
Kapitel 3

Synchronisation

Vorlesung: PD Dr. Peer Kröger

Dieses Skript basiert auf den Skripten zur Vorlesung Datenbanksysteme II an der LMU München von Prof. Dr. Christian Böhm (Sommersemester 2007), PD Dr. Peer Kröger (Sommersemester 2008) und PD Dr. Matthias Schubert (Sommersemester 2009)

http://www.dbs.ifi.lmu.de/cms/Datenbanksysteme_II_12

3 Synchronisation

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

3.1 Einleitung

Synchronisation (Concurrency Control)

- Serielle Ausführung von Transaktionen
 - unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist
 - Folgen: niedriger Durchsatz, hohe Wartezeiten
- Mehrbenutzerbetrieb
 - führt i.A. zu einer besseren Auslastung des Systems (z.B. Wartezeiten bei E/A-Vorgängen können zur Bearbeitung anderer Transaktionen genutzt werden)
- Aufgabe der Synchronisation
 - Gewährleistung des *logischen Einbenutzerbetriebs*, d.h. innerhalb einer TA ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen

3

3.1 Einleitung

Anomalien im Mehrbenutzerbetrieb

- Klassifikation
 - Verloren gegangene Änderungen (*Lost Updates*)
 - Zugriff auf „schmutzige“ Daten (*Dirty Read / Dirty Write*)
 - Nicht-reproduzierbares Lesen (*Non-Repeatable Read*)
 - Phantomproblem
- Beispiel: Flugdatenbank

Passagiere	FlugNr	Name	Platz	Gepäck
	LH745	Müller	3A	8
	LH745	Meier	6D	12
	LH745	Huber	5C	14
	BA932	Schmidt	9F	9
	BA932	Huber	5C	14

4

3.1 Einleitung

Lost Updates

- Änderungen einer TA können durch Änderungen anderer TA überschrieben werden und dadurch verloren gehen
- Bsp.: Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus
 - T1: `UPDATE Passagiere SET Gepäck = Gepäck+3
WHERE FlugNr = LH745 AND Name = „Meier“;`
 - T2: `UPDATE Passagiere SET Gepäck = Gepäck+5
WHERE FlugNr = LH745 AND Name = „Meier“;`

5

3.1 Einleitung

Lost Updates (cont.)

- Möglicher Ablauf

T1	T2
<code>read(Passagiere.Gepäck, x1); x1 := x1+3; write(Passagiere.Gepäck, x1);</code>	<code>read(Passagiere.Gepäck, x2); x2 := x2 + 5; write(Passagiere.Gepäck, x2);</code>

- In der DB ist nur die Änderung von T1 wirksam, die Änderung von T2 ist verloren gegangen
→ Verstoß gegen *Durability*

6

3.1 Einleitung

Dirty Read / Dirty Write

- Zugriff auf „schmutzige“ Daten, d.h. auf Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden
- Beispiel:
 - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
 - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen

7

3.1 Einleitung

Dirty Read / Dirty Write

- Möglicher Ablauf:

T1	T2
UPDATE Passagiere SET Gepäck = Gepäck+3; ROLLBACK;	UPDATE Passagiere SET Gepäck = Gepäck+5; COMMIT;

- Durch Abbruch von T1 werden die geänderten Werte ungültig, die T2 gelesen hat (*Dirty Read*). T2 setzt weitere Änderungen darauf auf (*Dirty Write*)
 - Verstoß gegen
 - *Consistency*: Ablauf verursacht inkonsistenten DB-Zustand
oder
 - *Durability*: T2 muss zurückgesetzt werden

8

3.1 Einleitung

Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Beispiel:
 - T1 liest das Gepäckgewicht der Passagiere auf Flug BA932 zwei mal
 - T2 bucht den Platz 3F auf dem Flug BA932 für Passagier Meier mit 5kg Gepäck

9

3.1 Einleitung

Non-Repeatable Read

- Möglicher Ablauf:

T1	T2
SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;	INSERT INTO Passagiere VALUES (BA932, Meier, 3F, 5); COMMIT;
SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;	

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl T1 den DB-Zustand nicht geändert hat

→ Verstoß gegen *Isolation*

10

3.1 Einleitung

Phantomproblem

- Ausprägung (Spezialfall) des nicht-reproduzierbaren Lesens, bei der neu generierte Daten, sowie meist bei der 2. TA Aggregat-Funktionen beteiligt sind
- Bsp.:
 - T1 druckt die Passagierliste sowie die Anzahl der Passagiere für den Flug LH745
 - T2 bucht den Platz 7D auf dem Flug LH745 für Phantomas

11

3.1 Einleitung

Phantomproblem

- Möglicher Ablauf

T1	T2
<pre>SELECT * FROM Passagiere WHERE FlugNr = „LH745“; SELECT COUNT(*) FROM Passagiere WHERE FlugNr = „ LH745“;</pre>	<pre>INSERT INTO Passagiere VALUES (LH745, Phantomas, 7D, 2); COMMIT;</pre>

