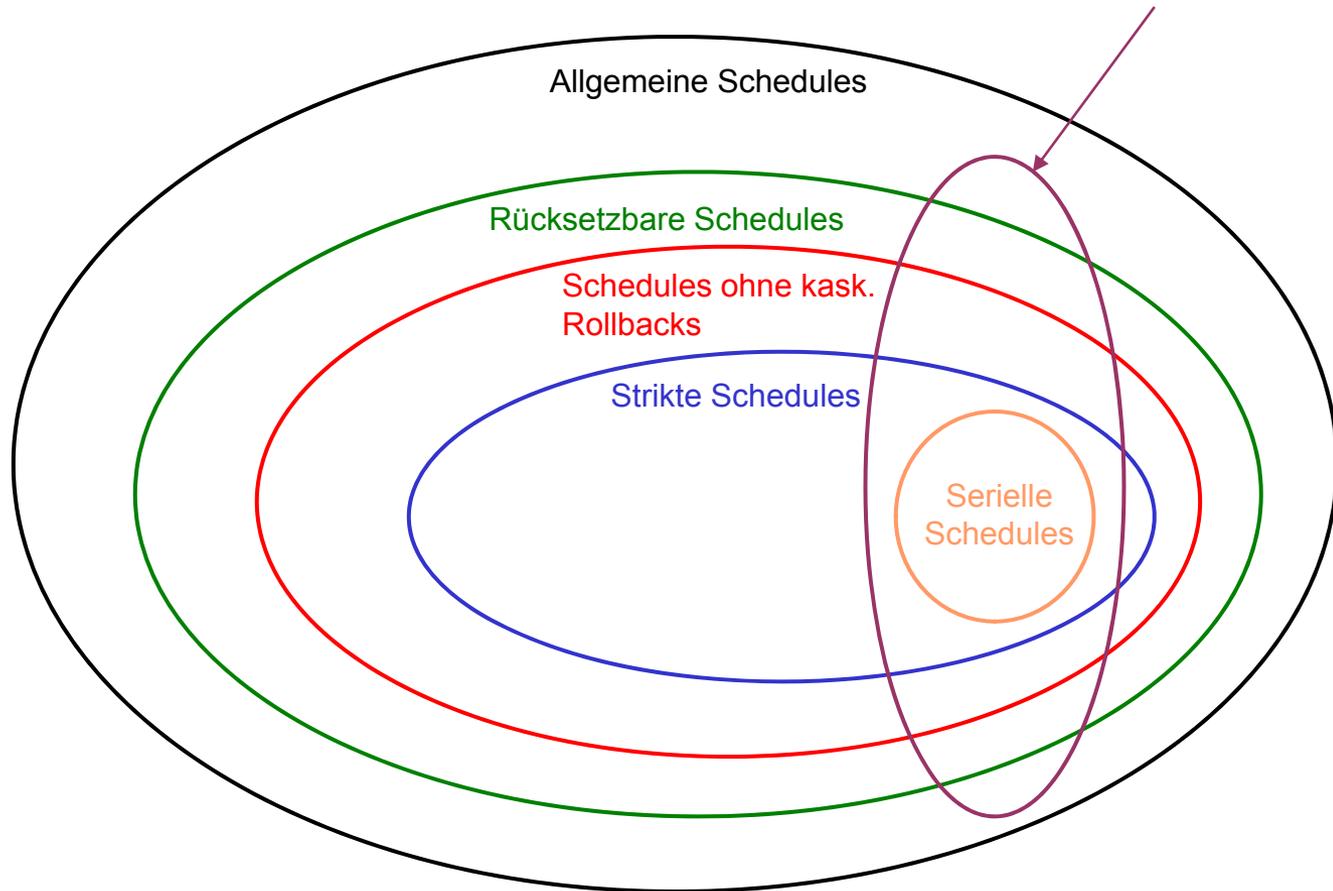


Nachtrag zu Scheduleklassen

- Serieller S.
 - TAs in einzelnen Blöcken
- Serialisierbarer S.
 - Konfliktäquivalent zu einem seriellen S.
- Rücksetzbarer S.
 - TA darf erst committen, wenn alle TAs von denen sie Daten gelesen hat committed haben
- S. ohne kaskadierendes Rollback
 - Veränderte Daten einer noch laufenden TA dürfen nicht gelesen werden
- Strikter S.
 - Zusätzlich dürfen veränderte Daten einer noch laufenden TA nicht überschrieben werden

Nachtrag zu Scheduleklassen

- Beziehungen zwischen S-Klassen



Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

Sperre (Lock)

- Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
- Anforderung einer Sperre durch *LOCK*, Freigabe durch *UNLOCK*
- *LOCK / UNLOCK* erfolgt atomar
- Sperrgranularität:
Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
- Sperrenverwalter führt Tabelle für aktuell gewährte Sperren

Legale Schedules

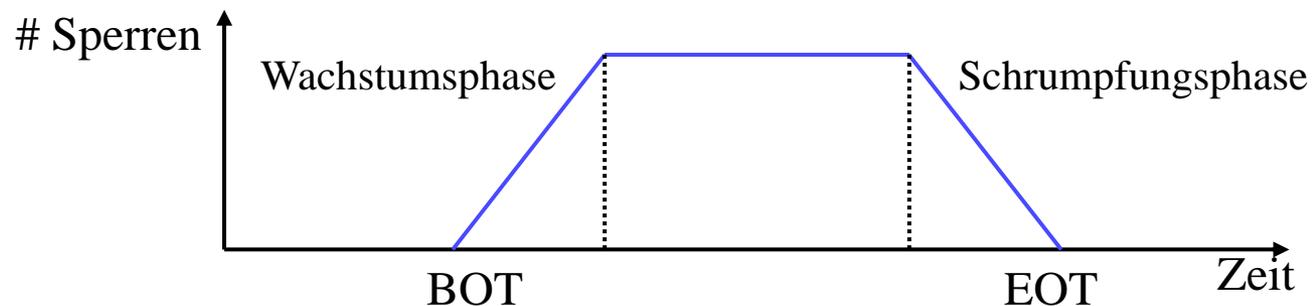
- Vor jedem Zugriff auf ein Objekt wird eine geeignete Sperre gesetzt.
- Keine Transaktion fordert eine Sperre an, die sie schon besitzt.
- Spätestens bei Transaktionsende werden alle Sperren zurückgegeben.
- Sperren werden respektiert, d.h. eine mit gesetzten Sperren unverträgliche Sperranforderung (z.B. exklusiver Zugriff auf Objekt x) muss warten.

Bemerkungen

- Anfordern und Freigeben von Sperren sollte das DBMS implizit selbst vornehmen.
- Die Verwendung legaler Schedules garantiert noch nicht die Serialisierbarkeit.

