K Log-Ausgaben (durchschnittlich K Aufträge pro Agent)
- 1 + 2 (N-1) Log-Ausgaben (bei Work&Prepare als letzten Aufruf)

■ **Spartanisches Protokoll**

- A_i geht nach jedem Auftrag (nach dem letzten Auftrag) in den PREPARED-Zustand; Weglassen der expliziten Ack-Nachricht
- 1 (N-1) Nachrichten
- 1 + (N-1)K Log-Ausgaben
- 1 + 2 (N-1) Log-Ausgaben (bei Work&Prepare als letzten Aufruf)

➥ Log-Aufwand bleibt gleich (oder erhöht sich drastisch) !

Integritätssicherung

■ **Einhaltung definierter Integritätsbedingungen**

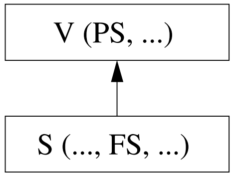
- anwendungsspezifische Bedingungen
  (Wertebereiche, CHECK-Constraints etc.)
- modellinhärente Bedingungen
  (Eindeutigkeit von Schlüsselkandidaten, referentielle Integrität)
- Gewährleistung der Konsistenzerhaltung von Replikaten

➥ VDBS bietet Sicht auf **eine logisch zentralisierte DB**!

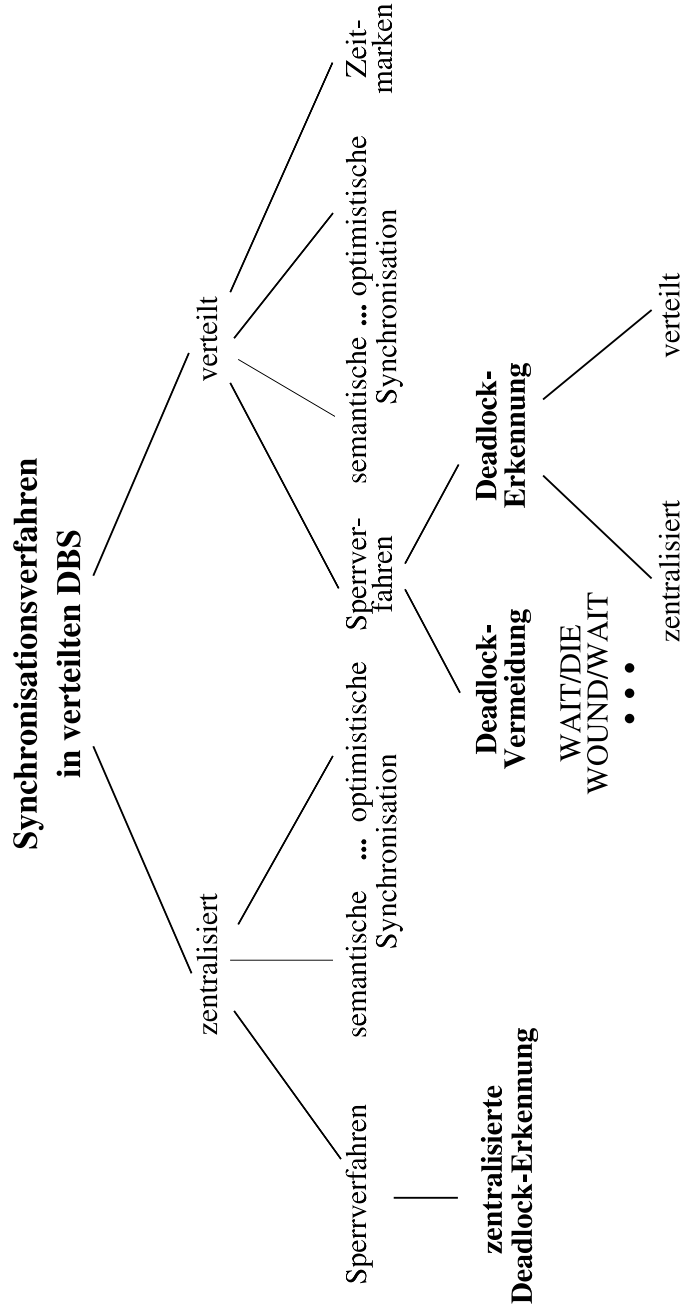
■ **Überwachung von Integritätsbedingungen in VDBS**
konzeptionell weitgehend wie im zentralen Fall

- jedoch ggf. erheblicher zusätzlicher Kommunikations-Overhead, Verzögerung der Überprüfbarkeit durch Knoten- oder Verbindungsausfall usw.
- Nutzung von Triggern mit verteilten Aktionen
- Bsp.: Eindeutigkeitsprüfung des Primärschlüssels bei Einfügen eines neuen Tupels bei horizontaler Datenverteilung
- Überprüfung verzögerter Integritätsbedingungen in/vor der ersten Commit-Phase (bei 2PC)

■ **Referentielle Integrität**



- **unproblematisch:**
  Löschen eines S-Satzes sowie Einfügen eines V-Satzes
- **Einfügen eines S-Satzes** bzw. FS-Änderung:
  neuer FS-Wert muß definiert sein
- **Löschen eines V-Satzes**, PS-Änderung:
  verschiedene Reaktionsmöglichkeiten
  (RESTRICT, CASCADE, SET NULL ...)



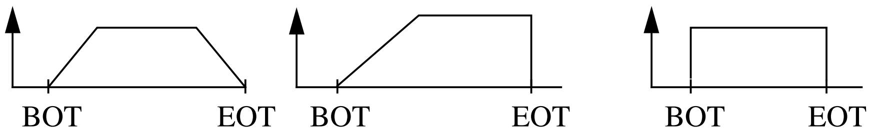
Synchronisation

- **Mehrbenutzerbetrieb führt ohne Synchronisation zu Anomalien:**
  Lost Updates, Non-repeatable Reads, Phantome, Dirty Reads

- **Korrektheitskriterium der globalen Serialisierbarkeit:**
  verteilte und gleichzeitige Ausführung mehrerer TA ist äquivalent zu wenigstens einer seriellen Ausführung derselben TA

- **Zwei-Phasen-Sperrprotokolle (2PL) garantieren Serialisierbarkeit**
  - Vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
  - Gesetzte Sperren anderer TA sind zu beachten
  - Eine TA darf nicht mehrere Sperren für ein Objekt anfordern
  - ***Zweiphasigkeit:*** Anfordern von Sperren erfolgt in einer *Wachstumsphase,* Freigabe der Sperren in *Schrumpfungsphase*. Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden
  - Spätestens bei EOT sind alle Sperren freizugeben