- Für Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere berücksichtigt ist

12

3 Synchronisation

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

13

3.2 Serialisierbarkeit von Transaktionen

Motivation

- Bearbeitung von Transaktionen
 - Nebenläufigkeit vor den Benutzern verbergen
 - Transparent für den Benutzer, als ob
TAs (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden
und NICHT als ob
TAs ineinander verzahnt ablaufen und sich dadurch (unbeabsichtigt) beeinflussen

14

3.2 Serialisierbarkeit von Transaktionen

Schedules

- Allgemeiner Schedule:
Ein Schedule („Historie“) für eine Menge $\{T_1, \dots, T_n\}$ von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der Transaktionen T_i entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.
- Frage: Warum darf die Reihenfolge der Aktionen innerhalb einer TA nicht verändert werden?

15

3.2 Serialisierbarkeit von Transaktionen

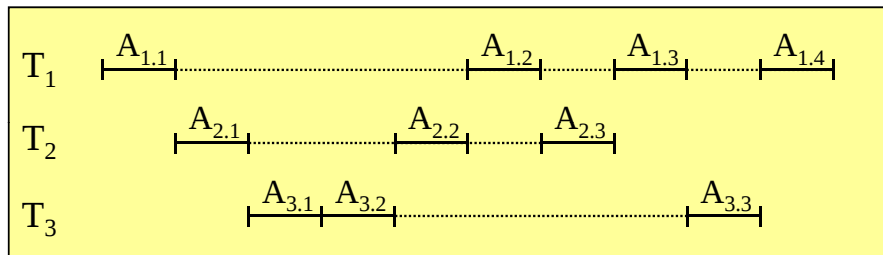
- Serieller Schedule:
Ein serieller Schedule ist ein Schedule S von $\{T_1, \dots, T_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden.
- Serialisierbarer Schedule:
Ein Schedule S von $\{T_1, \dots, T_n\}$ ist serialisierbar, wenn er dieselbe Wirkung hat wie ein beliebiger serieller Schedule von $\{T_1, \dots, T_n\}$.
- Nur serialisierbare Schedules dürfen zugelassen werden!

16

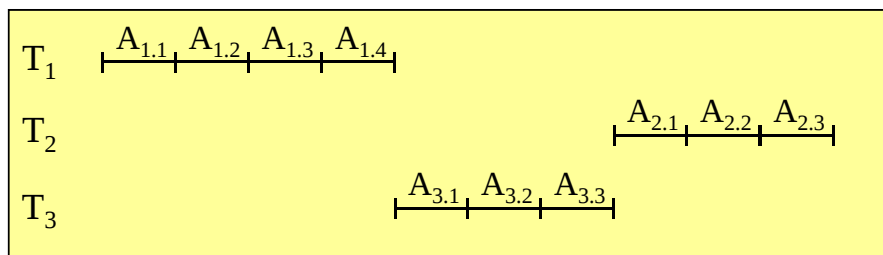
3.2 Serialisierbarkeit von Transaktionen

- Beispiele:

- Beliebiger Schedule:



- Serieller Schedule:



17

3.2 Serialisierbarkeit von Transaktionen

Wirkung von Schedules

- Frage: Wann haben zwei Schedules S1 und S2 die gleiche Wirkung auf den Datenbank-Inhalt?
- Achtung:
 - Gleiches Ergebnis kann u.a. Ergebnis eines Zufalls sein
 - Dies könnte aber nur durch nachträgliches Überprüfen der Datenbank-Zustände nach S1 und S2 festgestellt werden.

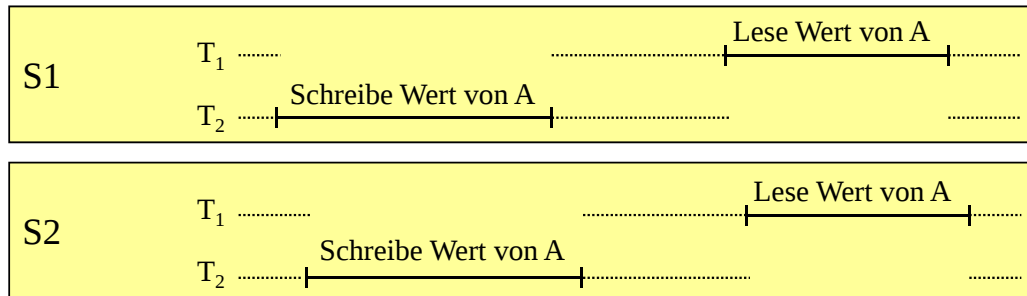
18

3.2 Serialisierbarkeit von Transaktionen

- Wir benötigen ein objektivierbares Kriterium:

Konflikt-Äquivalenz

- Idee:** Wenn in S1 eine Transaktion T_1 z.B. einen Wert liest, den T_2 geschrieben hat, dann muss das auch in S2 so sein.