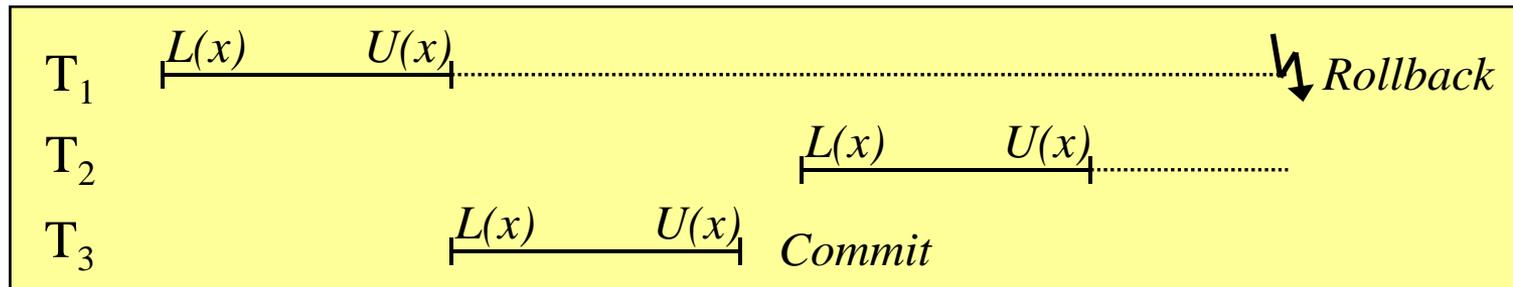
Zwei-Phasen-Sperrprotokoll (2PL)

- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- **Merkmal:** keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
 - *Wachstumsphase:* Anforderungen der Sperren
 - *Schrumpfungsphase:* Freigabe der Sperren



Zwei-Phasen-Sperrprotokoll (2PL)

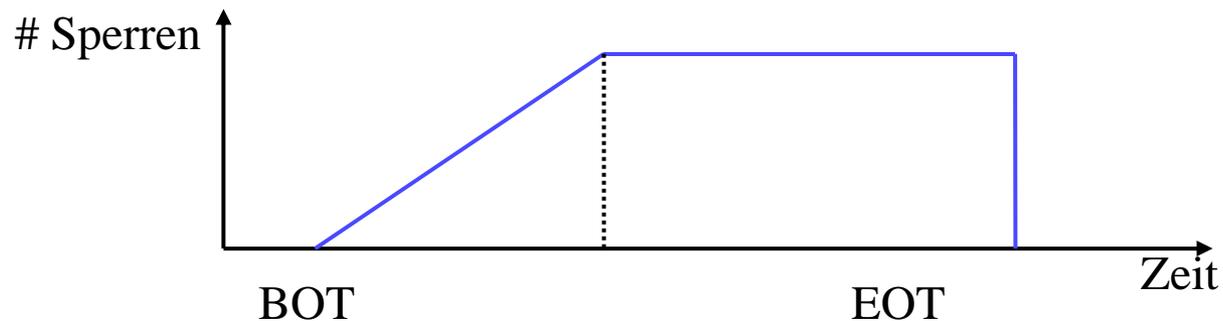
- Serialisierbarkeit ist gewährleistet, da Serialisierungsgraphen keine Zyklen enthalten können ☺
- Problem : Gefahr des kaskadierenden Rücksetzens im Fehlerfall (bzw. sogar **Nicht-Rücksetzbar**) ☹



- Transaktion T_1 wird nach $U(x)$ zurückgesetzt
- T_2 hat "schmutzig" gelesen und muss zurückgesetzt werden
- Sogar T_3 muss zurückgesetzt werden → eklatanter Verstoß gegen die Dauerhaftigkeit (ACID) des **COMMIT!**

Striktes Zwei-Phasen-Sperrprotokoll

- Abhilfe durch striktes (oder strenges) Zwei-Phasen-Sperrprotokoll:
 - Alle Sperren werden bis zum *COMMIT* gehalten
 - *COMMIT* wird atomar (d.h. nicht unterbrechbar) ausgeführt



RX-Sperrverfahren

- Bisher: kein paralleles Lesen oder Schreiben möglich
- Jetzt: Parallelität unter Lesern erlaubt
- 2 Arten von Sperren
 - Lesesperren oder **R-Sperren** (*read locks*)
 - Schreibsperren oder **X-Sperren** (*exclusive locks*)
- Verträglichkeit der Sperrentypen

		<i>bestehende Sperre</i>	
		R	X
<i>angeforderte Sperre</i>	R	+	-
	X	-	-

Serialisierungsreihenfolge bei RX

- RX-Sperrverfahren meist in Verbindung mit striktem 2PL um nur kaskadenfreie rücksetzbare Schedules zu erhalten
- Die Reihenfolge der Transaktionen im „äquivalenten seriellen Schedule“ ist die *Serialisierungsreihenfolge*.
- Bei RX-Sperrverfahren (in Verbindung mit striktem 2PL) wird die Serialisierungsreihenfolge durch die erste auftretende Konfliktoperation festgelegt.

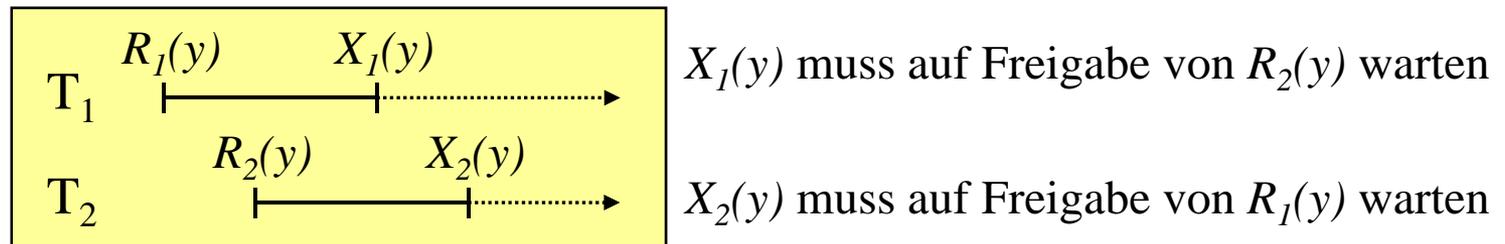
Serialisierungsreihenfolge bei RX

- Beispiel:
 - Situation:
 - T1 schreibt ein Objekt x
 - Danach möchte T2 x lesen
 - Folge:
 - T2 muss auf das COMMIT von T1 warten, d.h. der serielle Schedule enthält T1 vor T2.
 - Da T2 wartet, kommen auch alle weiteren Operationen erst nach dem COMMIT von T1.
 - Achtung:

Grundsätzlich sind zwar auch Abhängigkeiten von T2 nach T1 denkbar (z.B. auf einem Objekt y), diese würden aber zu einer Verklemmung (Deadlock, gegenseitiges Warten) führen.

RUX-Sperrverfahren

- Deadlockgefahr durch Sperrkonversionen (Umwandlung einer R -Sperrung in eine X -Sperrung)



- Lösung: Update-Sperren
 - U -Sperrung für Lesen mit Änderungsabsicht
 - Zur (späteren) Änderung des Objekts wird Konversion $U \rightarrow X$ vorgenommen
 - Erfolgt keine Änderung, kann Konversion $U \rightarrow R$ durchgeführt werden (Zulassen anderer Leser)

RUX-Sperrverfahren

- Verträglichkeit der Sperrentypen

		<i>bestehende Sperre</i>		
		<i>R</i>	<i>U</i>	<i>X</i>
<i>angeforderte Sperre</i>	<i>R</i>	+	-	-
	<i>U</i>	+	-	-
	<i>X</i>	-	-	-

- Kein Verhungern möglich, da spätere Leser keinen Vorrang haben
- Keine Konversionsverklemmung auf demselben Objekt
- Verklemmungen bzgl. verschiedener Objekte bleiben möglich