#Sperren



strikt zweiphasig **preclaiming**

- **Einfacher Ansatz: RUX - Sperrverfahren**

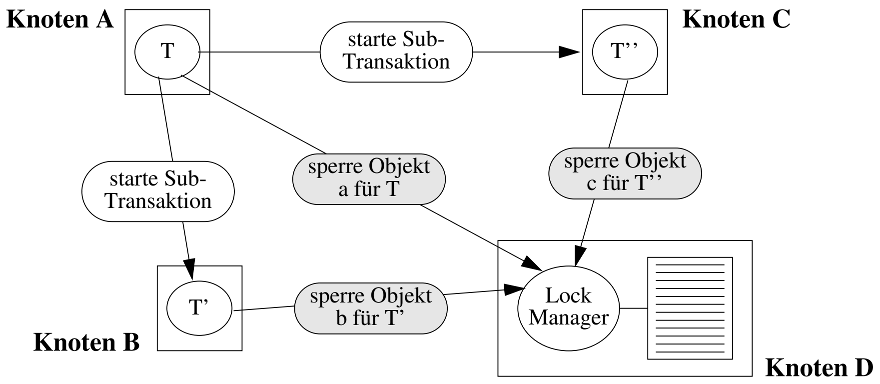
		aktueller Sperrmodus			
		NL	R	U	X
angeforderter Sperrmodus	R	+	+	-	-
	U	+	+	-	-
	X	+	-	-	-

NL: no lock, R: read lock, U: update lock (conversion), X: eXclusive lock

Synchronisation in VDBS

- **Sperrverfahren**
  - Eine nicht-serialisierbare Ausführungsreihenfolge wird dadurch erkannt bzw. verhindert, daß ggf. mehrere TA zyklisch auf die Freigabe einer Sperre warten, die von einer anderen TA gehalten wird.
  - Die äquivalente serielle Ausführungsreihenfolge einer TA wird beim 2PL durch den Zeitpunkt definiert, in dem alle ihre Sperren angefordert sind.
  - Typischerweise werden physische Objekte (Seiten, Sätze usw.) gesperrt. Implementierung des Sperrverfahrens erfolgt mittels einer Sperrtabelle.
  - Im Prinzip sind zwei Vorgehensweisen möglich:
    - zentralisiertes Sperren mittels einer Sperrtabelle
    - verteiltes Sperren mittels dezentraler Sperrtabellen

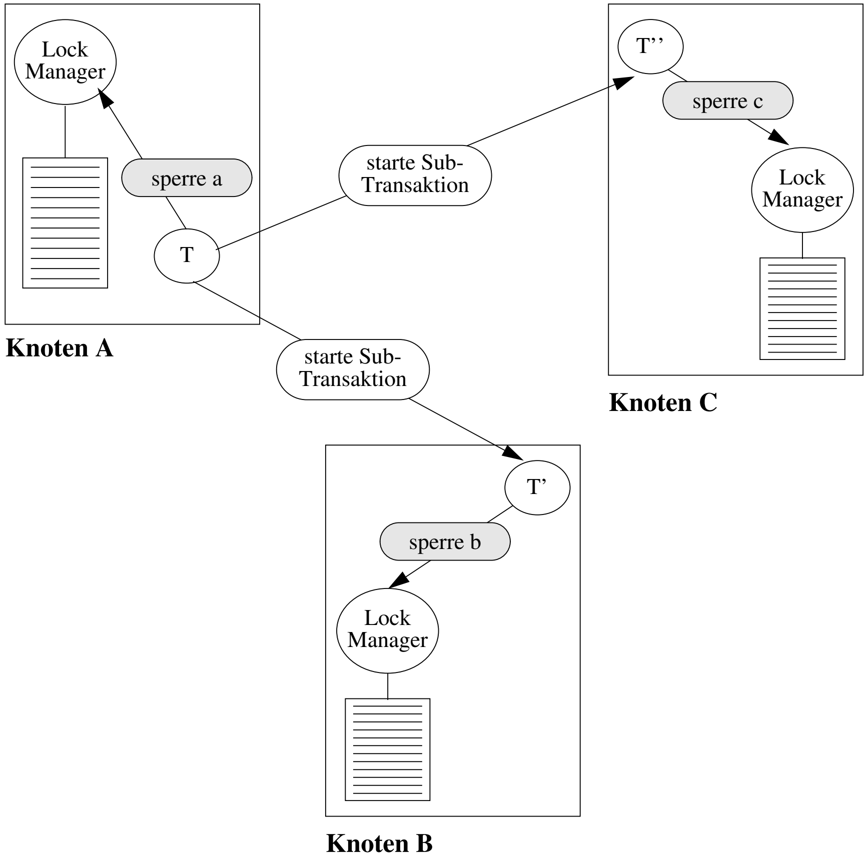
- **Zentralisierte Sperrverfahren**
  - erfordern einen hohen Kommunikationsaufwand
  - können Ursache für mangelnde Verfügbarkeit sein
  - implizieren eine eingeschränkte Knotenautonomie



➥ Wie erfolgt ein Scan in Knoten A?

Synchronisation in VDBS (2)

- **Verteilte Sperrverfahren**
  - Jeder Rechner verwaltet Sperren für seine lokalen Daten
  - Sperranforderungen können ohne externe Nachrichten abgewickelt werden
  - Sperrfreigabe erfolgt innerhalb des Commit-Protokolls
  - Sperren mit dezentralen Sperrtabellen stellen einen weitverbreiteten Ansatz für VDBS dar



➥ Synchronisationskosten (mit Ausnahme der Deadlock-Behandlung) sind mit denen in zentralisierten DBS vergleichbar

Semantische Synchronisationsverfahren

- **Erhöhung der Parallelität** durch Einführung kommutativer („semantischer") DB-Operationen als Einheit für die Synchronisation
  - Synchronisation erfolgt auf abstrakterer Ebene (Objektebene)
  - Realisierung muß auf niedrigerer Ebene Korrektheit gewährleisten (z. B. durch Escrow-Verfahren)

- **Beispiel**

  Zwei Kontobuchungen auf Konten K_1 und K_2 durch die TA T_1 und T_2, die kommutative Operationen „erhöhe um x" und „vermindere um x" verwenden, könnten z.B. in folgenden Reihenfolgen ausgeführt werden:

Schedule S_1	Schedule S_2
1. erhöhe (T_1,K_1,x_1)	1. erhöhe (T_1,K_1,x_1)
2. vermindere (T_1,K_2,x_1)	2. erhöhe (T_2,K_1,x_2)
3. erhöhe (T_2,K_1,x_2)	3. vermindere (T_2,K_2,x_2)
4. vermindere (T_2,K_2,x_2)	4. vermindere (T_1,K_2,x_1)

➥ Schedule S_1 wäre auch mit rein syntaktischen Verfahren (r/w) erzeugbar:

$$<\underbrace{r_1[K_1]\ w_1[K_1]}_{\text{erhöhe}(T_1,K_1,x_1)}\quad \underbrace{r_1[K_2]\ w_1[K_2]}_{\text{vermindere}(T_1,K_2,x_1)}\quad \underbrace{r_2K_1]\ w_2[K_1]}_{\text{erhöhe}(T_2,K_1,x_2)}\quad \underbrace{r_2[K_2]\ w_2[K_2]}_{\text{vermindere}(T_2,K_2,x_2)}>.$$

Schedule S_2 wäre hingegen mit rein syntaktisch arbeitenden Verfahren nicht erzeugbar.

