7. Transaktionsverwaltung

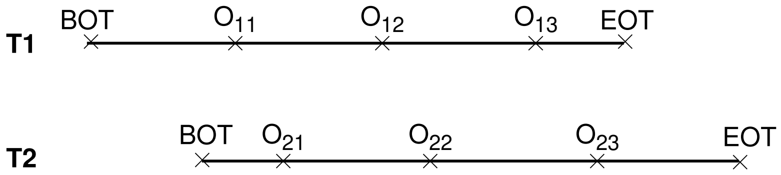
- **Gefährdung der DB-Konsistenz**
- **Transaktionskonzept**
  - ACID-Eigenschaften
  - Ablauf einer Transaktion
  - Aufgaben einer Transaktionsverwaltung
- **Überblick über Logging/Recovery (Atomarität, Dauerhaftigkeit)**
- **Synchronisation (Isolation)**
  - Mehrbenutzeranomalien
  - Korrektheitskriterium der Serialisierbarkeit
  - Sperrprotokolle
  - Konsistenzebenen
- **Commit-Protokolle (Atomarität)**
  - Einsatz von Commit-Protokollen (zentralisierter TA-Ablauf)
  - 2PC (Zweiphasen-Commit-Protokoll)

Gefährdung der DB-Konsistenz

	Korrektheit der Abbildungshierarchie	Übereinstimmung zwischen DB und Miniwelt
durch das Anwendungsprogramm	Mehrbenutzer-Anomalien **Synchronisation**	unzulässige Änderungen **Integritätsüberwachung des DBVS** **TA-orientierte Verarbeitung**
durch das DBVS und die Betriebsumgebung	Fehler auf den Externspeichern, Inkonsistenzen in den Zugriffspfaden **Fehlertolerante Implementierung** **Archivkopien (Backup)**	undefinierter DB-Zustand nach einem Systemausfall **Transaktionsorientierte Fehlerbehandlung (Recovery)**

Transaktionskonzept

- **Ablaufkontrollstruktur: Transaktion**



- **Welche Eigenschaften von Transaktionen sind zu garantieren? (zur Erinnerung)**

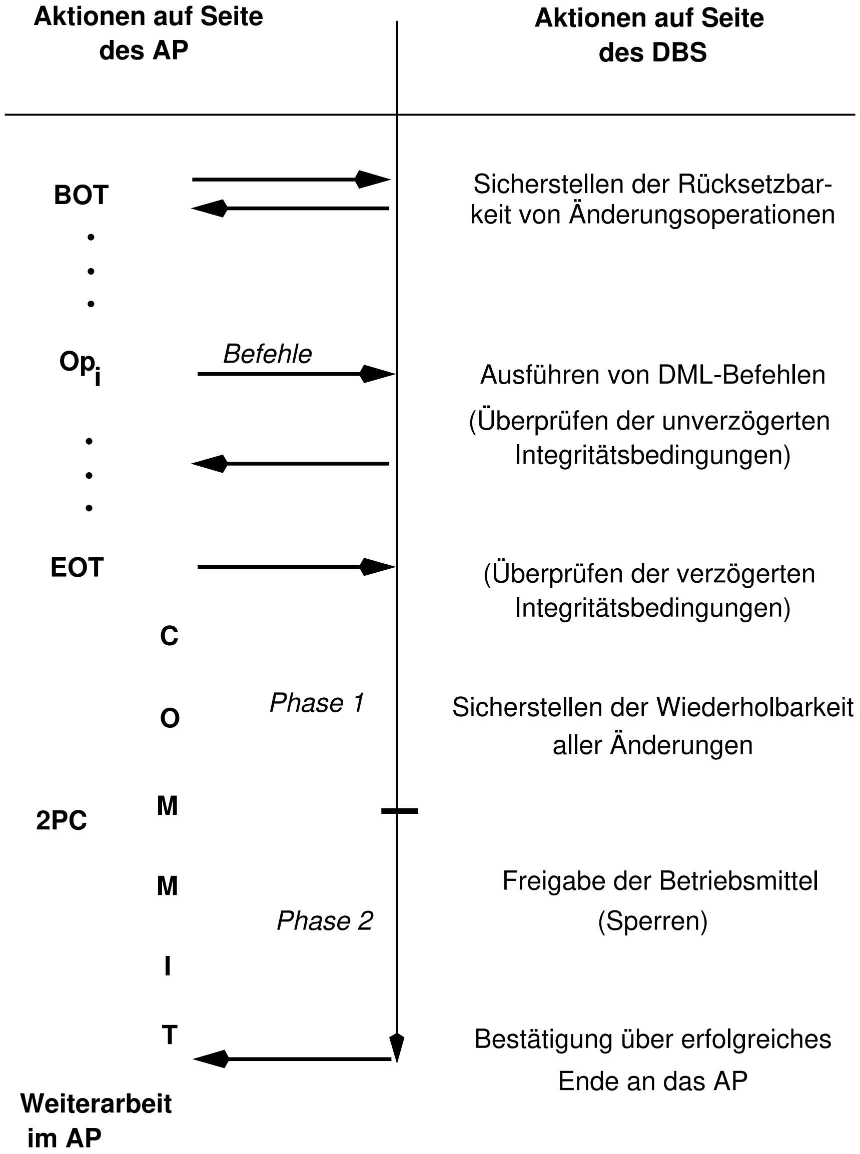
  - **Atomicity (Atomarität)**
    - TA ist kleinste, nicht mehr weiter zerlegbare Einheit
    - Entweder werden alle Änderungen der TA festgeschrieben oder gar keine („alles-oder-nichts"-Prinzip)