RAX-Sperrverfahren

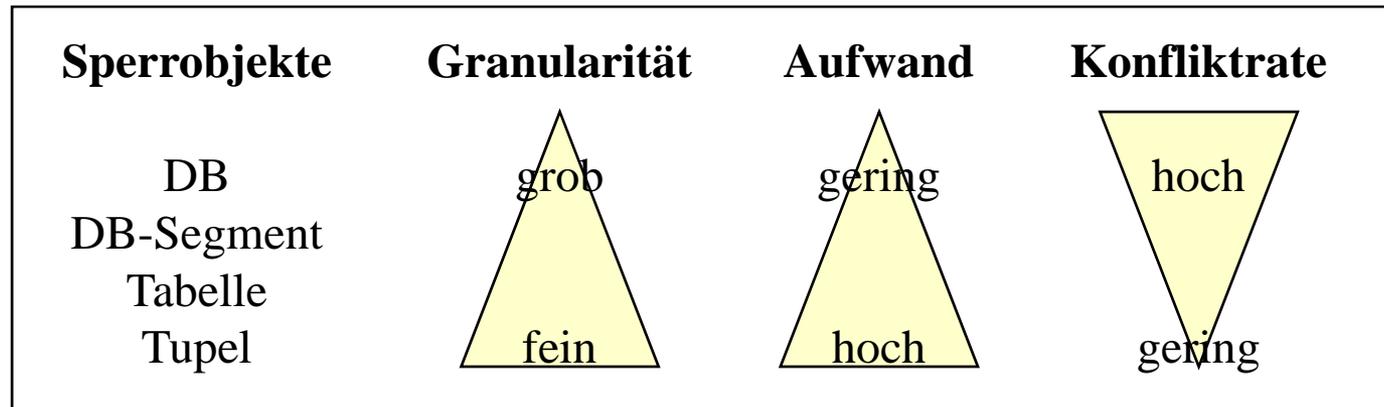
- Symmetrische Variante von RUX:
Bei gesetzter *A*-Sperrung wird weitere *R*-Sperrung erlaubt
- Verträglichkeit der Sperrentypen

		<i>bestehende Sperre</i>		
		<i>R</i>	<i>A</i>	<i>X</i>
<i>angeforderte Sperre</i>	<i>R</i>	+	+	-
	<i>A</i>	+	-	-
	<i>X</i>	-	-	-

- Beim Konvertierungswunsch $A \rightarrow X$ Verhungern möglich
- Tradeoff zwischen höherer Parallelität und Verhungern

Hierarchische Sperrverfahren

- Sperrgranularität bestimmt Parallelität / Aufwand



- Tradeoff
 - Geringe Konfliktrate ermöglicht hohen Parallelitätsgrad
 - Feine Granularität verursacht hohen Verwaltungsaufwand

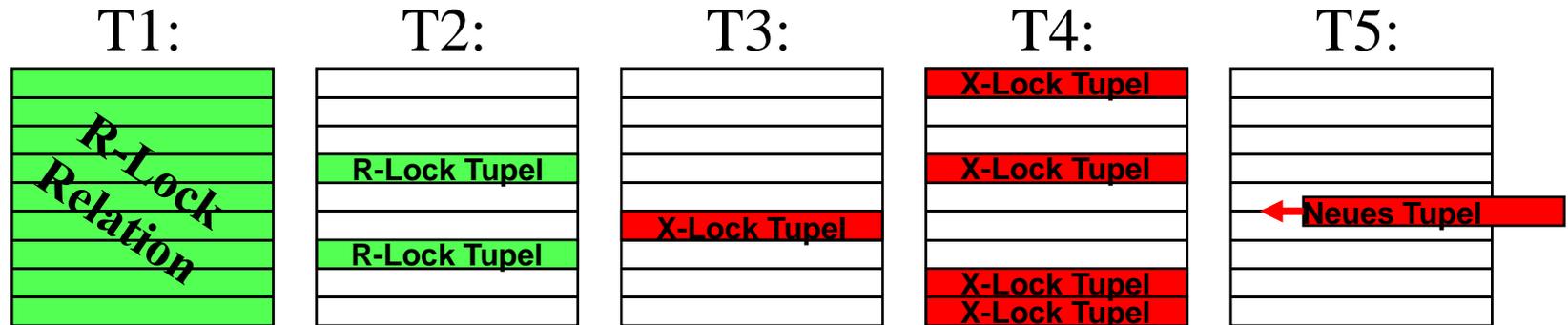
Hierarchische Sperrverfahren

- Lösung: variable Granularität durch hierarchische Sperren
- Kommerzielle DBS unterstützen zumeist 2-stufige Objekthierarchie, z.B. Segment-Seite oder Tabelle-Tupel
- Vorgehensweise bei hierarchischen Sperrverfahren:
 - Anwendung eines beliebigen Sperrprotokolls (z.B. RX) auf der fein-granularen Ebene (z.B. Tupel)
 - Zusätzlich Anwendung eines speziellen Protokolls (RIX) auf der grob-granularen Ebene (z.B. Relation)

Hierarchische Sperrverfahren

- Ziele von RIX:
 - Erkennung von Konflikten auf der Relationen-Ebene
 - Zusätzlich: *Effiziente* Erkennung der Konflikte zwischen den beiden *verschiedenen* Ebenen
 - Bei Anforderung einer Relationensperre soll vermieden werden, jedes einzelne Tupel auf eine Sperre zu überprüfen (wäre bei Tupelsperren erforderlich)
 - Trotzdem maximale Nebenläufigkeit von TAs, die nur mit einzelnen Tupeln arbeiten.

Hierarchische Sperrverfahren: Beispiel



- T2 kann (jeweils) mit T1, T3 oder T5 gleichzeitig arbeiten
- T3 und T4 können nicht mit T1 gleichzeitig arbeiten. Dies soll verhindert werden, ohne jedes einzelne Tupel auf Bestehen eines X-Lock zu überprüfen
- T2 und T4 können nicht gleichzeitig arbeiten, da sie unverträgliche Sperren auf *demselben* Tupel benötigen
- T1 und T5 können nicht gleichzeitig arbeiten (Phantomproblem!). Würden *nur* Tupelsperren verwendet, könnte dieser Konflikt nicht bemerkt werden

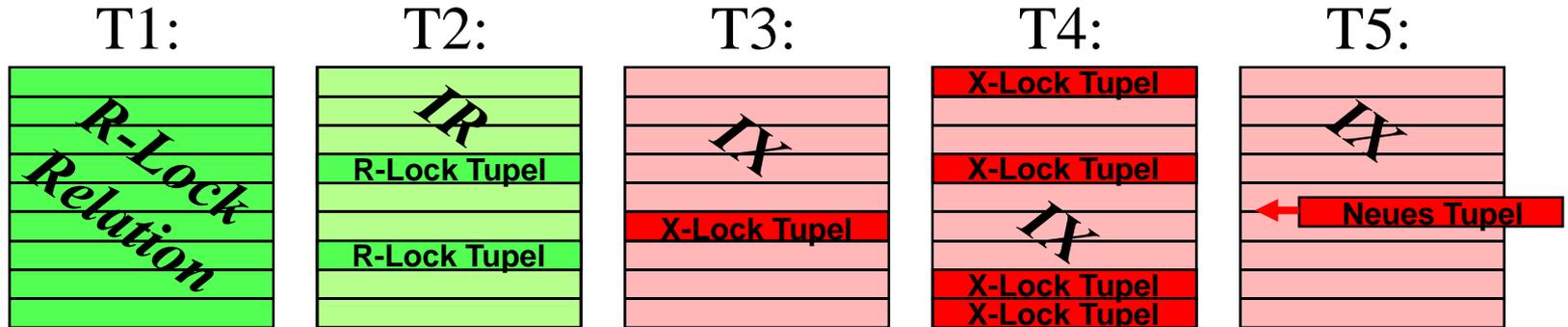
Hierarchische Sperrverfahren: Intentionssperren