➥ Welche Aspekte sind im verteilten Fall relevant?

Optimistische Synchronisationsverfahren

- **Drei Phasen**
  - Lesephase:
    - ungeschütztes Lesen
    - Vorbereitung von Änderungen auf lokalen Kopien
  - Validierungsphase:
    - kritischer Abschnitt
    - Prüfung, ob „dirty reads“ bzw. „veraltetes Lesen“ vorgekommen ist
    - Austritt[1] aus Validierungsphase definiert die **äquivalente serielle Ausführungsreihenfolge**
  - Schreibphase:
    - Einbringen der Änderungen in die Datenbank.

- **TA-Abschluß**
  - Eintritt in die Validierungsphase ist zwischen den beteiligten Teil-TA zu koordinieren

➥ Wie kann man die Validierung im verteilten Fall durchführen?

  - Auch die Schreibphase muß im verteilten Fall mittels 2PC koordiniert werden, sonst Gefahr nicht-serialisierbarer globaler Abläufe (Schedules)

➥ **Teil-TA müssen lokal in derselben Reihenfolge in Bezug auf andere TA validieren und ihre Änderungen in die DB einbringen**

1 Da die Validierungsphase jeweils exklusiv durchlaufen wird, kann auch der Eintrittszeitpunkt oder ein beliebigen Zeitpunkt dazwischen gewählt werden.

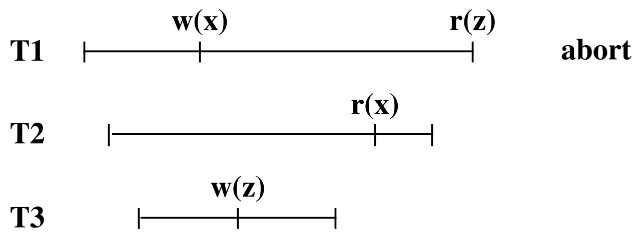
Zeitmarkenverfahren

- **Grundsätzliche Idee**
  - TA bekommt bei BOT einen systemweit eindeutigen Zeitstempel (Transaktionszeitmarke)
  - TA hinterläßt den Wert ihres Zeitstempels (als Lese- oder Schreibstempel RTS bzw. WTS) bei jedem Objekt, auf das sie zugreift (Objektzeitmarken)
  - Prüfung der Serialisierbarkeit ist sehr einfach (Zeitmarkenvergleich)

- TA T wird zurückgesetzt, falls bei

  ***Lesezugriff* auf Objekt O**
  ts (T) < WTS (O)
  oder bei *Schreibzugriff*
  ts (T) < max (RTS (O), WTS (O))

  gilt



- **Konflikt-Operationen** müssen in Zeitmarkenreihenfolge ausgeführt werden

  ➥ **Zurücksetzen "später" Operationen**

Zeitmarkenverfahren (2)

■ **Eigenschaften**

- Serialisierungsreihenfolge durch BOT-Zeitmarken festgelegt
- keine Deadlocks möglich
- **Globale Synchronisation ohne zusätzliche Nachrichten:** lokale Prüfung der Serialisierbarkeit direkt am Objekt O

➥ Verfahren ist zugeschnitten auf verteilte Synchronisation, **aber ...**

■ **Durch BOT-Zeitmarken festgelegte Serialisierungsreihenfolge führt meist zu vielen Rücksetzungen**

- hohe Rücksetzwahrscheinlichkeit, v.a. für lange TA
- Gefahr ständigen Scheiterns (Starvation)

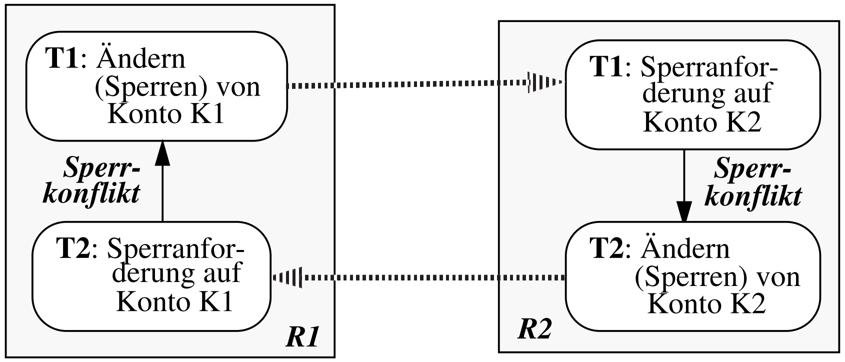
■ **Z**ur Verhinderung von **"dirty read"** sind Änderungen vor parallelen Transaktionen zu verbergen

➥ Verzögerung aller Zugriffe bis EOT (≡ X-Sperre)

➥ **Zeitmarkenverfahren versprechen kaum Parallelitätsgewinn gegenüber Sperrverfahren**

Deadlock-Behandlung

■ **Deadlock:**
zyklische Wartebeziehung zwischen zwei oder mehr TA



■ **Lösungsmöglichkeiten**

1. **Timeout-Verfahren**
   - nach Ablauf einer maximalen Wartezeit auf eine Sperre (Timeout) wird die Transaktion zurückgesetzt
   - problematische Bestimmung des Timeout-Wertes

2. **Deadlock-Verhütung (Prevention)**
   - *keine Laufzeitunterstützung* zur Deadlock-Behandlung erforderlich
   - Bsp.: Preclaiming (in DBS i. allg. nicht praktikabel)

3. **Deadlock-Vermeidung (Avoidance)**
   - potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden
     ➥*Laufzeitunterstützung nötig*

4. **Deadlock-Erkennung (Detection)**

Deadlock-Vermeidung (1)

- **Zuweisung einer eindeutigen *TA-Zeitmarke* bei BOT**

  z. B.: ts(T1) = 1; ts(T2) = 2

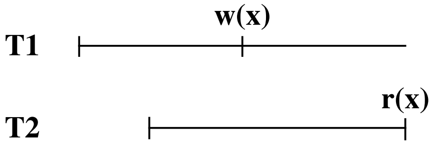
- Im Konfliktfall darf nur ältere (bzw. jüngere) TA warten

➥ **kein Zyklus möglich**

- **WAIT / DIE-Verfahren**
  - Anfordernde Transaktion darf auf die Freigabe der Sperre nur warten, falls sie älter als der Sperrbesitzer ist
  - Bei Abbruch (dann ist sie die jüngere TA) wird Ti mit der alten Zeitmarke erneut gestartet