- Wir sprechen hier von einer Schreib-Lese-Abhängigkeit (bzw. Konflikt) zwischen T_2 und T_1 (in Schedule S1 und S2)

19

3.2 Serialisierbarkeit von Transaktionen

Abhängigkeiten

Sei S ein Schedule. Wir sprechen von einer

- Schreib-Lese-Abhängigkeit** von $T_i \rightarrow T_j$
 - Es existiert Objekt x , so dass in S $w_i(x)$ vor $r_j(x)$ kommt
 - Abkürzung: $wr_{ij}(x)$
- Lese-Schreib-Abhängigkeit** von $T_i \rightarrow T_j$
 - Es existiert Objekt x , so dass in S $r_i(x)$ vor $w_j(x)$ kommt
 - Abkürzung: $rw_{ij}(x)$
- Schreib-Schreib-Abhängigkeit** von $T_i \rightarrow T_j$
 - Es existiert Objekt x , so dass in S $w_i(x)$ vor $w_j(x)$ kommt
 - Abkürzung: $ww_{ij}(x)$

20

3.2 Serialisierbarkeit von Transaktionen

Konfliktäquivalenz von Schedules

- Zwei Schedules S1 und S2 heißen konfliktäquivalent, wenn
 - S1 und S2 die gleichen Transaktions- und Aktionsmengen besitzen, d.h. wenn beide Schedules dieselben Operationen ausführen.
 - S1 und S2 die gleichen Abhängigkeitsmengen besitzen, d.h. wenn in der Abhängigkeitsmenge von S1 z.B. die Schreib-Lese-Abhängigkeit " $w_i(x)$ vor $r_j(x)$ " vorkommt, dann muss diese auch in der Abhängigkeitsmenge von S2 vorkommen.
- Zwei konflikt-äquivalente Schedules haben die gleiche Wirkung auf den Datenbank-Inhalt. (Gilt die Umkehrung?)

21

3.2 Serialisierbarkeit von Transaktionen

Konfliktäquivalenz von Schedules

- Beispiel: $S_1 = (r_1(x), r_1(y), r_2(x), w_2(x), w_1(x), w_1(y))$
 $S_2 = (r_2(x), r_1(x), r_1(y), w_2(x), w_1(x), w_1(y))$
 $S_3 = (r_1(x), r_1(y), r_2(x), w_1(x), w_2(x), w_1(y))$
 $S_4 = (r_2(x), r_1(y), r_1(x), w_2(x), w_1(y), w_1(x))$
 - Aktionsmengen von S_1 , S_2 und S_3 sind identisch
 - Abhängigkeitsmengen:
$$A_{S_1} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{2,1}(x)\}$$
$$A_{S_2} = \{rw_{2,1}(x), rw_{1,2}(x), ww_{2,1}(x)\}$$
$$A_{S_3} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{1,2}(x)\}$$
 - Schedule S_1 und S_2 sind konfliktäquivalent
 - Schedule S_1 und S_3 , bzw. S_2 und S_3 sind nicht konfliktäquivalent
 - Schedule S_4 ist kein Schedule derselben Transaktionen, da die Aktionen transaktionsintern vertauscht sind.

22

3.2 Serialisierbarkeit von Transaktionen

Serialisierungs-Graph

- Überprüfung, ob ein Schedule von $\{T_1, \dots, T_n\}$ serialisierbar ist (d.h. ob ein *konflikt-äquivalenter serieller Schedule existiert*)
- Die beteiligten Transaktionen $\{T_1, \dots, T_n\}$ sind die *Knoten* des Graphen
- Die *Kanten* beschreiben die Abhängigkeiten der Transaktionen: Eine Kante $T_i \rightarrow T_j$ wird eingetragen, falls im Schedule
 - $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr(x)$
 - $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw(x)$
 - $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww(x)$

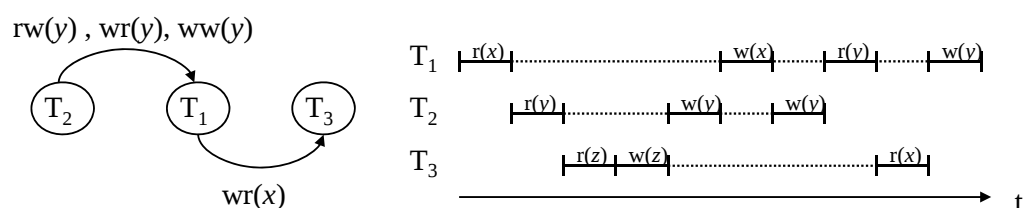
Die Kanten werden mit der Abhängigkeit beschriftet.