- *IR-Sperre (intention read)*: auf feinerer Granularitätsstufe existiert (mindestens) eine *R-Sperre*
- *IX-Sperre (intention exclusive)*: auf feinerer Stufe *X-Lock*
- *RIX-Sperre (R-Sperre + IX-Sperre)*: volle Lesesperre und feinere Schreibsperre (sonst zu große Behinderung)
- Verträglichkeit der Sperrentypen:

		<i>bestehende Sperre</i>				
		<i>R</i>	<i>X</i>	<i>IR</i>	<i>IX</i>	<i>RIX</i>
<i>angeforderte Sperre</i>	<i>R</i>	+	-	+	-	-
	<i>X</i>	-	-	-	-	-
	<i>IR</i>	+	-	+	+	+
	<i>IX</i>	-	-	+	+	-
	<i>RIX</i>	-	-	+	-	-

Im markierten Bereich ist eine Überprüfung der Sperren auf der fein-granulareren Ebene zusätzlich erforderlich

Hierarchische Sperrverfahren: Beispiel



bestehende Sperre

		<i>R</i>	<i>X</i>	<i>IR</i>	<i>IX</i>	<i>RIX</i>
<i>angeforderte Sperre</i>	<i>R</i>	+	-	+	-	-
	<i>X</i>	-	-	-	-	-
	<i>IR</i>	+	-	+	+	+
	<i>IX</i>	-	-	+	+	-
	<i>RIX</i>	-	-	+	-	-

Mehrversionen-Sperren: RAC

– Prinzip:

- Änderungen erfolgen in lokalen Kopien im TA-Puffer
- A-Sperren zur Änderung erforderlich
- Bei *COMMIT* erfolgt Konvertierung von $A \rightarrow C$
- C-Sperre zeigt Existenz zweier gültiger Objektversionen V_{old} und V_{new} an
- C-Sperre wird erst freigegeben wenn letzter alter Leser fertig ist
- Zustand eines Objekts mit

- *R-Lock*: V_{old} oder V_{new} wird von ein oder mehreren TAs gelesen
- *A-Lock*: Objektversion wird im lokalen TA-Puffer zu V_{new} geändert, alle Leser sehen V_{old}
- *C-Lock*: Objekt wurde per *COMMIT* geändert
 - » neue Leser sehen V_{new}
 - » alte Leser sehen V_{old}

Mehrversionen-Sperren: RAC

– Verträglichkeit der Sperrentypen

		<i>bestehende Sperre</i>		
		<i>R</i>	<i>A</i>	<i>C</i>
<i>angeforderte Sperre</i>	<i>R</i>	+	+	+
	<i>A</i>	+	-	-
	<i>C</i>	+	-	-

Mehrversionen-Sperren: RAC

- Eigenschaften:
 - Leseanforderungen werden nie blockiert
 - Schreiber müssen bei gesetzter C-Sperre auf alle Leser der alten Version warten
 - Höherer Aufwand für Datensicherheit durch parallel gültige Versionen
 - Hoher Aufwand für Serialisierung (Abhängigkeits-beziehungen prüfen)

Konsistenzstufen

- Serialisierbare Abläufe gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs, erzwingen aber u.U. lange Blockierungszeiten paralleler Transaktionen
- Kommerzielle DBS unterstützen deshalb häufig schwächere Konsistenzstufen als die Serialisierbarkeit unter Inkaufnahme von Anomalien
- Schwächere Konsistenz tolerierbar z.B. für statistische Auswertungen
- Verschiedene Konzepte für Konsistenzstufen
 - Definition über Sperrentypen (“Konsistenzstufen” nach Jim Gray):
 - Definition über Anomalien (“Isolation Levels” in SQL92)

Konsistenzstufen nach J. Gray

- Definition über die Dauer der Sperren:
 - lange Sperren: werden bis EOT gehalten
 - kurze Sperren: werden nicht bis EOT gehalten

	Schreibsperre	Lesesperre
Konsistenzstufe 0	kurz	-
Konsistenzstufe 1	lang	-
Konsistenzstufe 2	lang	kurz
Konsistenzstufe 3	lang	lang

Konsistenzstufen nach J. Gray

– Konsistenzstufe 0

- ohne Bedeutung, da Dirty Write und Lost Update möglich

– Konsistenzstufe 1

- kein Dirty Write mehr, da Schreibsperrern bis EOT
- Dirty Read möglich, da keine Lesesperren

– Konsistenzstufe 2

- praktisch sehr relevant
- kein Dirty Read mehr, da Lesesperren
- Non-Repeatable Read möglich, da zwischen zwei Lesevorgängen eine andere TA das Objekt ändern kann
- Lost Update möglich, da nur kurze Lesesperren (kann durch *Cursor Stability* verhindert werden)

Konsistenzstufen nach J. Gray

– Konsistenzstufe 3

- entspricht strengem 2PL, Serialisierbarkeit ist gewährleistet
- Non-Repeatable Read und Lost Update werden verhindert

– **Cursor Stability** (Modifikation von Konsistenzstufe 2)

- Lesesperren bleiben solange bestehen, bis der Cursor zum nächsten Objekt übergeht
- (Mögliche) Änderungen am aktuellen Objekt können nicht verloren gehen
- Nachteil: Anwendungsprogrammierer hat Verantwortung für korrekte Synchronisation

Isolation Levels in SQL92

- Je länger ein “Read”-Lock bestehen bleibt, desto eher ist die Transaktion “isoliert” von anderen
- Definition der “Isolation Levels ” über erlaubte Anomalien

Isolation Level	Lost Update	Dirty Read	Non-Rep. Read	Phantom
READ UNCOMMITTED	-	+	+	+
READ COMMITTED	-	-	+	+
REPEATABLE READ	-	-	-	+
SERIALIZABLE	-	-	-	-

- Lost Update ist immer ausgeschlossen
- SQL-Anweisung

```
SET TRANSACTION ISOLATION LEVEL <level>
```

(Default <level> ist SERIALIZABLE)

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

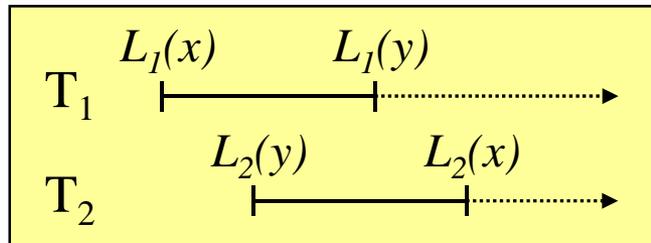
3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

Verklemmung (Deadlock)

- Zwei Transaktionen warten gegenseitig auf die Freigabe einer Sperre (L)
- Beispiel: $L_1(x)$, $L_2(y)$, $L_1(y)$, $L_2(x)$



T_1 wartet auf Freigabe von y durch T_2

T_2 wartet auf Freigabe von x durch T_1

Verklemmung (Deadlock)

- Voraussetzungen für das Auftreten von Verklemmungen (vgl. Betriebssysteme)
 - Datenbankobjekte sind zugriffsbeschränkt
 - Sperren auf bereits gelesenen oder geschriebenen Objekten sind nicht entziehbar
 - TAs sperren nicht alle Objekte gleichzeitig, sondern fordern Sperren nach und nach an
 - TAs sperren Objekte in beliebiger Reihenfolge
 - TAs warten auf Sperrenfreigabe durch andere TAs, ohne selbst Sperren freizugeben