Ti fordert Sperre an, Konflikt mit Tj:

IF ts (Ti) < ts (Tj)
THEN WAIT (Ti)
ELSE ABORT (Ti) "DIE"



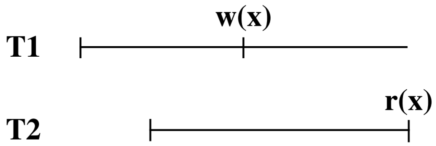
➥ Ältere TA warten auf jüngere

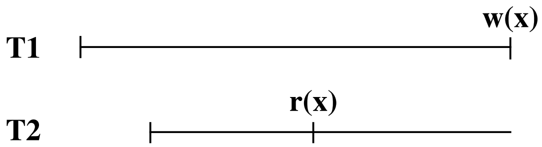
Deadlock-Vermeidung (2)

- **WOUND / WAIT-Verfahren**
  - Sperrbesitzer wird „verwundet”, wenn er jünger als anfordernde TA ist
  - Er wird dann zurückgesetzt und neu gestartet

Ti fordert Sperre an, Konflikt mit Tj:

IF ts (Ti) < ts (Tj)
THEN ABORT (Tj) "WOUND"
ELSE WAIT (Ti)





- Jüngere TA warten auf ältere
- preemptiver Ansatz

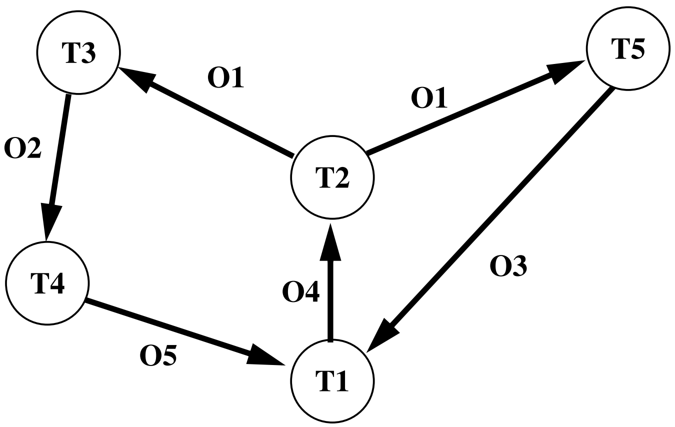
- **Verbesserung für Wait/Die und Wound/Wait**

  statt BOT-Zeitmarke Zuweisung der TA-Zeitmarke erst bei erstem Sperrkonflikt ("dynamische Zeitmarken")

➥ **Erster Sperrkonflikt kann stets ohne Rücksetzung behandelt werden**

Deadlock-Erkennung

- **Explizites Führen eines Wartegraphen** (WfG: wait-for graph) und Zyklensuche zur Erkennung von Verklemmungen

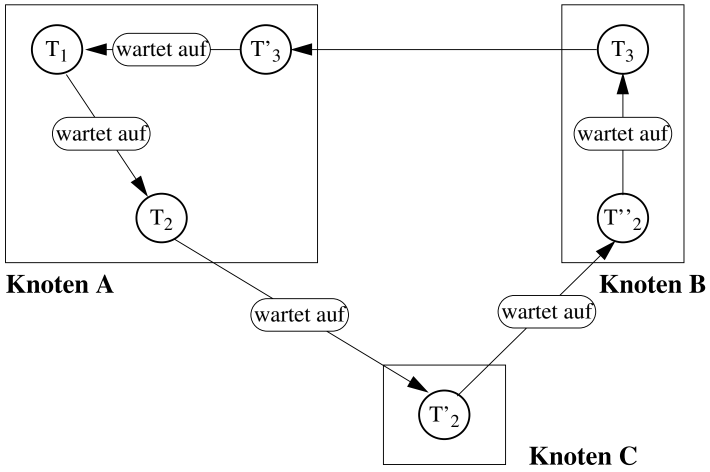


- Deadlock-Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA, z. B. von
  - Verursacher
  - "billigster" TA
    - geringste CPU-Zeit
    - geringste Anzahl von Änderungen

- **Zyklensuche entweder**
  - bei jedem Sperrkonflikt oder
  - verzögert (z.B. über Timeout gesteuert)

Deadlock-Erkennung in VDBS

- **Beispiel für globale Verklemmung**



- **Explizite Analyse von Wartesituationen**
  - aufwendig und schwierig
  - Nachrichtenaustausch zur Erstellung des Wartegraphen
  - zentralisierter oder verteilter Wartegraph

- **Zentralisierte Deadlock-Erkennung**
  - hoher Kommunikationsaufwand (v.a. bei Ortsverteilung)
  - single-point-of-failure

- **Verteilte Deadlock-Erkennung**
  - **korrektes Verfahren** schwierig zu realisieren:
    - Nachrichtenverzögerungen
    - Empfangs- ≠ Sendereihenfolge
    - Aborts / Rechnerausfälle
  - oftmals
    - doppelte Erkennung von Deadlocks
    - "falsche" Deadlocks

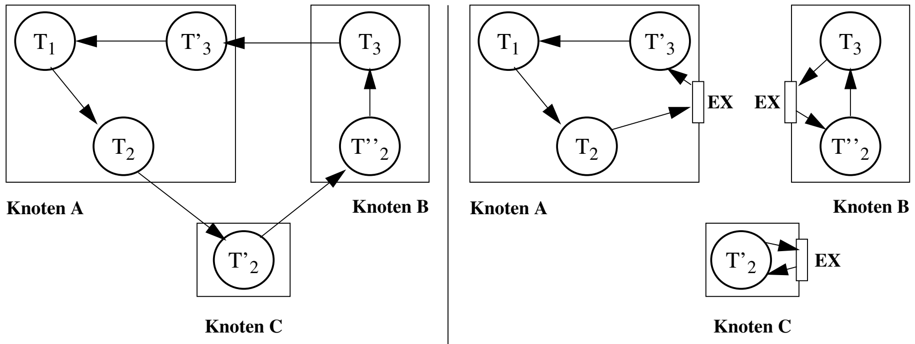
Deadlock-Erkennung in VDBS (2)

- **Verteilte Deadlock-Erkennung**
  - Im Prinzip nehmen **alle** Knoten am Suchprozeß teil
  - Wait-for-Information wird oft weiter vergröbert, um Kommunikationsaufwand zu sparen, z. B.