  - **Consistency**
    - TA hinterläßt einen konsistenten DB-Zustand, sonst wird sie komplett (siehe Atomarität) zurückgesetzt
    - Zwischenzustände während der TA-Bearbeitung dürfen inkonsistent sein
    - Endzustand muß die Integritätsbedingungen des DB-Schemas erfüllen

  - **Isolation**
    - Nebenläufig (parallel, gleichzeitig) ausgeführte TA dürfen sich nicht gegenseitig beeinflussen
    - Parallele TA bzw. deren Effekte sind nicht sichtbar

  - **Durability (Dauerhaftigkeit)**
    - Wirkung erfolgreich abgeschlossener TA bleibt dauerhaft in der DB
    - TA-Verwaltung muß sicherstellen, daß dies auch nach einem System-fehler (HW- oder System-SW) gewährleistet ist
    - Wirkung einer erfolgreich abgeschlossenen TA kann nur durch eine sog. kompensierende TA aufgehoben werden

Schnittstelle zwischen AP und DBS - transaktionsbezogene Aspekte



Transaktionsverwaltung

- **DB-bezogene Definition der Transaktion:**
  Eine TA ist eine ununterbrechbare Folge von DML-Befehlen, welche die Datenbank von einem logisch konsistenten Zustand in einen neuen logisch konsistenten Zustand überführt.

➥ Diese Definition eignet sich insbesondere für relativ kurze TA, die auch als ACID-Transaktionen bezeichnet werden.

- **Wesentliche Abstraktionen aus Sicht der DB-Anwendung**
  - Alle Auswirkungen auftretender Fehler bleiben der Anwendung verborgen (*failure transparency*)
  - Es sind keine anwendungsseitigen Vorkehrungen zu treffen, um Effekte der Nebenläufigkeit beim DB-Zugriff auszuschließen (*concurrency transparency*)

➥ Gewährleistung einer fehlerfreien Sicht auf die Datenbank im logischen Einbenutzerbetrieb

- **Transaktionsverwaltung**
  - koordiniert alle DBS-seitigen Maßnahmen, um ACID zu garantieren
  - besitzt zwei wesentliche Komponenten
    - Synchronisation
    - Logging und Recovery
  - kann zentralisiert oder verteilt (z. B. bei VDBS) realisiert sein
  - soll Transaktionsschutz für heterogene Komponenten bieten

DB-Recovery

- **Aufgabe des DBVS:**
  Automatische Behandlung aller erwarteten Fehler

- **Voraussetzung:**
  Sammeln redundanter Informationen während des normalen Betriebes (*Logging*)

- **Fehlermodell von zentralisierten DBVS**
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler

- **Probleme**
  - Fehlererkennung
  - Fehlereingrenzung
  - Abschätzung des Schadens
  - Durchführung der Recovery

- **“A recoverable action is 30% harder and requires 20% more code than a non-recoverable action” (J. Gray)**

DB-Recovery (2)

- **Transaktionsparadigma verlangt:**
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen

- **Zielzustand nach erfolgreicher Recovery:**

  Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritätsbedingungen entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt

  ➥ ***jüngster transaktionskonsistenter DB-Zustand***

  In welchem Zustand befindet sich die Systemumgebung? (Betriebssystem, Anwendungssystem, andere Komponenten)

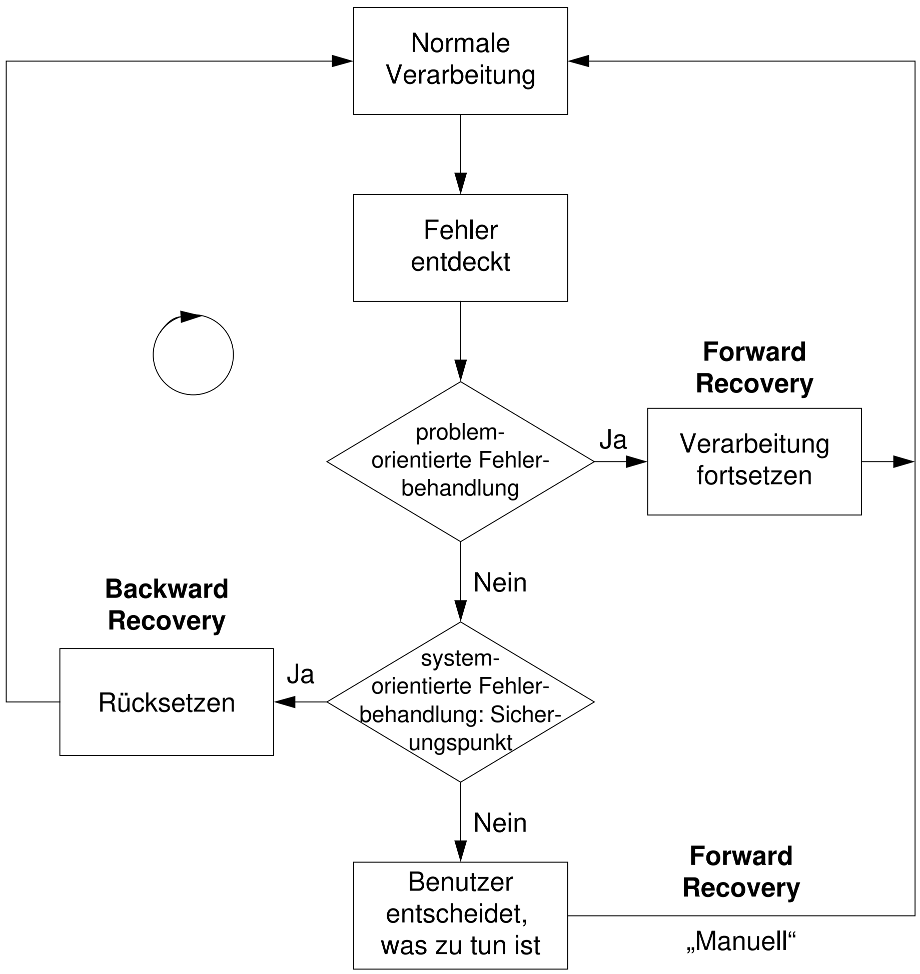
- **Forward-Recovery i. allg. nicht anwendbar**
  - Fehlerursache häufig falsche Programme, Eingabefehler u. ä.
  - durch Fehler unterbrochene Transaktionen sind zurückzusetzen (*Backward Recovery*)