Erkennen und Auflösen von Deadlocks

– Time-Out Strategie

- Falls eine TA innerhalb einer Zeiteinheit t keinen Fortschritt macht, wird sie als verklemmt betrachtet und zurückgesetzt
- t zu klein: TAs werden u.U. beim Warten auf Ressourcen abgebrochen
- t zu groß: Verklemmungszustände werden zu lange geduldet

– Wartegraphen

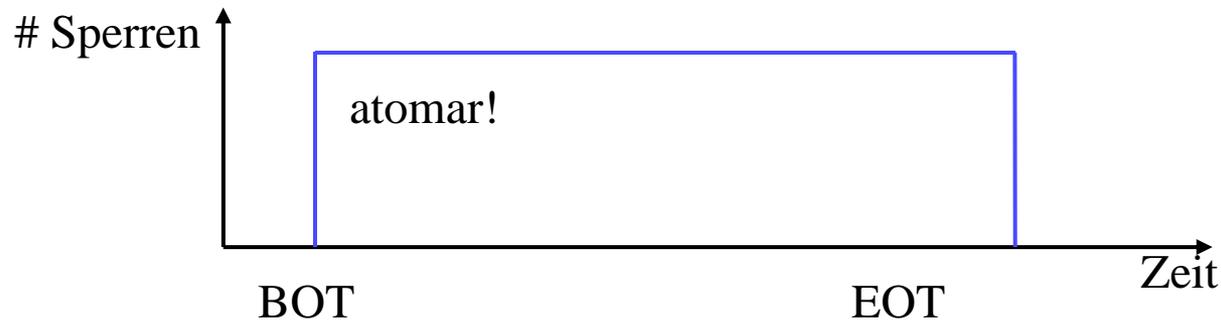
- Knoten des Wartegraphen sind TAs, Kanten sind die Wartebeziehungen
- Verklemmung liegt vor, wenn Zyklen im Wartegraph auftreten
- Zyklen können eine Länge > 2 haben (ist in der Praxis untypisch)
- Die Verwaltung von Wartegraphen ist für die Praxis zu aufwändig

Erkennen und Auflösen von Deadlocks

- Strategien zur Auflösung von Verklemmungen durch Rücksetzen beteiligter TAs:
 - **Minimierung des Rücksetzaufwands:** Wähle jüngste TA oder TA mit den wenigsten Sperren aus
 - **Maximierung der freigegebenen Ressourcen:** Wähle TA mit den meisten Sperren aus, um die Gefahr weiterer Verklemmungen zu verkleinern
 - **Mehrfache Zyklen:** Wähle TA aus, die an mehreren Zyklen beteiligt ist
 - **Vermeidung der Aushungerung (Starvation):** Setze früher bereits zurückgesetzte TAs möglichst nicht noch einmal zurück

Vermeidung von Deadlocks: Preclaiming

- **Preclaiming:** alle Sperrenanforderungen werden zu Beginn einer TA gestellt



- **Vorteile**

- sehr einfache und effektive Methode zur Vermeidung von Deadlocks
- keine Rücksetzungen zur Auflösung von Deadlocks nötig
- in Verbindung mit strengem 2PL wird kaskadierendes Rücksetzen vermieden

Vermeidung von Deadlocks: Preclaiming

– Nachteile

- benötigte Sperren sind bei BOT i. a. noch nicht bekannt, z.B. bei
 - interaktiven TAs
 - Fallunterscheidungen in TAs
 - dynamischer Bestimmung der gesperrten Objekte
- z. T. Abhilfe durch Sperren einer Obermenge der tatsächlich benötigten Objekte:
 - unnötige Ressourcenbelegung
 - Einschränkung der Parallelität

Vermeidung von Deadlocks: Ordnung

– Prinzip

- Auf den Datenbank-Objekten wird eine totale Ordnung definiert, z.B. Relation $R1 < R2 < R3 < \dots$
- Annahme: Es gibt nur eine Sperren-Art (X).
- Es wird festgelegt, dass Sperren nur in aufsteigender Reihenfolge (bezüglich dieser Ordnung) vergeben werden (ggf. werden nicht benötigte Objekte mit gesperrt).

– Eigenschaften

- Gegenseitiges Warten ist nicht mehr möglich.
- Szenario ist ähnlich restriktiv wie Preclaiming.
- Für Spezial-Anwendungen ist die Definition einer Ordnung auf den DB-Objekten durchaus denkbar.

Vermeidung von Deadlocks: Zeitstempel

– Prinzip

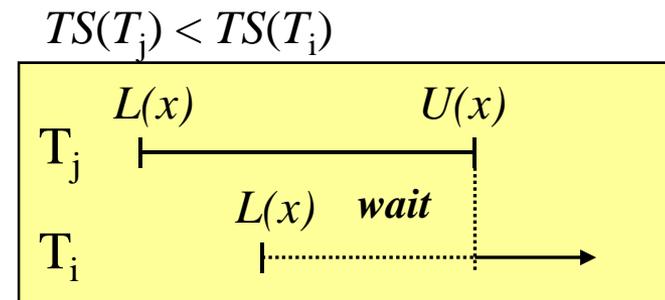
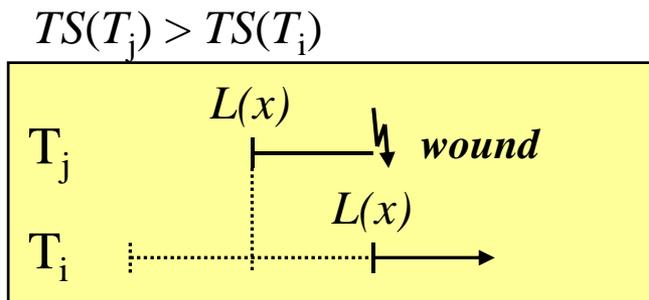
- Jeder Transaktion T_i wird zu Beginn ein Zeitstempel $TS(T_i)$ zugeordnet (*Time Stamp*):
 - Beginn (BoT) einer TA
 - oder (besser) erste Datenbank-Operation einer TA
- Objekte tragen nach wie vor Sperren
- TAs warten nicht bedingungslos auf die Freigabe von Sperren
- In Abhängigkeit von den Zeitstempeln werden TAs im Konfliktfall zurückgesetzt
- Zwei Strategien, falls T_i auf Sperre von T_j trifft:
 - *wound-wait*
 - *wait-die*

Vermeidung von Deadlocks: Zeitstempel

– wound-wait:

T_i fordert Sperre $L(x)$ an.