23

3.2 Serialisierbarkeit von Transaktionen

Serialisierungs-Graph

- Es gilt:
 - Ein Schedule ist serialisierbar, falls der Serialisierungs-Graph **zyklenfrei** ist
 - Einen zugehörigen konfliktäquivalenten seriellen Schedule erhält man durch topologisches Sortieren des Graphen
 - Es kann i.A. mehrere serielle Schedules geben.
 - Beispiel:
 $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$



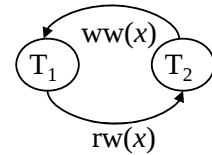
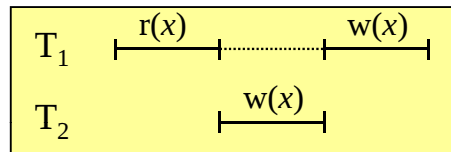
Serialisierungsreihenfolge: (T_2, T_1, T_3)

24

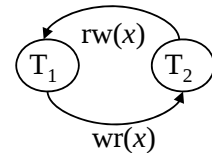
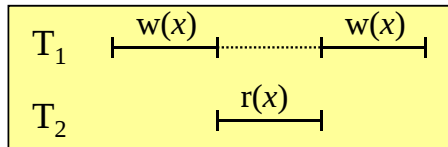
3.2 Serialisierbarkeit von Transaktionen

Beispiele für nicht-serialisierbare Schedules

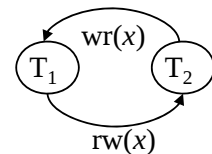
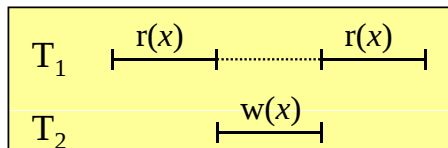
- Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



- Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$



- Non-repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$

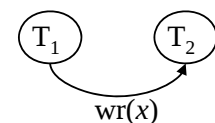
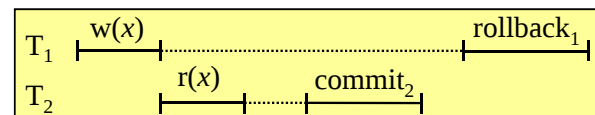


25

3.2 Serialisierbarkeit von Transaktionen

Rücksetzbare Schedules

- Bisher: Serialisierbarkeit
- Frage: was passiert, wenn eine Transaktion (z.B. auf eigenen Wunsch) zurückgesetzt wird?
- Beispiel:
 - T_1 schreibt Datensatz x
 - T_2 liest Datensatz x
 - T_2 führt COMMIT aus
 - Schedule ist serialisierbar, der Serialisierungs-Graph ist zyklensfrei



- ABER
 - T_1 wird zurückgesetzt (d.h. Datensatz x wird wieder auf den Ursprungswert zurückgesetzt)
 - T_2 müsste eigentlich auch zurückgesetzt werden, hat aber schon COMMIT ausgeführt

26

3.2 Serialisierbarkeit von Transaktionen

Rücksetzbare Schedules (cont.)

- Also: Serialisierbarkeit alleine reicht leider nicht aus, wenn TAs zurückgesetzt werden können
- Rücksetzbarer Schedule:
Eine Transaktion T_i darf erst dann ihr *COMMIT* durchführen, wenn alle Transaktionen T_j , von denen sie Daten gelesen hat, beendet sind.
- Andernfalls Problem: Falls ein T_j noch zurückgesetzt wird, müsste auch T_i zurückgesetzt werden, was nach *COMMIT* (T_i) nicht mehr möglich wäre

27

3.2 Serialisierbarkeit von Transaktionen

Rücksetzbare Schedules (cont.)

- Rücksetzbare Schedules können eine Lawine weiterer Rollbacks in Gang setzen:

Schritt	T_1	T_2	T_3	T_4	T_5
1.	$w_1(A)$	$r_2(A)$ $w_2(B)$	$r_3(B)$ $w_3(C)$	$r_4(C)$ $w_5(D)$	$r_5(D)$
2.					
3.					
4.					
5.					
6.					
7.					
8.	abort ₁				
9.					

- Schedule ohne kaskadierendes Rücksetzen:
Änderungen werden erst nach dem *COMMIT* für andere Transaktionen zum Lesen freigegeben