    $T_{global,1} \rightarrow T_{lokal} \rightarrow T_{global,2}$ wird reduziert zu: $T_{global,1} \rightarrow T_{global,2}$

  - Dadurch jedoch Gefahr des Erkennens von Pseudo-Verklemmungen (➥ unnötige Zurücksetzungen)

- **Mögliche Vorgehensweise**
  - Jeder lokale WfG erhält genau **einen** externen Ein-/Ausgang (➥ EX)
  - Externe Wait-for-Beziehungen werden bzgl. lokalem EX formuliert



„Echter“ globaler WfG | **Lokale WfGs**

  - Kritisch sind Zyklen der Form EX → ... → EX, da sie auf einen möglichen globalen Deadlock hinweisen
  - Information über solche Zyklen werden an andere Knoten verschickt

    Beispiel:
    $EX \rightarrow T_1 \rightarrow T_7 \rightarrow T_5 \rightarrow EX$ wird zum String: „EX, 1, 7, 5“

Verteilte Deadlock-Erkennung: Verfahren von Obermarck[1]

- **Ein „Deadlock Detector” pro Rechner,** der periodisch lokalen Wartegraphen auf Zyklen durchsucht

- **Einsatz spezieller Knoten „EXTERNAL”** im Wartegraph
  - Darstellung von Wartebeziehungen zu Teil-TA auf anderen Rechnern
  - Zyklus mit EX-Knoten kennzeichnet potentiellen globalen Deadlock
  - Weitergabe der Zyklusinformation an andere Rechner, um globalen Deadlock ggf. zu erkennen

- **Kooperation mit anderen Rechnern**
  1. Empfange Deadlock-Information anderer Rechner
  2. Erweitere damit lokalen Wartegraphen
  3. Löse lokale/vollständige Zyklen durch Bestimmung und Rücksetzung eines "Opfers" auf
  4. Für globale Zyklen sende lineare Darstellung

     EX -> T_1 -> T_2 -> ... -> T_n -> EX

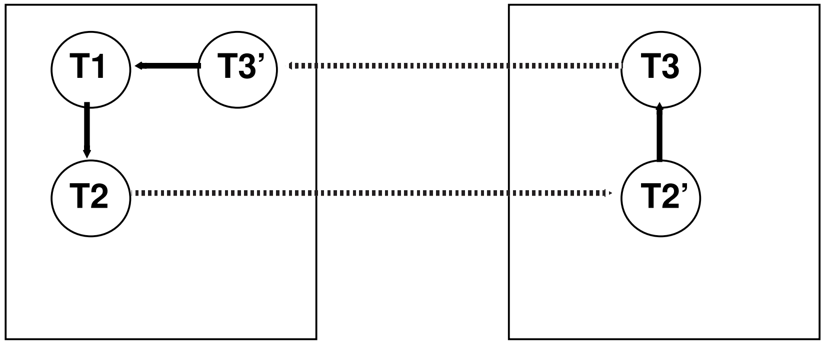
     an Rechner, auf den T_n wartet (falls $T_1 > T_n$)

- **max. N (N-1) / 2 Nachrichten** zur Erkennung eines globalen Deadlocks (bei N beteiligten Rechnern)

- **Erkennung falscher Deadlocks möglich**

1 R. Obermarck: *Deadlock detection for all resource classes.* ACM Trans. on Database Systems 7 (2), 187-208, 1982

Beispiel



Algorithmus: „Distributed Deadlock Detection" (DDD)

1. Konstruiere den lokalen WfG (wie beschrieben). Gehe zu Schritt 5
2. Falls Strings und Deadlock-Information von anderen Knoten eintreffen, so werden sie im lokalen WfG wie folgt berücksichtigt:
   a. Alle Knoten und Kanten von Deadlock-Opfern werden aus dem WfG entfernt.
   b. Alle Strings, die ein bereits bekanntes Deadlock-Opfer enthalten, werden ignoriert.
   c. Noch nicht bekannte TA werden eingefügt.
   d. Für die im String enthaltenen Nachfolger-TA werden im WfG ggf. Pfeile hinzugefügt.
3. Füge für jede TA T im WfG, auf deren Nachricht eine nicht-lokale TA wartet, dem WfG eine EX $\rightarrow$ T - Kante hinzu (sofern noch nicht vorhanden).
4. Füge für jede TA T' im WfG, die auf eine Nachricht einer nicht-lokalen TA wartet, dem WfG eine T' $\rightarrow$ EX - Kante hinzu (sofern noch nicht vorhanden).
5. Analysiere den WfG und erstelle eine Liste aller **elementaren Zyklen** (im folgenden kurz **Zyklenliste** genannt).

Algorithmus: „Distributed Deadlock Detection“ (2)

Die nachfolgenden Schritte beziehen sich nur noch auf die Zyklenliste:

6. Ermittle alle Zyklen in der Zyklenliste, die nicht den Knoten EX enthalten, und wähle daraus jeweils eine TA T_V als Deadlock-Opfer aus.

7. Entferne T_V (falls vorhanden) aus dem WfG sowie alle Kanten, die von T_V ausgehen oder in T_V einmünden. Entferne alle Strings (und die darauf basierenden Zyklen), die T_V enthalten.

8. Informiere (falls T_V eine globale TA war) die anderen Knoten über das Zurücksetzen von T_V.

In der Zyklenliste sind jetzt nur noch Zyklen mit EX enthalten:

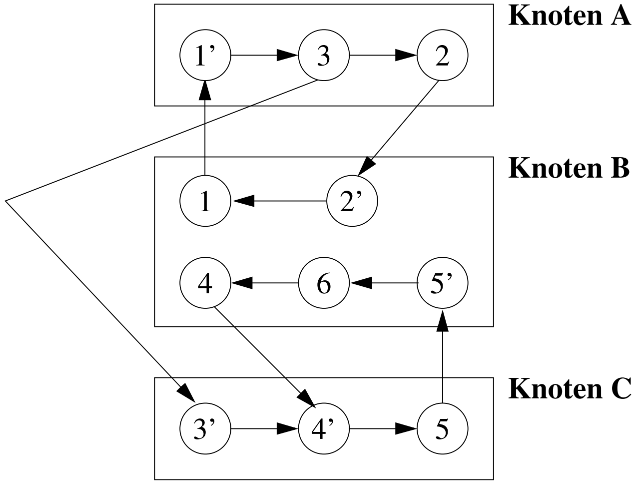
9. Ermittle die Zyklen EX $\rightarrow T_i \rightarrow ... \rightarrow T_j \rightarrow$ EX in der Zyklenliste, für die gilt: Zykluslänge > 2 und TAID(T_i) > TAID(T_j).[1]
   a. Transformiere diese Zyklen in einen String (wie beschrieben).
   b. Sende den String zu dem Knoten, auf dessen Nachricht die letzte TA im String wartet.

Gehe zu Schritt 2.

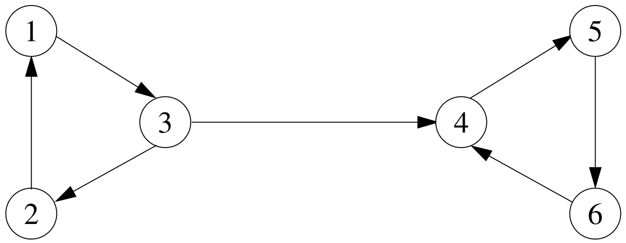
1 Es werden also keine Strings der Form „EX, 1,3,4,...“ erzeugt, um Mehrfach-Übertragungen zu vermeiden. Bei der Berechnung der Zykluslänge werden die EX-Knoten jeweils nur einfach gezählt.