- Jüngere TA T_j , d.h. $TS(T_j) > TS(T_i)$, hält bereits Sperre auf x :
=> T_i läuft weiter, jüngere TA T_j wird zurückgesetzt (**wound**)
- Ältere TA T_j , d.h. $TS(T_j) < TS(T_i)$, hält bereits Sperre auf x :
=> T_i wartet auf Freigabe der Sperre durch ältere TA T_j (**wait**)



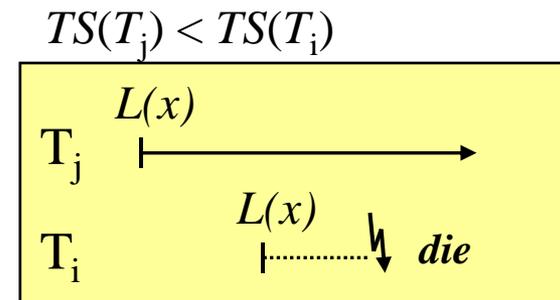
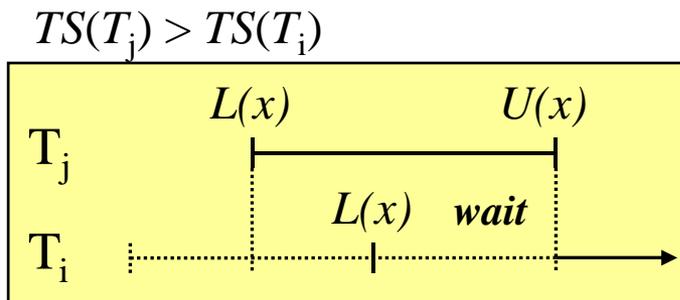
→ ältere TAs „bahnen“ sich ihren Weg durch das System

Vermeidung von Deadlocks: Zeitstempel

– wait-die:

T_i fordert Sperre $L(x)$ an.

- Jüngere TA T_j , d.h. $TS(T_j) > TS(T_i)$, hält bereits Sperre auf x :
=> T_i wartet auf Freigabe der Sperre durch jüngere TA T_j (**wait**)
- Ältere TA T_j , d.h. $TS(T_j) < TS(T_i)$, hält bereits Sperre auf x :
=> T_i wird zurückgesetzt (**die**), ältere TA T_j läuft weiter



→ ältere TAs müssen zunehmend mehr warten

Vermeidung von Deadlocks: Zeitstempel

- Wound-Wait ist Deadlock-frei
 - Die Zeitstempel („Alter der Transaktion“) definieren eine strikte, totale Ordnung auf den Transaktionen:
 - $TS(T_1) < TS(T_2) < TS(T_3) < \dots < TS(T_n)$
 - Bei der Wound-Wait-Strategie warten jüngere auf ältere Transaktionen, aber nie umgekehrt:
 - T_i wartet auf $T_j \Rightarrow TS(T_j) < TS(T_i)$
 - Wird ein Wartegraph gezeichnet, in dem die Transaktionen nach Alter geordnet sind (die älteste zuerst), so gehen Kanten niemals von links nach rechts):



- Somit ist kein Zyklus möglich

Vermeidung von Deadlocks: Zeitstempel

- Wound-Wait ist serialisierbar
 - Die Serialisierbarkeit der durch Wound-Wait zugelassenen Schedules ergibt sich aus den Sperren:
 - Sperren nach dem RX-Protokoll (o.ä.) werden beachtet
 - Strenges 2-Phasen-Sperrprotokoll
 - Rücksetzungen wesentlich häufiger als nötig
- Wait-Die: Analog
 - Pfeile im Wartegraphen nie von rechts nach links

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

Motivation

– Synchronisation mit Sperren

- *pessimistische* Annahme: Konflikte möglich / treten (oft) auf
- Vorgehen: *Verhinderung* von Konflikten
- Methode: *Blockierung* von Transaktionen
 - reale Gefahr von Verklemmungen
 - Sperrenverwaltung ist sehr aufwändig
 - mögliche Leistungseinbußen durch lange Wartezeiten

– Nicht-sperrende Synchronisation

- *optimistische* Annahme: Konflikte sind seltene Ereignisse
- Vorgehen: *Auflösung* von Konflikten
- Methode: *Rücksetzen* von Transaktionen
 - keine Verklemmungen
 - aufwändige Konfliktprävention wird eingespart
 - mögliche Leistungseinbußen durch häufige Rücksetzungen

Zeitstempel statt Sperren

– Motivation:

- Zeitstempel nicht zur Verklemmungsvermeidung sondern zur Synchronisation ohne Sperren
- Zählt zu den pessimistischen Sperrverfahren

– Idee:

- Jede TA bekommt zu BOT einen Zeitstempel
- Hierdurch Definition des äquivalenten seriellen Schedule
- Bei jedem Zugriff: Test, ob Verletzung des äquivalenten seriellen Schedules
- Keine Sperren, sondern über Zeitstempel auf Objekten

Zeitstempel statt Sperren (cont.)

– Beispiel:

äquivalenter serieller Schedule:

älteste T_1 T_2 T_3 T_4 T_5 jüngste TA

– Welche Zugriffe müssen verhindert werden?

- Bei Lesezugriff von T_3 auf ein Objekt x :
 - Lesezugriff ist verboten, wenn *vorher* T_4/T_5 x geschrieben hat
 - *nachher* dürfen T_1 und T_2 das Objekt x nicht mehr schreiben
- Bei Schreibzugriff durch T_3 auf ein Objekt x :
 - T_4/T_5 dürfen x nicht *vorher* gelesen oder geschrieben haben
 - T_1/T_2 dürfen x nicht *nachher* das Objekt lesen oder schreiben

Zeitstempel statt Sperren (cont.)

- Nicht nur Transaktionen, sondern auch Objekte O tragen Zeitstempel:
 - $readTS(O)$: Zeitstempel der jüngsten TA, die das Objekt O gelesen hat.
 - $writeTS(O)$: Zeitstempel der jüngsten TA, die das Objekt O geschrieben hat.
- Prüfungen beim **Lesezugriff** von T_i auf ein Objekt O :

- Falls $TS(T_i) < writeTS(O)$:
 T_i ist älter als die TA, die O geschrieben hat $\rightarrow T_i$ zurücksetzen
- Falls $TS(T_i) \geq writeTS(O)$:
 T_i ist jünger als die TA, die O geschrieben hat $\rightarrow T_i$ darf O lesen,
Lesemarke wird aktualisiert: $readTS(O) = \max(TS(T_i), readTS(O))$

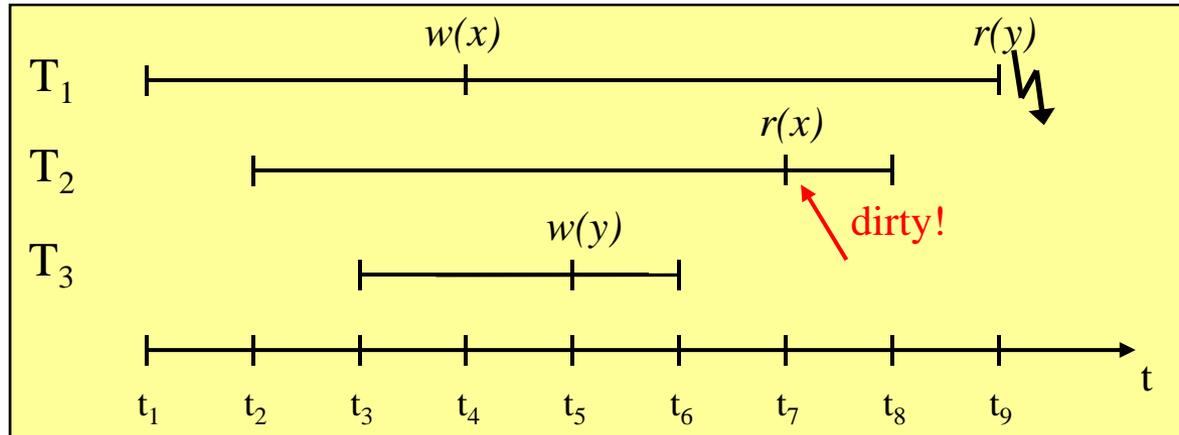
Zeitstempel statt Sperren (cont.)