- **Backward-Recovery**

  setzt voraus, daß auf allen Abstraktionsebenen genau definiert ist, auf welchen Zustand die DB im Fehlerfall zurückzusetzen ist.

Recovery – Begriffsklärung

- **Grundsätzliche Vorgehensweisen**



Recovery-Arten

1. **Transaktions-Recovery**
   Zurücksetzen einzelner (noch nicht abgeschlossener) Transaktionen im laufenden DB-Betrieb (Transaktionsfehler, Deadlock, etc.)
   - vollständiges Zurücksetzen auf Transaktionsbeginn (TA-UNDO) bzw.
   - partielles Zurücksetzen auf Rücksetzpunkt (*Savepoint*) innerhalb der Transaktion

2. **Crash-Recovery nach Systemfehler**

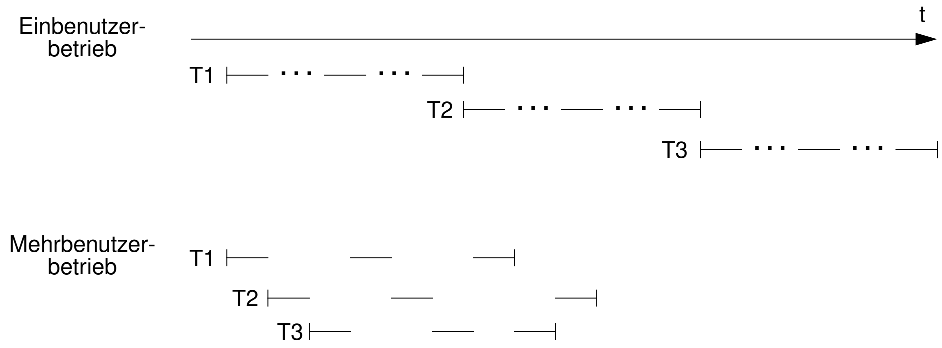
   Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:
   - (partielles) REDO für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
   - UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen der Änderungen aus der permanenten DB)

3. **Medien-Recovery nach Gerätefehler**
   - Spiegelplatten bzw.
   - vollständiges Wiederholen (REDO) aller Änderungen auf einer Archivkopie

4. **Katastrophen-Recovery**
   - Nutzung einer aktuellen DB-Kopie in einem “entfernten” System oder
   - stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf der Basis gesicherter Archivkopien (Datenverlust!)

Warum Mehrbenutzerbetrieb?

- **Ausführung von Transaktionen**



  - CPU-Nutzung während TA-Unterbrechungen
    - E/A
    - Denkzeiten bei Mehrschritt-TA
    - Kommunikationsvorgänge in verteilten Systemen
  - bei langen TA zu große Wartezeiten für andere TA (Scheduling-Fairneß)

Anomalien im unkontrollierten Mehrbenutzerbetrieb

1. Abhängigkeit von nicht freigegebenen Änderungen *(dirty read, dirty overwrite)*
2. Verlorengegangene Änderung *(lost update)*
3. Inkonsistente Analyse *(non-repeatable read)*
4. Phantom-Problem

➥ **nur durch Änderungs-TA verursacht**

Unkontrollierter Mehrbenutzerbetrieb

- **Abhängigkeit von nicht freigegebenen Änderungen**

T1	T2
read (A); A := A + 100 write (A);	
	read (A); read (B); B := B + A; write (B); commit;
abort;	

  - Geänderte, aber noch nicht freigegebene Daten werden als „schmutzig“ bezeichnet (dirty data), da die TA ihre Änderungen bis Commit (einseitig) zurücknehmen kann
  - Schmutzige Daten dürfen von anderen TAs nicht in „kritischen” Operationen benutzt werden

- **Verlorengegangene Änderung (Lost Update)**

T1	T2	A in DB
read (A);		
	read (A);	
A := A - 1;		
write (A);		
	A := A - 1;	
	write (A);	

  - **Verlorengegangene Änderungen sind auszuschließen!**

Inkonsistente Analyse (Non-repeatable Read)

Das wiederholte Lesen einer gegebenen Folge von Daten führt auf verschiedene Ergebnisse:

Lesetransaktion (Gehaltssumme berechnen)	Änderungstransaktion	DB-Inhalt (Pnr, Gehalt)
		2345 39.000 3456 48.000
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 2345 summe := summe + gehalt		
	UPDATE Pers SET Gehalt = Gehalt + 1000 WHERE Pnr = 2345	2345 40.000
	UPDATE Pers SET Gehalt = Gehalt + 2000 WHERE Pnr = 3456	3456 50.000
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 3456 summe := summe + gehalt		

Zeit

Phantom-Problem

Einfügungen oder Löschungen können Leser zu falschen Schlußfolgerungen verleiten:

Lesetransaktion (Gehaltssumme überprüfen)	Änderungstransaktion (Einfügen eines neuen Angestellten)
SELECT SUM (Gehalt) INTO :summe FROM Pers WHERE Anr = 17	
	INSERT INTO Pers (Pnr, Anr, Gehalt) VALUES (4567, 17, 55.000)
	UPDATE Abt SET Gehaltssumme = Gehaltssumme + 55.000 WHERE Anr = 17
SELECT Gehaltssumme INTO :gsumme FROM Abt WHERE Anr = 17	
IF gsumme <> summe THEN <Fehlerbehandlung>	

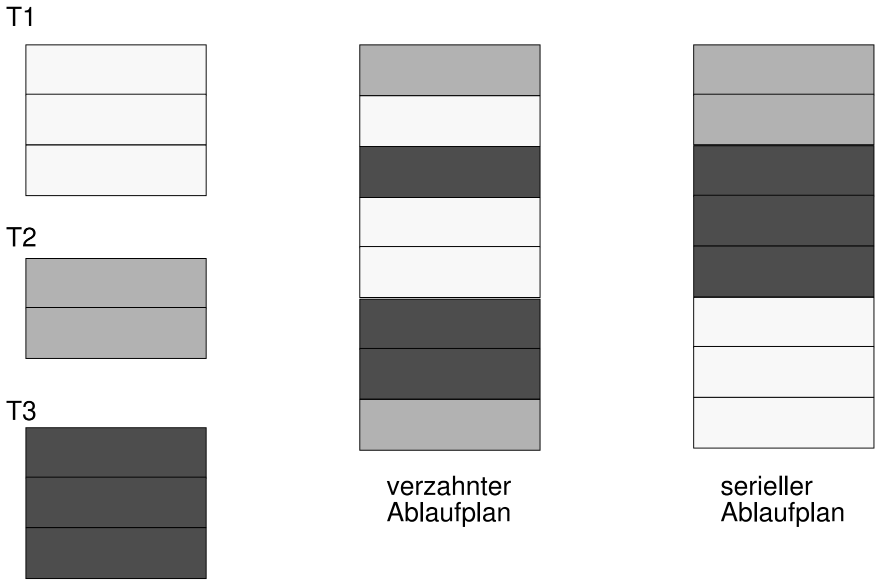
Zeit

Synchronisation von Transaktionen

- **TRANSAKTION:** Ein Programm T mit DML-Anweisungen, das folgende Eigenschaft erfüllt:

  Wenn T **allein** auf einer konsistenten DB ausgeführt wird, dann terminiert T (irgendwann) und hinterläßt die DB in einem konsistenten Zustand. (Während der TA-Verarbeitung werden keine Konsistenzgarantien eingehalten)

- **Ablaufpläne für 3 Transaktionen**



➥ Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.

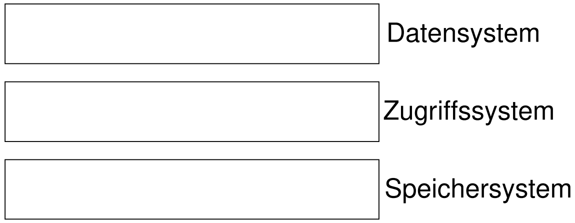
- **Ziel der Synchronisation:**

  logischer Einbenutzerbetrieb,
  d.h. Vermeidung aller Mehrbenutzeranomalien

  ➥ **Wann ist die parallele Ausführung von n Transaktionen auf gemeinsamen Daten korrekt?**

Synchronisation - Modellannahmen

- **Modellbildung**
  für die Synchronisation



- **Read/Write-Modell**
  - DB ist Menge von unteilbaren, uninterpretierten Datenobjekten (z. B. Seiten)
  - DB-Anweisungen lassen sich nachbilden durch atomare Lese- und Schreiboperationen auf Objekten:
    - $r_i(A)$, $w_i(A)$ zum Lesen bzw. Schreiben des Datenobjekts A
    - c_i , a_i zur Durchführung eines **commit** bzw. **abort**
  - **Transaktion** wird modelliert als eine endliche Folge von Operationen p_i:

    $T = p_1\, p_2\, p_3 \ldots p_n$ mit $p_i \in \{r(x_i), w(x_i)\}$
  - Eine vollständige TA hat als letzte Operation entweder einen Abbruch a oder ein Commit c

    $T = p_1 \ldots p_n\, a$ oder $T = p_1 \ldots p_n\, c$

  ➥ **Für eine TA T_i werden diese Operationen mit r_i, w_i, c_i oder a_i bezeichnet, um sie zuordnen zu können**

- **Die Ablauffolge von TA mit ihren Operationen kann durch einen *Schedule* (Ablaufplan) beschrieben werden:**

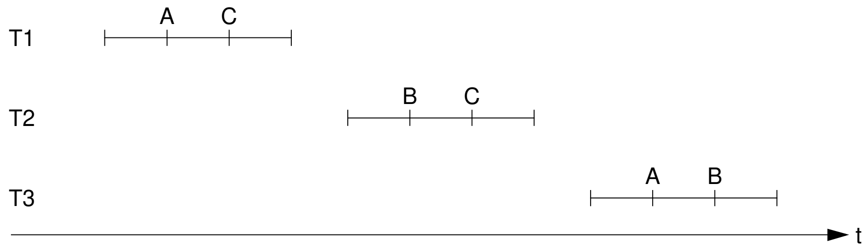
  Beispiel:
  $r_1(A)$, $r_2(A)$, $r_3(B)$, $w_1(A)$, $w_3(B)$, $r_1(B)$, c_1, $r_3(A)$, $w_2(A)$, a_2, $w_3(C)$, c_3, ...