DDD-Algorithmus - Beispiel

■ **Gegeben ist folgende globale „Wait-for“-Situation**

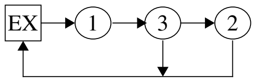
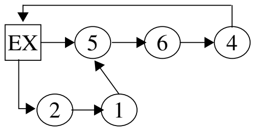
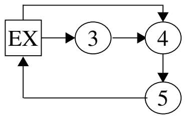
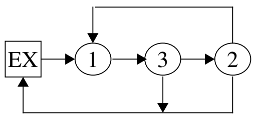
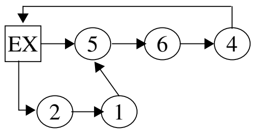
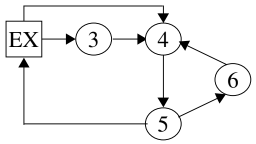


■ **WfG bei zentralisierter Deadlock-Erkennung**



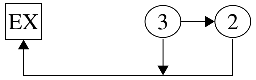
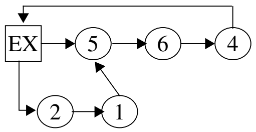
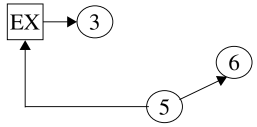
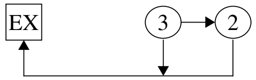
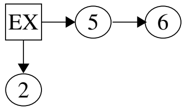
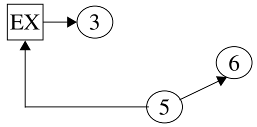
Elementare Zyklen: $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$
$4 \rightarrow 5 \rightarrow 6 \rightarrow 4$

DDD-Algorithmus - Beispiel (2)

It.	Knoten A	Knoten B	Knoten C
1.	Eingabe: Deadlock-Suche!	Eingabe: Deadlock-Suche!	Eingabe: Deadlock-Suche!
			
	Analyse: EX, 1, 3, EX EX, 1, 3, 2, EX	Analyse: EX, 5, 6, 4, EX EX, 2, 1, EX	Analyse: EX, 3, 4, 5, EX EX, 4, 5, EX
	Ausgabe: --[1]	Ausgabe: EX, 5, 6, 4 ⇒ C EX, 2, 1 ⇒ A	Ausgabe: --
2.	Eingabe: B: EX, 2, 1	Eingabe: --	Eingabe: B: EX, 5, 6, 4
			
	Analyse: EX, 1, 3, EX EX, 1, 3, 2, EX (1, 3, 2, 1) ⇒ Zyklus $\boxed{\text{Opfer = 1}}$		Analyse: EX, 4, 5, EX EX, 3, 4, 5, EX (4, 5, 6, 4) ⇒ Zyklus $\boxed{\text{Opfer = 4}}$

1. Wenn $TAID_{first} < TAID_{last}$, wird EX ... nicht verschickt

DDD-Algorithmus - Beispiel (3)

It.	Knoten A	Knoten B	Knoten C
			
	Ausgabe: Opfer = 1 ⇒ B,C		Ausgabe: Opfer 4 ⇒ A,B
3.	Eingabe: C: Opfer = 4	Eingabe: C: Opfer = 4 A: Opfer = 1	Eingabe: A: Opfer = 1
			
	Analyse: --	Analyse: --	Analyse: --

Logging und Recovery

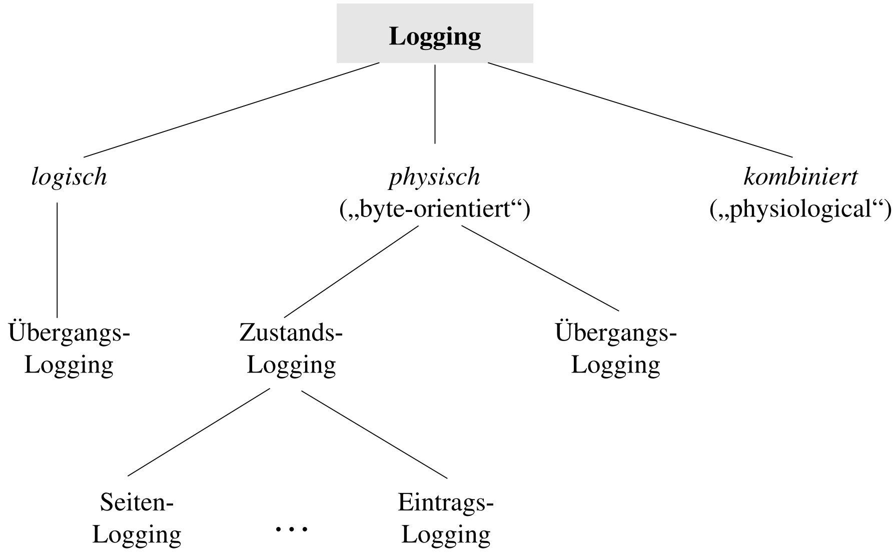
- **Murphy:**
  What can go wrong, will go wrong
- **Fehlermodell**
  - Transaktionsfehler
  - Systemfehler
    - i. allg. kein Gesamtausfall
    - partielle Fehler (Rechner, Verbindungen, ...)
  - Gerätefehler
  - Katastrophen
- **Was wird benötigt?**
  - Logging:
    Sammeln redundanter Informationen während des Normalbetriebs
  - Sicherungspunkte:
    Maßnahmen zur Begrenzung des Redo-Aufwandes nach Systemfehlern
  - Ersetzungs- und Einbringstrategien für DB-Änderungen
  - Recovery-Verfahren (bezogen auf das Fehlermodell)

➥ Verfahren sind für den zentralisierten Fall bekannt!

- **Annahmen**
  (Unter welchen Voraussetzungen funktioniert die Wiederherstellung der Daten?)
  - quasi-stabiler Speicher
  - fehlerfreier DBVS-Code
  - fehlerfreie Log-Daten
  - Unabhängigkeit der Fehler

Logging und Recovery (2)

- **Logging-Verfahren**



➥ Welche Verfahren eignen sich für den verteilten Fall?

- **Was ist zu protokollieren?**
  - wie im zentralisierten Fall (BOT, EOT, Undo/Redo-Information für Aktualisierungsoperationen, Sicherungspunkte ...)
  - zusätzlich: Informationen
    - zur TA-Struktur: Primär-TA, initiierte Teil-TA
    - über erreichte Zustände (z. B. PREPARED im 2PC-Protokoll)
    - über gehaltene Sperren
    - über Cursor-Positionen bei transaktionsinternen Rücksetzpunkten usw.