28

3.2 Serialisierbarkeit von Transaktionen

Überblick: Scheduleklassen

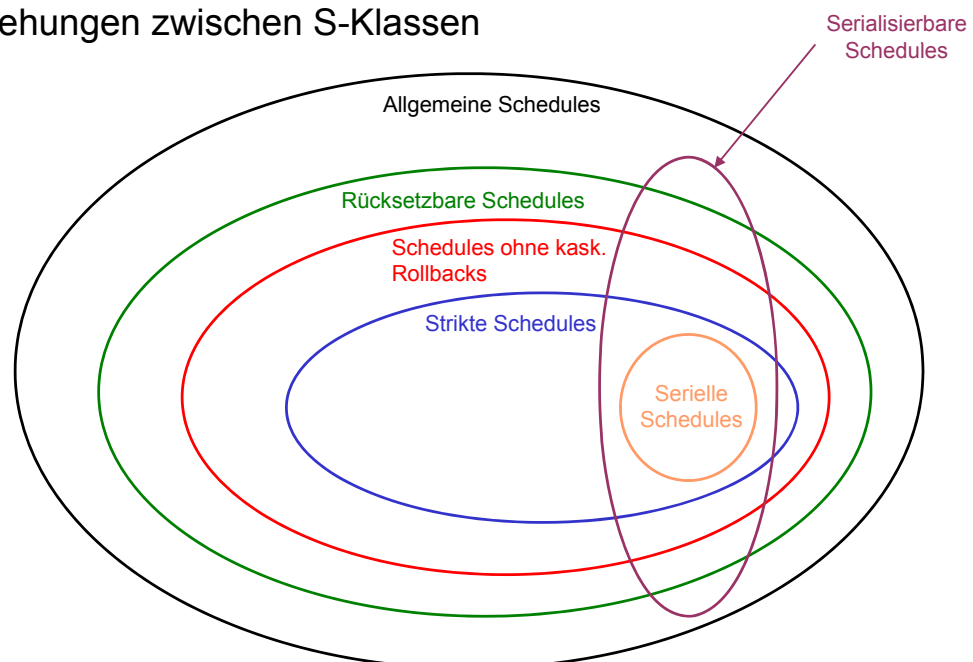
- Serieller S.
 - TAs in einzelnen Blöcken, phys. Einbenutzerbetrieb
- Serialisierbarer S.
 - Konfliktäquivalent zu einem seriellen S.
- Rücksetzbarer S.
 - TA darf erst committen, wenn alle TAs von denen sie Daten gelesen hat committed haben
- S. ohne kaskadierendes Rollback
 - Veränderte Daten einer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

29

3.2 Serialisierbarkeit von Transaktionen

Überblick: Scheduleklassen

- Beziehungen zwischen S-Klassen



30

3.2 Serialisierbarkeit von Transaktionen

Techniken zur Synchronisation

- Verwaltungsaufwand für Serialisierungsgraphen ist in der Praxis zu hoch.
Deshalb: Andere Verfahren, die die Serialisierbarkeit gewährleisten
- Pessimistische Ablaufsteuerung (*Locking*)
 - Konflikte werden vermieden, indem Transaktionen durch Sperren blockiert werden
 - Nachteil: ggf. lange Wartezeiten
 - Vorteil: I.d.R. nur wenig Rücksetzungen aufgrund von Synchronisationsproblemen nötig
 - Standardverfahren

31

3.2 Serialisierbarkeit von Transaktionen

Techniken zur Synchronisation (cont.)

- Optimistische Ablaufsteuerung
 - Transaktionen werden im Konfliktfall zurückgesetzt
 - Transaktionen arbeiten bis zum *COMMIT* ungehindert. Anschließend erfolgt Prüfung (z.B. anhand von Zeitstempeln), ob ein Konflikt aufgetreten ist
 - Nur geeignet, falls Konflikte zwischen Schreibern eher selten auftreten

32

3 Synchronisation

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

33

3.3 Sperrverfahren (Locking)

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

34

3.3 Sperrverfahren (Locking)

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.

35

3.3 Sperrverfahren (Locking)

Bemerkungen

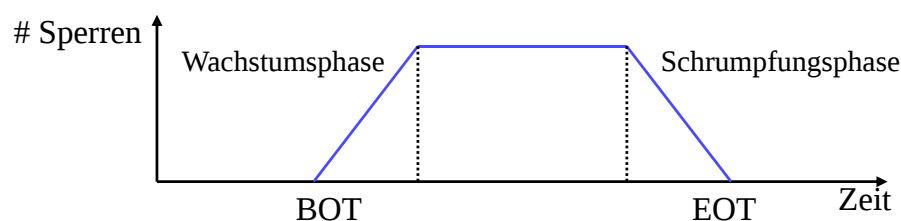
- Anfordern und Freigeben von Sperren sollte das DBMS implizit selbst vornehmen.
- Die Verwendung legaler Schedules garantiert noch nicht die Serialisierbarkeit.

36

3.3 Sperrverfahren (Locking)

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

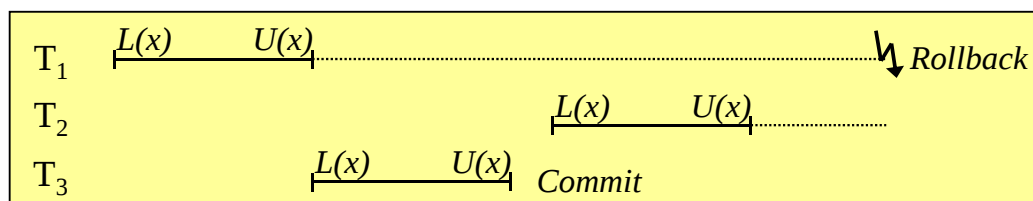


37

3.3 Sperrverfahren (Locking)

Zwei-Phasen-Sperrprotokoll (2PL)