Korrektheitskriterium der Synchronisation

- **Serieller Ablauf von Transaktionen**

  TA = {T1, T2, T3}

  DB = {A, B, C}



  Ausführungsreihenfolge:

- **T1 | T2 bedeutet:**

  T1 sieht keine Änderungen von T2 und
  T2 sieht alle Änderungen von T1

- **Formales Korrektheitskriterium: *Serialisierbarkeit*:**

Die parallele Ausführung einer Menge von TA ist **serialisierbar**, wenn es eine serielle Ausführung derselben TA-Menge gibt, die den **gleichen DB-Zustand** und die **gleichen Ausgabewerte** wie die ursprüngliche Ausführung erzielt.

- **Hintergrund:**
  - Serielle Ablaufpläne sind korrekt!
  - Jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar

Synchronisationsverfahren

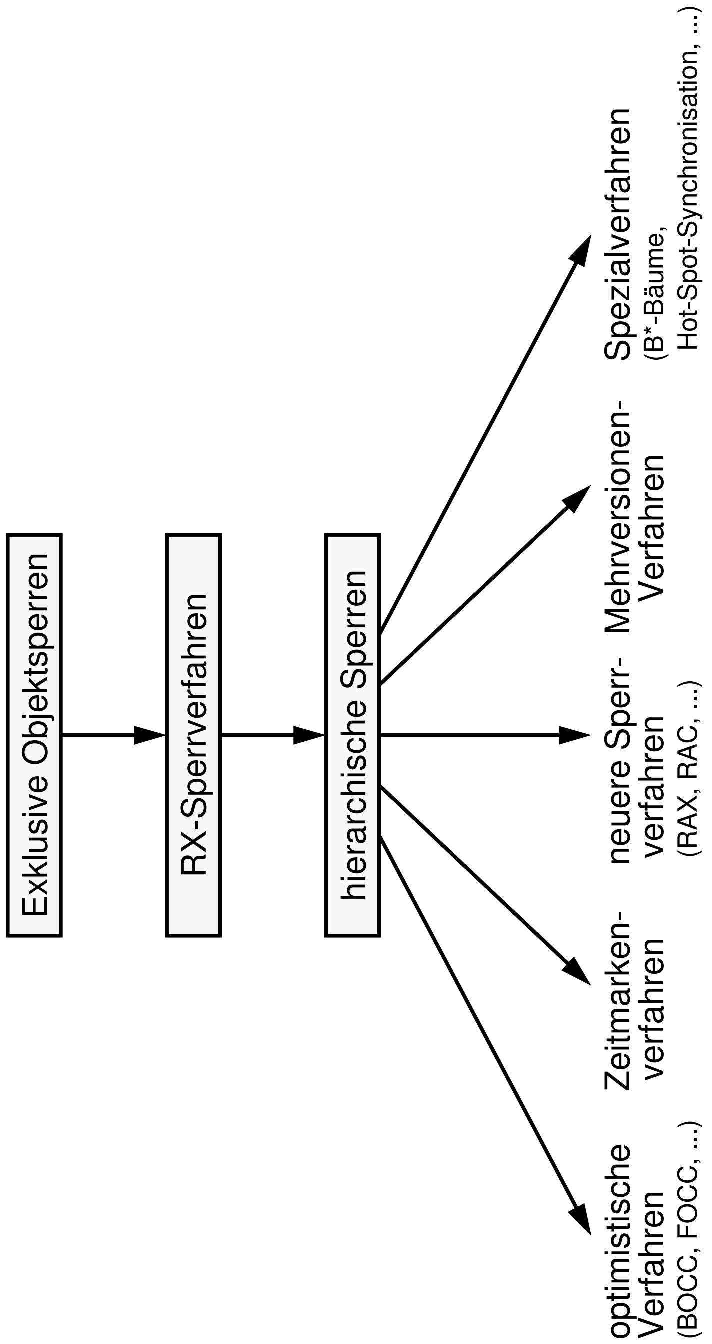
- **Scheduler**
  - Komponente der Transaktionsverwaltung
  - kontrolliert die beim TA-Ablauf auftretenden Konfliktoperationen (R/W, W/R, W/W) und garantiert insbesondere, daß nur „serialisierbare“ TA erfolgreich beendet werden
  - Nicht serialisierbare TA müssen verhindert werden.
    Dazu ist eine Kooperation mit der Recovery-Komponente erforderlich (Rücksetzen von TA)

- **Zur Realisierung der Synchronisation gibt es viele Verfahren**
  - Pessimistische Verfahren: Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt.
  - Optimistische Verfahren: Erst bei Commit wird überprüft, ob die TA serialisierbar ist
  - Versionsverfahren, Zeitmarkenverfahren usw.
  - . . .