Logging und Recovery (3)

- **Arten von Sicherungspunkten**
  - direkte Sicherungspunkte
    - transaktionskonsistent
    - aktionskonsistent
  - indirekte/unscharfe Sicherungspunkte
    (Fuzzy Checkpoints)

  ➥ Welche Verfahren eignen sich für den verteilten Fall?

- **Ersetzungs- und Einbringverfahren**
  - Atomic/Non-Atomic
  - Steal/NoSteal
  - Force/NoForce

  ➥ Gibt es Präferenzen für den verteilten Fall?

- **Mischformen bei der Realisierung**
  - in der Regel ein systemweites Verfahren bei homogenen VDBS
  - Prinzipiell sind knotenweise verschiedene Verfahrenskombinationen möglich, was aber losere Kopplungsformen impliziert
    (Heterogene VDBS, Föderierte DBS, Multi-DBS)

- **Durchführung der Crash-Recovery**
  - knotenweise Undo-/Redo-Recovery wie im zentralisierten Fall
  - Benachrichtigung von Primär- und Teil-TA bei Undo abhängiger TA
  - Selektive Recovery: Re-Aktivierung von Teil-TA in PREPARED-Zustand
    (vor Zulassung neuer TA!)
    - Sperren der Objekte
    - „I am alive again“-Nachricht an beteiligte Knoten

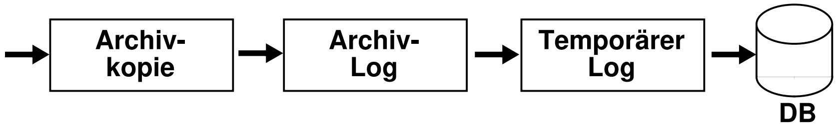
Geräte-Recovery[1]

- **Spiegelplatten/replizierte DB**
  - schnellste und einfachste Lösung bei Gerätefehlern
  - hohe Speicherkosten, Doppelfehler nicht auszuschließen
  - bei Katastrophen nicht ausreichend

- **Alternative: Archivkopie + Archiv-Log**
  - wird auch als Backup bezeichnet
  - sind längerfristig verfügbar zu halten (auf Band)

  ➥ Problem von Alterungsfehlern

  - Führen von Generationen der Archivkopie
  - Duplex-Logging für Archiv-Log
  - Prinzipielle Schritte der Geräte-Recovery
    (auch Langzeit-Recovery genannt)



- **Ableitung von Archivdaten**
  - Sammlung sehr großer Datenvolumina als nachgelagerter Prozeß
  - Archiv-Log kann offline aus temporärer Log-Datei abgeleitet werden
  - Erstellung von Archivkopien und Archiv-Log erfolgt **knotenorientiert**

1 „Don’t worry, be happy.“ (Bobby McFerrin)

Geräte-Recovery (2)

■ **Problemstellung: Erzeugung globaler Archivkopien** (Backups, Checkpoints) als Voraussetzung für globales Zurücksetzen im Fall katastrophaler Fehler, wie z.B.

- Brand, Hardware-Schaden
- Gerätefehler und Redo-Log nicht mehr lesbar
- . . .

■ **Knotenorientierte Erstellung der Archivkopie**

- offline (z. B. Incremental Dumping)
- online (parallel zum Änderungsbetrieb)
  - aktions-/transaktionskonsistente Archivkopie
  - in jedem Knoten: Black-/White- oder Copy-on-Update-Verfahren

➥ Wie wird **Abstimmung/Synchronisation zwischen Knoten** erreicht?

■ **Verschiedene Vorgehensweisen möglich**

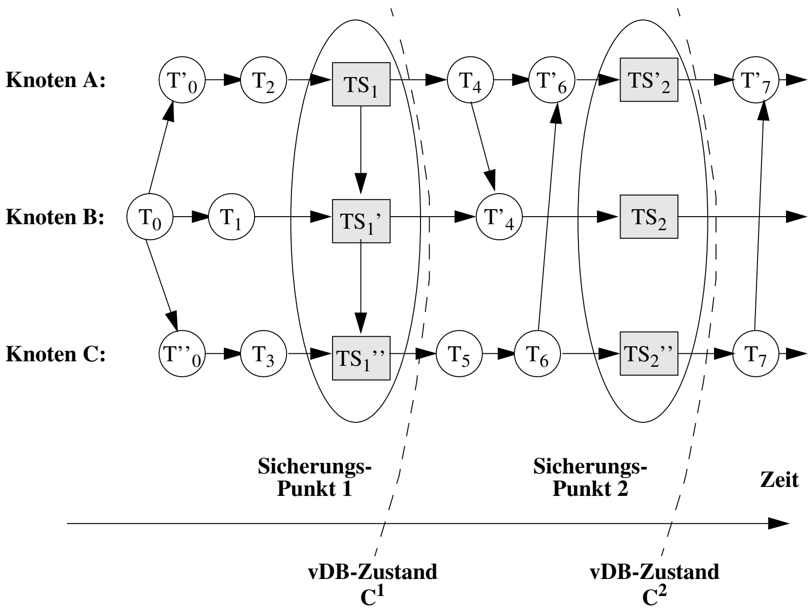
- Varianten der Erzeugung lokaler Archivkopien unwesentlich
- Wichtig ist der logische Zeitpunkt (auch Sicherungspunkt genannt), in dem sich das lokale System in einem konsistenten Zustand befunden hat
- Ansätze: Archivkopie auf Basis
  - strikt synchronisierter Sicherungspunkte (checkpoints)
  - lose synchronisierter Sicherungspunkte
  - nicht-synchronisierter Sicherungspunkte

➥ **Zielkonflikt:** Laufzeit-Aufwand (Vorsorgeaufwand) oder Aufwand bei Recovery

Strikt synchronisierte Sicherungspunkte

■ **Strikt synchronisierte Sicherungspunkte (SP)**

- Archivkopie wird an allen Knoten zeitgleich erzeugt und ist **transaktionskonsistent**
- Implementierungsmöglichkeit: Globale Transaktion mit exklusivem Zugriff