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



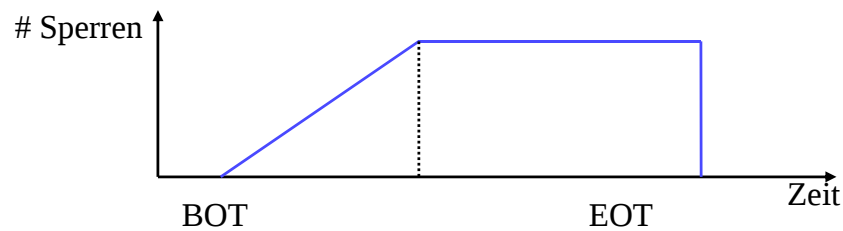
- Transaktion T_1 wird nach $U(x)$ zurückgesetzt
- T_2 hat "schmutzig" gelesen und muss zurückgesetzt werden
- Sogar T_3 muss zurückgesetzt werden → eklatanter Verstoß gegen die Dauerhaftigkeit (ACID) des COMMIT!

38

3.3 Sperrverfahren (Locking)

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



39

3.3 Sperrverfahren (Locking)

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

		bestehende Sperre	
		R	X
angeforderte Sperre	R	+	-
	X	-	-

40

3.3 Sperrverfahren (Locking)

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.

41

3.3 Sperrverfahren (Locking)

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:

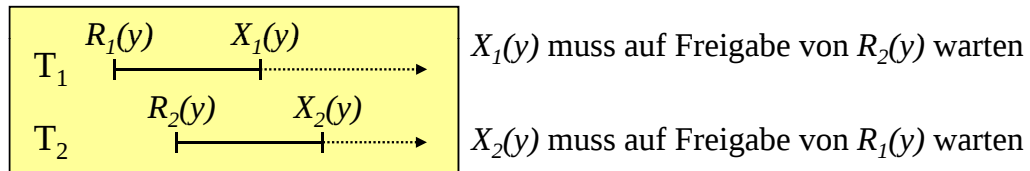
Grundsätzlich sind zwar auch Abhängigkeiten von T2 nach T1 denkbar (z.B. auf einem Objekt y), diese würden aber zu einer Verklemmung (Deadlock, gegenseitiges Warten) führen.

42

3.3 Sperrverfahren (Locking)

RUX-Sperrverfahren

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



- Lösung: Update-Sperren
 - U -Sperre 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)

43

3.3 Sperrverfahren (Locking)

RUX-Sperrverfahren

- Verträglichkeit der Sperrentypen

		bestehende Sperre		
angeforderte Sperre		R	U	X
	R	+	-	-
	U	+	-	-
	X	-	-	-

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

44

3.3 Sperrverfahren (Locking)

RAX-Sperrverfahren

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

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

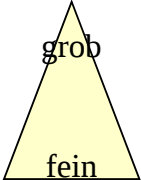
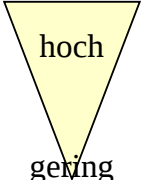
- Beim Konvertierungswunsch $A \rightarrow X$ Verhungern möglich
- Tradeoff zwischen höherer Parallelität und Verhungern

45

3.3 Sperrverfahren (Locking)

Hierarchische Sperrverfahren

- Sperrgranularität bestimmt Parallelität / Aufwand

Sperrobjekte	Granularität	Aufwand	Konfliktrate
DB			
DB-Segment			
Tabelle			
Tupel			

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

46

3.3 Sperrverfahren (Locking)

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 feingranularen Ebene (z.B. Tupel)
 - Zusätzlich Anwendung eines speziellen Protokolls (RIX) auf der grobgranularen Ebene (z.B. Relation)

47

3.3 Sperrverfahren (Locking)

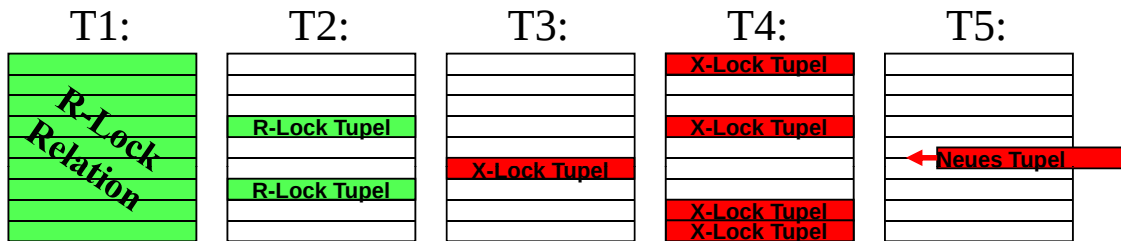
Hierarchische Sperrverfahren

- 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.