  ➥ **Sperrverfahren sind pessimistisch und universell einsetzbar.**

- **Sperrbasierte Synchronisation**
  - Sperren stellen während des laufenden Betriebs sicher, daß die resultierende Historie serialisierbar bleibt
  - Es gibt mehrere Varianten

Historische Entwicklung von Synchronisationsverfahren



RX-Sperrverfahren

- **Sperrmodi**
  - Sperrmodus des Objektes: NL (no lock), R (read), X (exclusive)
  - Sperranforderung einer Transaktion: R, X
- **Kompatibilitätsmatrix:**

angeforderter Modus der TA \ aktueller Modus des Objekts	NL	R	X
R	+	+	-
X	+	-	-

  - Falls Sperre nicht gewährt werden kann, muß die anfordernde TA warten, bis das Objekt freigegeben wird (Commit/Abort der die Sperre besitzenden TA)
  - Wartebeziehungen werden in einem Wait-for-Graph verwaltet

- **Ablauf von Transaktionen**

T1	T2	a	b	Bem.
		NL	NL	
lock (a, X)		X		
. . .				
	lock (b, R)		R	
	. . .			
lock (b, R)			R	
	lock (a, R)	X		T2 wartet
. . .				
unlock (a)		NL --> R		T2 wecken
. . .	. . .			
unlock(b)			R	

Zweiphasen-Sperrprotokolle[1]

- **Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:**
  1. Vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
  2. Gesetzte Sperren anderer TA sind zu beachten
  3. Eine TA darf nicht mehrere Sperren für ein Objekt anfordern
  4. ***Zweiphasigkeit:***
     - Anfordern von Sperren erfolgt in einer *Wachstumsphase*
     - Freigabe der Sperren in *Schrumpfungsphase*
     - Sperrfreigabe kann erst beginnen, wenn alle benötigten Sperren gehalten werden
  5. Spätestens bei Commit sind alle Sperren freizugeben

- **Beispiel für ein 2PL-Protokoll** (2PL: two-phase locking)

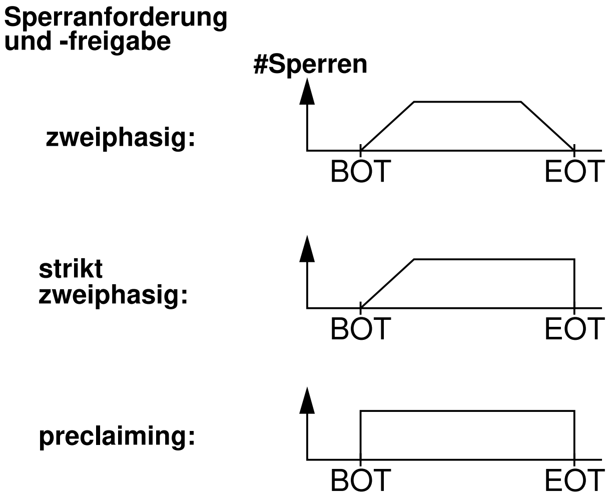
BOT
lock (a, X)
. . .
lock (b, R)
. . .
lock (c, X)
. . .
unlock (b)
unlock (c)
unlock (a)
Commit

An der SQL-Schnittstelle ist die Sperranforderung und -freigabe nicht sichtbar!

1. Eswaran, K.P. et al.: The notions of consistency and predicate locks in a data base system, Comm. ACM 19:11, 1976, pp. 624-633

Zweiphasen-Sperrprotokolle (2)

- **Formen der Zweiphasigkeit**



- **Anwendung des 2PL-Protokolls**

T1	T2	Bem.
BOT		
lock (a, X)		
read (a)		
write (a)		
	BOT	
	lock (a, X)	T2 wartet
lock (b, X)		
read (b)		
unlock (a)		T2 wecken
	read (a)	
	write (a)	
	unlock (a)	
	commit	
unlock (b)		

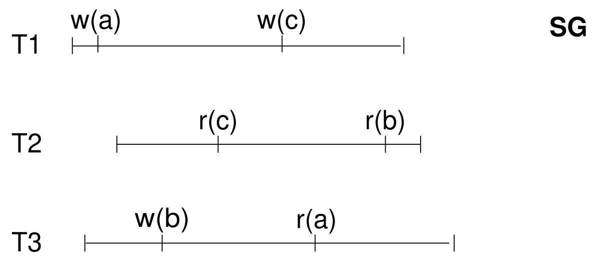
➥ Praktischer Einsatz erfordert **striktes 2PL-Protokoll**!

Verklemmungen (Deadlocks)

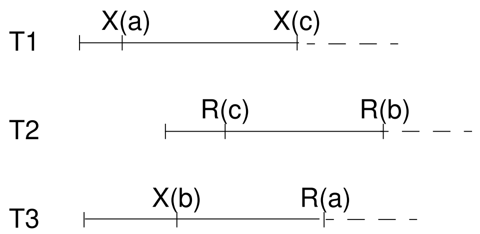
- **Striktes 2PL-Protokoll**
  - gibt alle Sperren erst bei Commit frei und
  - verhindert dadurch kaskadierendes Rücksetzen

➥ Auftreten von Verklemmungen ist **inhärent** und kann bei pessimistischen Methoden (blockierende Verfahren) nicht vermieden werden.