– Prüfungen beim **Schreibzugriff** von T_i auf ein Objekt O :

- Falls $TS(T_i) < readTS(O)$:
 T_i ist älter als eine TA, die O gelesen hat $\Rightarrow T_i$ zurücksetzen
- Falls $TS(T_i) < writeTS(O)$:
 T_i ist älter als die TA, die O geschrieben hat $\Rightarrow T_i$ zurücksetzen
- Sonst:
 T_i darf O schreiben,
Schreibmarke wird aktualisiert: $writeTS(O) = TS(T_i)$

Zeitstempel statt Sperren (cont.)

– Beispiel:



Seien $writeTS(x)$, $writeTS(y)$, $readTS(x)$ und $readTS(y)$ kleiner als t_1

- $t_1: TS(T_1) = t_1$ $t_2: TS(T_2) = t_2$ $t_3: TS(T_3) = t_3$
- $t_4: write(x)$ in T_1 : Da $TS(T_1) > readTS(x)$ darf T_1 auf x schreiben, dann: $writeTS(x) := t_4$
- $t_5: write(y)$ in T_3 : Da $TS(T_3) > readTS(y)$ darf T_3 auf y schreiben, dann: $writeTS(y) := t_5$
- t_6 : keine Prüfung bei $COMMIT$ von T_3
- $t_7: read(x)$ in T_2 : Da $TS(T_2) \geq writeTS(x)$ darf T_2 auf x lesen, dann: $readTS(x) := t_7$
- t_8 : keine Prüfung bei $COMMIT$ von T_2 (eigenes Problem mit dirty read siehe unten)
- $t_9: read(y)$ in T_1 : Da $TS(T_1) < writeTS(y)$ wird T_1 zurückgesetzt. Die von t_9 geänderten Zeitstempel müssen ebenfalls zurückgesetzt werden.

Zeitstempel statt Sperren (cont.)

- Problem mit Dirty Read
 - im Beispiel: T_2 liest x , obwohl T_1 noch kein *COMMIT* hatte
 - geänderte, aber noch nicht festgeschriebene Daten müssen noch gegen Lesen bzw. Überschreiben gesichert werden (z.B. durch *dirty*-Bit) → damit aber wieder Deadlocks möglich
- Auswirkungen
 - Methode garantiert Serialisierbarkeit bis auf Dirty Read
 - es treten keine Deadlocks auf (möglicherweise jedoch durch *dirty*-Bit)
 - äquivalente serielle Reihenfolge entspricht den Zeitstempeln der TAs

Zeitstempel statt Sperren (cont.)

- Nachteil für lange TAs
 - Rücksetzgefahr steigt mit Dauer der TA
 - Verhungern von TAs durch wiederholtes Zurücksetzen wird nicht verhindert

- Bewertung
 - Verwaltung der Objektmarken ist sehr aufwändig und häufig nicht feingranularer als auf Seitenebene praktikabel
 - Zeitstempel müssen für jedes Objekt verwaltet werden, während Sperren nur bei Zugriff auf Objekte angelegt werden

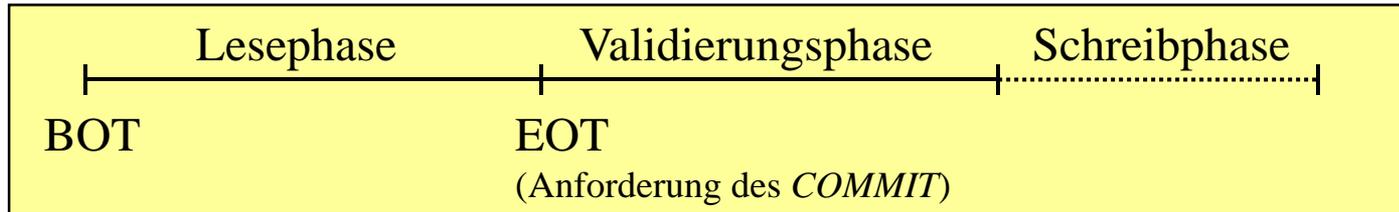
Optimistische Synchronisation

– Konzept

- Keine Konfliktprävention
- Konflikte werden erst bei *COMMIT* festgestellt
- Im Konfliktfall werden Transaktionen zurückgesetzt
- nahezu beliebige Parallelität, da TAs nicht blockiert werden

Optimistische Synchronisation (cont.)

– Drei Phasen einer TA



- *Lesephase*:
eigentliche TA-Verarbeitung, Änderungen nur im lokalen TA-Puffer
- *Validierungsphase*:
Prüfung, ob die abzuschließende TA mit nebenläufigen TAs in Konflikt geraten ist; im Konfliktfall wird die TA zurückgesetzt
- *Schreibphase*:
nach erfolgreicher Validierung werden die Änderungen dauerhaft gespeichert
- Schreiben und Validieren muss atomar durchgeführt werden.

Optimistische Synchronisation (cont.)

– Validierungstechniken

- Für jede Transaktion T_i werden zwei Mengen geführt:
 - $RS(T_i)$: die von T_i gelesenen Objekte (*Read Set*)
 - $WS(T_i)$: die von T_i geschriebenen Objekte (*Write Set*)
- Konflikterkennung
 - Konflikt zwischen T_i und T_j liegt vor, wenn $WS(T_i) \cap RS(T_j) \neq \emptyset$
 - Annahme: $WS(T_i) \subseteq RS(T_i)$, d.h. jedes Objekt wird vor dem Schreiben gelesen (neue Objekte müssen nicht in die Lesemenge eingetragen werden). Motivation für die Annahme: Objekte werden erst in den TA-Puffer gelesen und dort verändert.
- Zwei Validierungsstrategien
 - *Backward-Oriented Optimistic Concurrency Control (BOCC)*: Validierung nur gegenüber bereits beendeten TAs
 - *Forward-Oriented Optimistic Concurrency Control (FOCC)*: Validierung nur gegenüber noch laufenden TAs

Optimistische Synchronisation (cont.)

– Bemerkungen

- Serialisierungsreihenfolge ist durch Validierungsreihenfolge gegeben
- Validierung und Schreiben muss sequenziell und atomar durchgeführt werden