48

3.3 Sperrverfahren (Locking)

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

49

3.3 Sperrverfahren (Locking)

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:

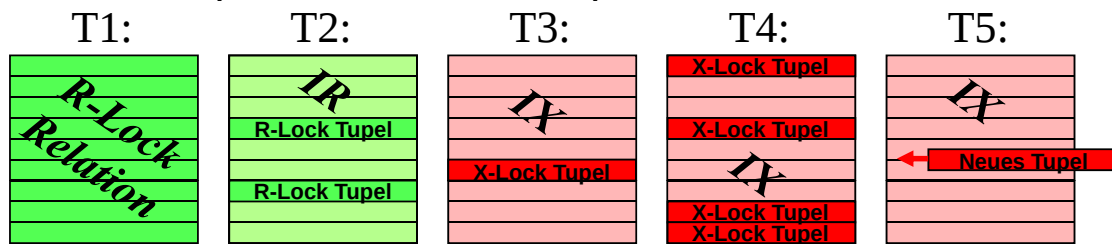
		bestehende Sperre				
		R	X	IR	IX	RIX
angeforderte Sperre	R	+	-	+	-	-
	X	-	-	-	-	-
	IR	+	-	+	+	+
	IX	-	-	+	+	-
	RIX	-	-	+	-	-

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

50

3.3 Sperrverfahren (Locking)

Hierarchische Sperrverfahren: Beispiel



		bestehende Sperre				
		R	X	IR	IX	RIX
angeforderte Sperre	R	+	-	+	-	-
	X	-	-	-	-	-
	IR	+	-	+	+	+
	IX	-	-	+	+	-
	RIX	-	-	+	-	-

51

3.3 Sperrverfahren (Locking)

Mehrversionen-Sperren: RAC

- Prinzip:
 - Änderungen erfolgen in lokalen Kopien im TA-Puffer
 - A-Sperren zur Änderung erforderlich
 - Bei COMMIT erfolgt Konvertierung von A→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}

52

3.3 Sperrverfahren (Locking)

Mehrversionen-Sperren: RAC

- Verträglichkeit der Sperrentypen

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

- 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)

53

3.3 Sperrverfahren (Locking)

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)

54

3.3 Sperrverfahren (Locking)

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

55

3.3 Sperrverfahren (Locking)

Konsistenzstufen nach J. Gray

- **Konsistenzstufe 0**
 - ohne Bedeutung, da Dirty Write und Lost Update möglich
- **Konsistenzstufe 1**
 - kein Dirty Write mehr, da Schreibsperren 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)

56

3.3 Sperrverfahren (Locking)

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

57

3.3 Sperrverfahren (Locking)

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)

58

3 Synchronisation

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

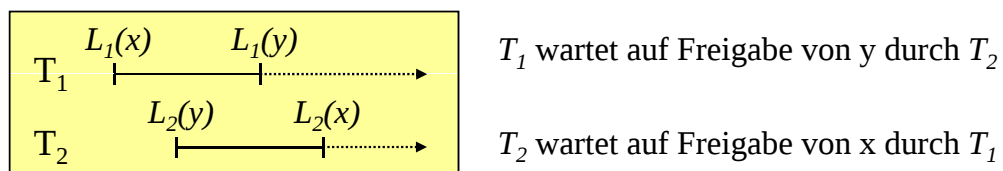
3.5 Synchronisation ohne Sperren

59

3.4 Behandlung von Verklemmungen

Verklemmung (Deadlock)

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



60

3.4 Behandlung von Verklemmungen

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

61

3.4 Behandlung von Verklemmungen

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

62

3.4 Behandlung von Verklemmungen

Erkennen und Auflösen von Deadlocks

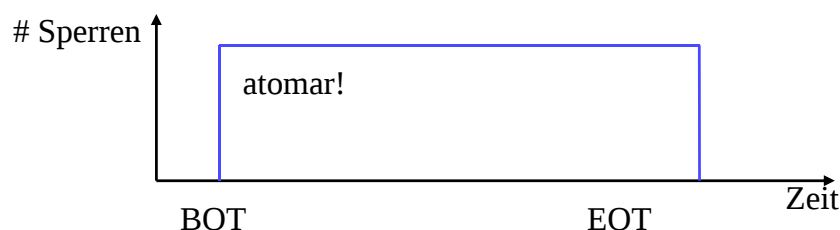
- Strategien zur Auflösung von Verklemmungen durch Rücksetzen beteiligter TAs:
 - **Minimierung des Rücksetzaufwands:** Wähle jüngste TA oder TA mit den wenigsten Sperren aus
 - **Maximierung der freigegebenen Ressourcen:** Wähle TA mit den meisten Sperren aus, um die Gefahr weiterer Verklemmungen zu 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

63

3.4 Behandlung von Verklemmungen

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

64

3.4 Behandlung von Verklemmungen

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

65

3.4 Behandlung von Verklemmungen

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.