- **Nicht-serialisierbare Historie**



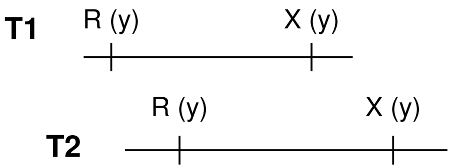
- **RX-Verfahren verhindert** das Auftreten einer nicht-serialisierbaren Historie, **aber nicht (immer) Deadlocks**



RUX-Sperrverfahren

- **Forderung**
  - Wahl des gemäß der Operation schwächst möglichen Sperrmodus
  - Möglichkeit der Sperrkonversion (upgrading), falls stärkerer Sperrmodus erforderlich
  - Anwendung: viele Objekte sind zu lesen, aber nur wenige zu aktualisieren

- **Problem: Sperrkonversionen**



- **Erweitertes Sperrverfahren:**
  - Ziel: Verhinderung von Konversions-Deadlocks
  - U-Sperre für Lesen mit Änderungsabsicht (Prüfmodus)
  - bei Änderung Konversion U $\rightarrow$ X, andernfalls U $\rightarrow$ R (downgrading)

- **Wirkungsweise**

RUX-Sperrverfahren (2)

- **Symmetrische Variante**
  - Was bewirkt eine Symmetrie bei U?

	R	U	X
R	+	+	-
U	+	-	-
X	-	-	-

- **Beispiel**

- **Unsymmetrie bei U**

	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

  - u. a. in DB2 eingesetzt

- **Beispiel**

Konsistenzebenen

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch" Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
  - „Schwächere" Konsistenzebene bei der Synchronisation von Leseoperationen erlaubt höhere Parallelitätsgrade und Reduktion von Blockierungen, erfordert aber Programmierdisziplin!

  ➥ **Inkaufnahme von Anomalien reduziert die TA-Behinderungen**

- **Konsistenzebenen** (basierend auf verkürzte Sperrdauern)

Ebene 3: Transaktion T sieht Konsistenzebene 3, wenn gilt:

a) T verändert keine schmutzigen Daten anderer Transaktionen

b) T gibt keine Änderungen vor EOT frei

c) T liest keine schmutzigen Daten anderer Transaktionen

d) Von T gelesene Daten werden durch andere Transaktionen erst nach EOT von T verändert

Ebene 2: Transaktion T sieht Konsistenzebene 2, wenn sie die Bedingungen a, b und c erfüllt

Ebene 1: Transaktion T sieht Konsistenzebene 1, wenn sie die Bedingungen a und b erfüllt

Ebene 0: Transaktion T sieht Konsistenzebene 0, wenn sie nur Bedingung a erfüllt

Konsistenzebenen (2)

- ***Konsistenzebene 3*:**
  - wünschenswert, jedoch oft viele Sperrkonflikte wegen langer Schreib- und Lesesperren

- ***Konsistenzebene 2:***
  - nur lange Schreibsperren, jedoch kurze Lesesperren
  - *'unrepeatable read'* möglich

- ***Konsistenzebene 1:***
  - lange Schreibsperren, keine Lesesperren
  - *'dirty read'* (und 'lost update') möglich

- ***Konsistenzebene 0:***
  - kurze Schreibsperren ('Chaos')

  ➥ Kommerzielle DBS empfehlen meist Konsistenzebene 2

- **Wahlangebot**

  Einige DBS (DB2, Tandem NonStop SQL, ...) bieten Wahlmöglichkeit zwischen:
  - 'repeatable read' (Ebene 3) und
  - 'cursor stability' (Ebene 2)

  Einige DBS bieten auch *BROWSE-Funktion*, d. h. Lesen ohne Setzen von Sperren (Ebene 1)

Konsistenzebenen (4)

- SQL erlaubt Wahl zwischen vier Konsistenzebenen (Isolation Level)
- **Konsistenzebenen sind durch die Anomalien bestimmt,** die jeweils in Kauf genommen werden:
  - Abgeschwächte Konsistenzanforderungen betreffen nur Leseoperationen!
  - **Lost Update** muß generell vermieden werden, d. h., W/W-Abhängigkeiten müssen stets beachtet werden

Konsistenz-ebene	Anomalie: Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

  - Default ist **Serialisierbarkeit** (serializable)

- **SQL-Anweisung zum Setzen der Konsistenzebene:**

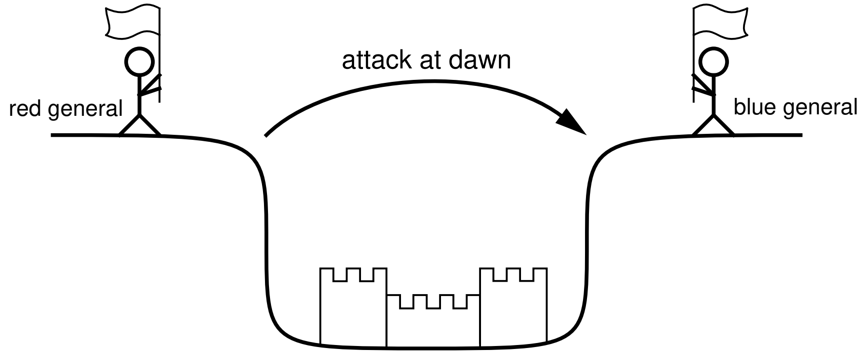
```
SET TRANSACTION  [mode] [ISOLATION LEVEL level]
```

  - Transaktionsmodus: READ WRITE (Default) bzw. READ ONLY
  - **Beispiel:**
    SET TRANSACTION READ ONLY, ISOLATION LEVEL READ COMMITTED
  - READ UNCOMMITTED für Änderungstransaktionen unzulässig

Verarbeitung in Verteilten Systemen

- Ein ***verteiltes System*** besteht aus autonomen Subsystemen, die koordiniert zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen
  - Client/Server-Systeme
  - Mehrrechner-DBS, . . .