BOCC

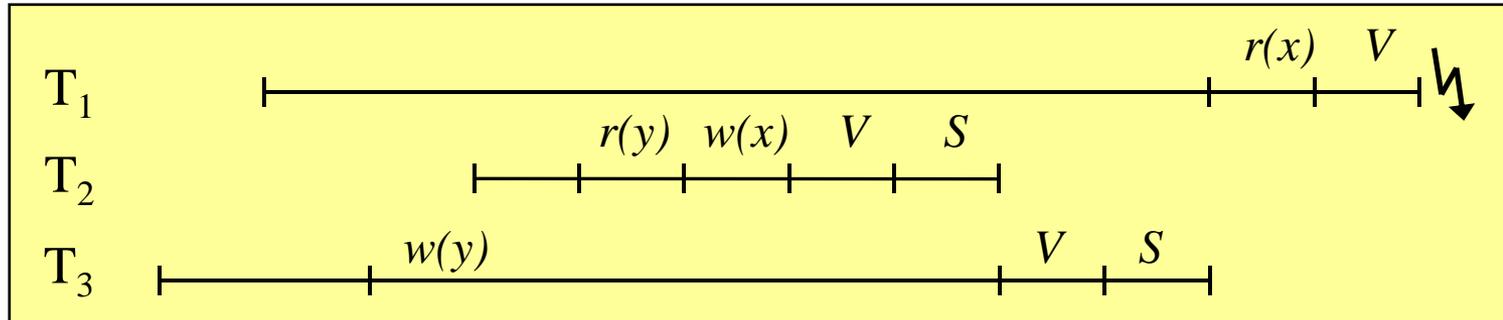
- Validierung von T_i
 - “Wurde eines der während der Lese-Phase von T_i gelesenen Objekte von einer anderen (bereits beendeten) Transaktion T_j geändert?”
 - D.h. Read-Set $RS(T_i)$ wird mit allen Write-Sets $WS(T_j)$ von Transaktionen T_j verglichen, die während der Lese-Phase von T_i validiert haben
- Algorithmus

```
VALID := true;  
for (alle während Ausführung von  $T_i$  beendeten  $T_j$ ) do  
    if  $RS(T_i) \cap WS(T_j) \neq \emptyset$  then VALID := false;  
end;  
if VALID then Schreiphase( $T_i$ ); Commit ( $T_i$ );  
    else Rollback( $T_i$ ); // Nothing to do
```

BOCC (cont.)

– Beispiel

V: Validierung
S: Schreibphase



– Ablauf

- T_2 wird erfolgr. validiert, da es noch keine validierten TAs gibt
 - T_3 wird erfolgreichvalidiert, da für $y \in WS(T_3) \subseteq RS(T_3)$ gilt:
 $y \notin WS(T_2)$
 - T_1 steht wegen $x \in WS(T_2)$ in Konflikt mit T_2 und wird abgebrochen
- Zurücksetzen war unnötig, da T_1 bereits die aktuelle Version von x gelesen hat.

BOCC (cont.)

– Abhilfe: BOCC+

- Objekte bekommen Änderungszähler oder Versionsnummern
- TAs werden nur zurückgesetzt, wenn sie tatsächlich veraltete Daten gelesen haben

– Nachteile für lange TAs

- Verhungern von Transaktionen wird nicht verhindert
- Anzahl der zu vergleichenden Write-Sets steigt mit TA-Dauer
- TAs mit großen Read-Sets können in viele Konflikte geraten
- spätes Zurücksetzen erst bei der Validierung verursacht hohen Arbeitsverlust

FOCC

- Validierung von T_i
 - “Wurde eines der von T_i geänderten Objekte von einer anderen (noch laufenden) Transaktion T_j gelesen?”
 - D.h. Write-Set $WS(T_i)$ wird mit allen Read-Sets $RS(T_j)$ von Transaktionen T_j verglichen, die sich gerade in der Lesephase befinden
- Algorithmus

```
VALID := true;  
for (alle laufenden  $T_j$ ) do  
    if  $WS(T_i) \cap RS(T_j) \neq \emptyset$  then VALID := false;  
end;  
if VALID then Schreiphase( $T_i$ ) ; commit ( $T_i$ );  
    else löse Konflikt auf;
```

FOCC (cont.)

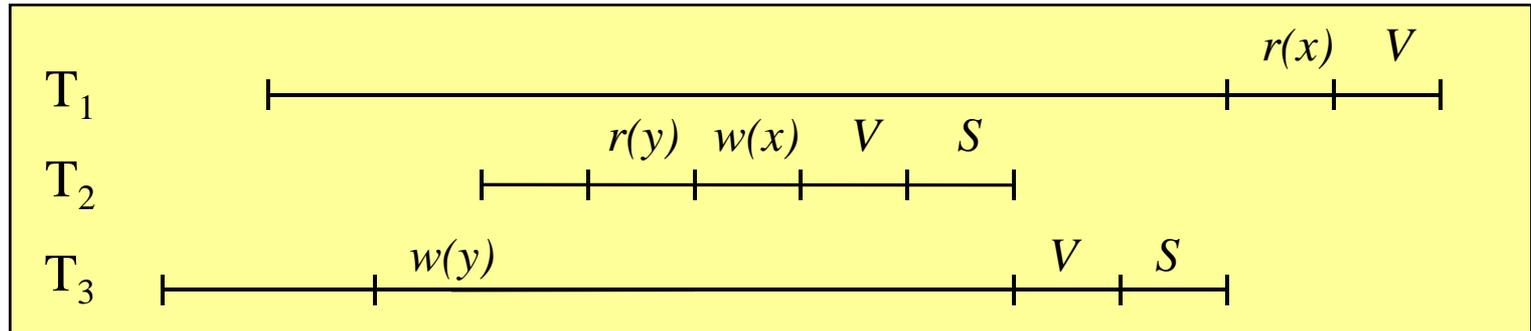
– Bewertung

- Validierung muss nur von ändernden Transaktionen durchgeführt werden.
- die überflüssigen Rücksetzungen von BOCC werden vermieden
- überflüssige Rücksetzungen wegen der vorgegebenen Serialisierungs-Reihenfolge sind weiterhin möglich
- mehr Freiheiten bei der **Konfliktauflösung**: beliebige TA kann abgebrochen werden, z.B.
 - **Kill**-Ansatz: die noch laufenden TAs werden abgebrochen.
 - **Die**-Ansatz: die validierende TA wird abgebrochen (“stirbt”).
 - Verhindern von Verhungerung: z.B. Anzahl der Rücksetzungen einer TA beachten.

FOCC (cont.)

– Beispiel

V: Validierung
S: Schreibphase



– Ablauf:

- T_2 wird erfolgr. validiert, da x noch von keiner TA gelesen wurde
- T_3 wird erfolgreich validiert, da y von keiner (noch) laufenden TA gelesen wurde
- T_1 ist eine reine Lese-TA und muss nicht validiert werden
- Hätte T_2 das Objekt y auch geändert, so wäre der Konflikt mit T_3 bei der Validierung von T_2 erkannt worden, und eine der beiden TAs hätte abgebrochen werden müssen

Abschließende Bemerkungen

– Scheduler

- Einheit im DBMS
- Bestimmt die Ausführungs-Reihenfolge der einzelnen Operationen verschiedener Transaktionen
- Der resultierende Schedule sollte „vernünftig“ sein
 - Mindestanforderung: Serialisierbarkeit, Rücksetzbarkeit
 - Meist zusätzliche Anforderung: kein kaskadierende Rollbacks
- Verwendet pessimistische (Locking, Zeitstempel) oder optimistische Verfahren

Abschließende Bemerkungen (cont.)

– Qualitätsmerkmale von Synchronisationsverfahren

- Effektivität: Serialisierbarkeit, Vermeidung von Anomalien
- Parallelitätsgrad (Blockierung nebenläufiger TAs)
- Verklemmungsgefahr
- Häufigkeit von Rücksetzungen; Vermeidung überflüssiger Rücksetzungen
- Benachteiligung bestimmter (z.B. langer) TAs (“Verhungern”) durch lange Blockierungen oder häufige Rücksetzungen
- Verwaltungsaufwand für die Synchronisation (Sperren, Zeitstempel, ...)

Abschließende Bemerkungen (cont.)

– Praktische Bewertung

- Oft Implementierungsprobleme für feinere Granul. als DB-Seiten
- Kombinationen der Verfahren teilweise möglich (z.B. “*Optimistic Locking*”, IMS Fast Path)
- Synchronisation von Indexstrukturen als eigenes Problem
- nahezu alle kommerziellen DBS setzen auf Sperrverfahren