■ **Bewertung:**

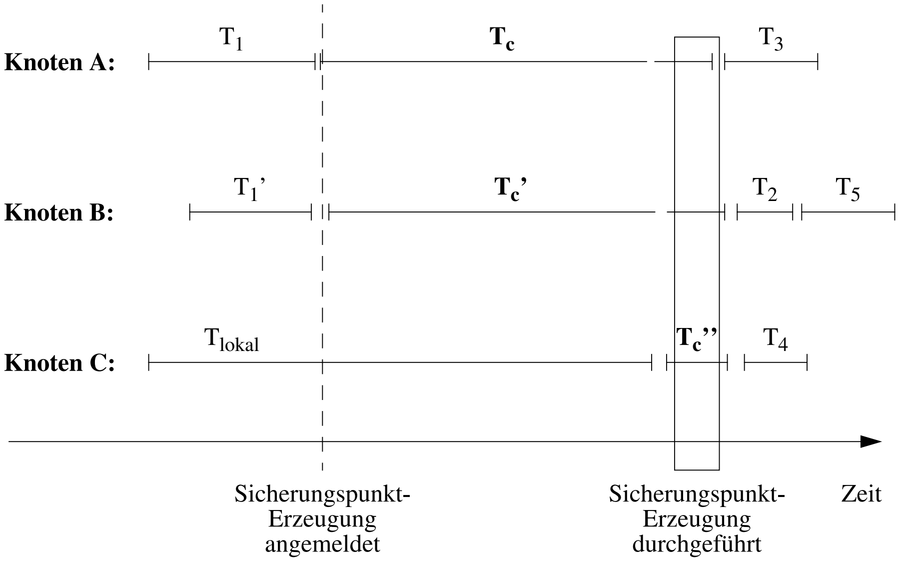
+ Verfahren zur Erzeugung der SP sind einfach zu implementieren
+ Zurücksetzen auf alten Zustand ist schnell durchzuführbar[1]
+ Für das Zurücksetzen sind nur „normale“ Archivkopie-Daten erforderlich, jedoch keine weitere Undo-/Redo-Information

- Erzeugung der SP ist „teuer“, da netzweite TA-Inaktivität erzwungen wird
- Daher ist Verfahren oftmals nur in größeren Zeitabständen durchführbar
- In Hochleistungssystemen ist es praktisch nicht einsetzbar

1 Zumindest fügt das Verfahren keinen zusätzlichen Aufwand hinzu

Lose synchronisierte Sicherungspunkte

■ **Problem bei strikt synchronisierten SP:**
Lokale TA können SP-Erzeugung verzögern



■ **Beobachtung**

- Ob eine **rein lokale** TA (wie z.B. T_{lokal}) im globalen SP (markiert durch $\{T_c, T_c', T_c''\}$ enthalten ist oder nicht, ist für die **globale** Konsistenz unerheblich[1]
- Teil-TA globaler TA (wie z.B. T_1 und T_1') müssen entweder vollständig oder garnicht im globalen SP enthalten sein
- Erzeugung einer systemweiten transaktionskonsistenten Archivkopie

1 Im Sinne von „knotenübergreifenden Konsistenzbedingungen“

Lose synchronisierte Sicherungspunkte (2)

■ **Mögliche Lösung:**

- Erzeugen der (lokalen) Sicherungspunkte

  Die globalen Sicherungspunkte werden aufsteigend durchnumeriert, etwa $C^1, C^2, ..., C^i, ...$

  Alle lokalen Sicherungspunkte, die zusammen einen globalen Sicherungspunkt bilden, haben dieselbe SP-Nummer.

  1. Ein Knoten initiiert das Erzeugen eines globalen SP durch Versenden der Nachricht „Sicherungspunkt C^i“ an alle anderen Knoten
  2. Jeder Knoten k
     - erzeugt einen lokalen Sicherungspunkt C_k^i, sobald es ihm möglich ist (keine offenen TA!)
     - fährt mit der TA-Verarbeitung unmittelbar nach Erzeugen seines SP wieder fort[1]

- Ausführung globaler Transaktionen

  1. Eine Primär-TA T am Knoten A erzeugt eine Sub-TA T' am Knoten B durch die Nachricht (T', i) an den Knoten B:

     T'= nähere Spezifikation der Sub-TA

     i = Nummer des zuletzt an Knoten A durchgeführten SP

  2. Sei C_B^j der zuletzt am Knoten B durchgeführte SP.

  Nach Empfangen der Nachricht (T', i) prüft Knoten B die Ausführbarkeit der Sub-TA wie folgt:

  IF j = i THEN führe T' aus ELSE
  IF j < i THEN verzögere die Ausführung von T' bis j = i
  ELSE weise T' zurück[2]

1 wartet also nicht auf das Erzeugen der SP durch die anderen Knoten
2 Transaktion T basiert auf einem „veralteten“ Sicherungspunkt

Lose synchronisierte Sicherungspunkte (3)

■ **Bewertung:**

+ Lokales Zurücksetzen - analog zur strikt synchronisierten Lösung - schnell durchführbar
+ Einfache Implementierung
+ I.allg. nicht so „teuer“ wie strikt synchronisierte Lösung
- Manche Transaktionen müssen zurückgewiesen werden
- Bei hohem Anteil an globalen Änderungs-TA ist Verhalten ähnlich dem bei strikt synchronisierten Sicherungspunkten

➥ Verfahren arbeitet am besten bei einem hohen Anteil rein lokaler TA

■ **Nicht-synchronisierte lokale Sicherungspunkte**

- Bei den bisherigen Verfahren wurde bei der SP-Erzeugung ein gewisser Aufwand betrieben, um im Fehlerfall relativ schnell einen global konsistenten Zustand wiederherstellen zu können
- Ohne Synchronisation sind **starke Annahmen** erforderlich:
  - Beim Zurücksetzen eines Knotens X auf einen „alten” SP C_X ist eine Analyse des lokalen Log möglich
  - Es kann festgestellt werden, welche globalen TA durch dieses lokale Zurücksetzen unvollständig geworden sind
  - Die Änderungen dieser TA können ggf. wieder aus den betroffenen Datenbanken entfernt werden, um zu einem global konsistenten Zustand zu gelangen

Zusammenfassung

■ **Gewährleistung der ACID-Eigenschaften** für verteilte Transaktionen

■ **Verteilte Commit-Protokolle**

- Sicherstellung der Atomarität und Dauerhaftigkeit bei verteilten Änderungen
- Standardverfahren: hierarchisches 2PC
- Varianten mit verbesserter Leistungsfähigkeit oder Verfügbarkeit
- relativ hoher Aufwand

■ **Integritätssicherung**

- konzeptionell weitgehend wie im zentralisierten Fall, jedoch zu höheren Kosten
- Verzögerung der Überprüfbarkeit durch Knoten- oder Verbindungsausfall usw.

■ **Synchronisation**

- Sicherstellung der globalen Serialisierbarkeit
- möglichst wenig Blockierungen und Rücksetzungen von Transaktionen sowie geringer Kommunikationsaufwand
- verteilte Sperrverfahren vorzuziehen

■ **Globale Deadlock-Behandlung**

- Deadlock-Vermeidung (z.B. Wound/Wait) vermeidet Kommunikationsaufwand, führt jedoch zu unnötigen Rücksetzungen
- verteilte Deadlock-Erkennung: reduziert Anzahl Rücksetzungen, jedoch aufwendige Realisierung und teuer ($O(N^2)$)

■ **Logging und Recovery**

- in jedem Knoten Techniken wie im zentralisierten Fall
- Geräte-Recovery verlangt systemweite Synchronisation