- **Beispiel: The „Coordinated Attack“ Problem**



Generals-Paradoxon

- **Grundproblem verteilter Systeme**

> *Das für verteilte Systeme charakteristische Kernproblem ist der Mangel an globalem (zentralisiertem) Wissen*

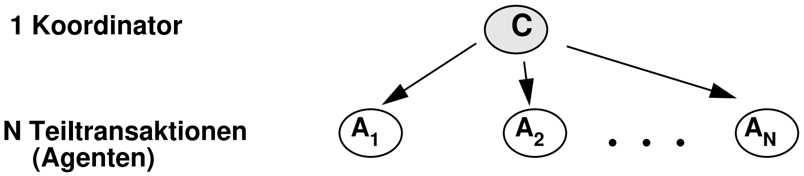
➥ **symmetrische Kontrollalgorithmen sind oft zu teuer oder zu ineffektiv**

➥ **fallweise Zuordnung der Kontrolle**

Verarbeitung in Verteilten Systemen (2)

- **Erweitertes Transaktionsmodell**

  verteilte Transaktionsbearbeitung (Primär-, Teiltransaktionen)



➥ *rechnerübergreifendes Mehrphasen-Commit-Protokoll notwendig, um Atomizität einer globalen Transaktion sicherzustellen*

- **Anforderungen an geeignetes Commit-Protokoll:**
  - Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
  - Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern

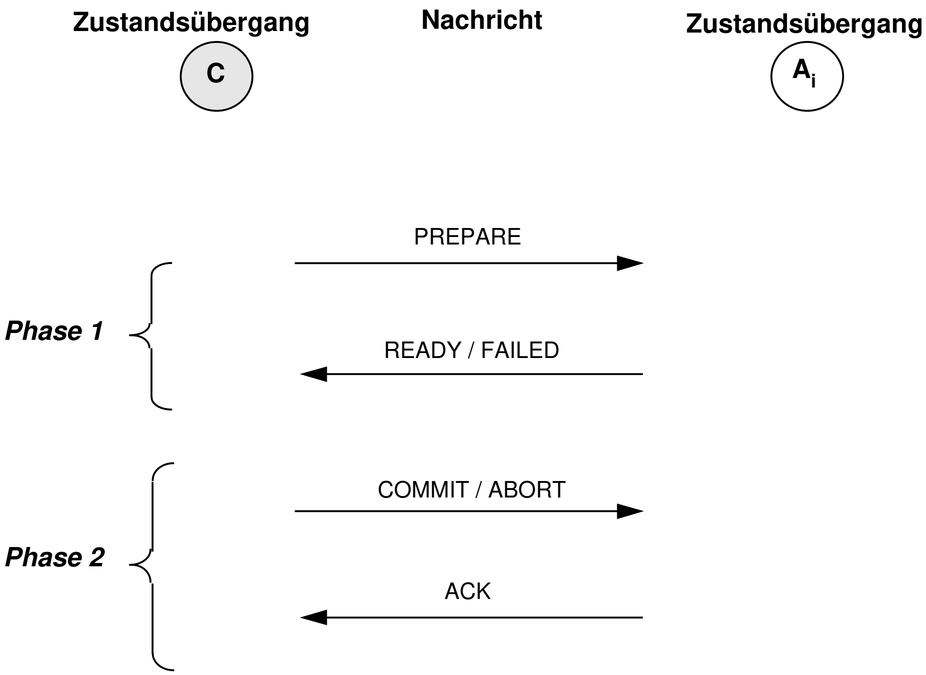
➥Zentralisiertes Zweiphasen-Commit-Protokoll stellt geeignete Lösung dar

- **Erwartete Fehlersituationen**
  - Transaktionsfehler
  - Systemfehler (Crash)

    ➥ i. allg. partielle Fehler (Rechner, Verbindungen, ...)
  - Gerätefehler

➥ Fehlererkennung z. B. über Timeout

Zentralisiertes Zweiphasen-Commit



- **Protokoll erfordert Folge von Zustandsübergängen**
  - für Koordinator
  - für jeden Agenten

➥ Zustandsübergänge müssen auf „sicherem Platz“ (Log) vermerkt sein!

Zusammenfassung

- **Transaktionsparadigma**
  - Verarbeitungsklammer für die Einhaltung von semantischen Integritätsbedingungen
  - Verdeckung von (erwarteten) Fehlerfällen (*failure isolation*)

    ➥ Logging/Recovery
  - Verdeckung der Nebenläufigkeit (*concurrency isolation*)

    ➥ Synchronisation
  - im SQL-Standard: COMMIT WORK, ROLLBACK WORK

    ➥ Beginn einer Transaktion implizit

- **Logging**/**Recovery**
  - Recovery-Arten

- **Synchronisation**
  - Korrektheitskriterium Serialisierbarkeit
  - Sperrverfahren
  - Konsistenzebenen

- **Zweiphasen-Commit-Protokoll**
  - Einsatz in allen Systemen