66

3.4 Behandlung von Verklemmungen

Vermeidung von Deadlocks: Zeitstempel

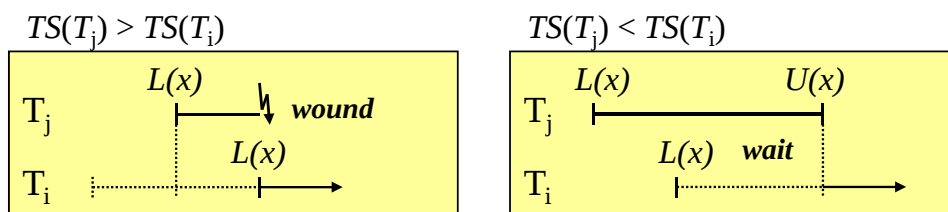
- Prinzip
 - Jeder Transaktion T_i wird zu Beginn ein Zeitstempel $TS(T_i)$ zugeordnet (*Time Stamp*):
 - Begin (BoT) einer TA
 - oder (besser) erste Datenbank-Operation einer TA
 - Objekte tragen nach wie vor Sperren
 - TAs warten nicht bedingungslos auf die Freigabe von Sperren
 - In Abhängigkeit von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

67

3.4 Behandlung von Verklemmungen

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

68

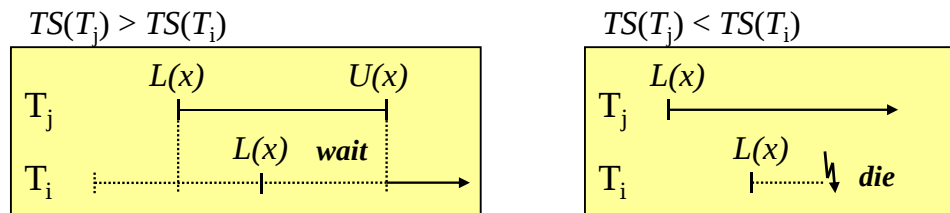
3.4 Behandlung von Verklemmungen

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

69

3.4 Behandlung von Verklemmungen

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

70

3.4 Behandlung von Verklemmungen

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

71

3 Synchronisation

Übersicht

3.1 Einleitung

3.2 Serialisierbarkeit von Transaktionen

3.3 Sperrverfahren (Locking)

3.4 Behandlung von Verklemmungen

3.5 Synchronisation ohne Sperren

72

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

73

3.5 Synchronisation ohne Sperren

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

74

3.5 Synchronisation ohne Sperren

- 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

75

3.5 Synchronisation ohne Sperren

- 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: $readTS(O) = \max(TS(T_i), readTS(O))$

76

3.5 Synchronisation ohne Sperren

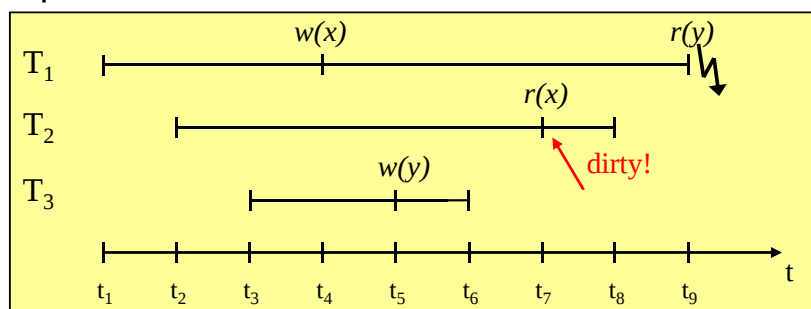
- Prüfungen beim **Schreibzugriff** von T_i auf ein Objekt O :

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

77

3.5 Synchronisation ohne Sperren

- 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: $writeTS(x) := t_1$
- t_5 : $write(y)$ in T_3 : Da $TS(T_3) > readTS(y)$ darf T_3 auf y schreiben, dann: $writeTS(y) := t_3$
- 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: $readTS(x) := t_2$
- 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.

78

3.5 Synchronisation ohne Sperren

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

79

3.5 Synchronisation ohne Sperren

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

80

3.5 Synchronisation ohne Sperren

Optimistische Synchronisation

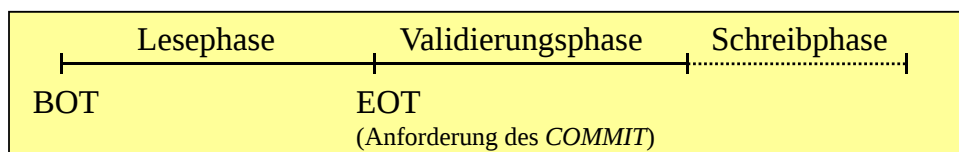
- Konzept
 - Keine Konfliktprävention
 - Konflikte werden erst bei *COMMIT* festgestellt
 - Im Konfliktfall werden Transaktionen zurückgesetzt
 - nahezu beliebige Parallelität, da TAs nicht blockiert werden

81

3.5 Synchronisation ohne Sperren

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.

82

3.5 Synchronisation ohne Sperren

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: $WS(T_i) \subseteq RS(T_j)$, 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

83

3.5 Synchronisation ohne Sperren

Optimistische Synchronisation (cont.)

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

84

3.5 Synchronisation ohne Sperren

BOCC

- Validierung von T_i
 - “Wurde eines der während der Lesephase 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 Lesephase von T_i validiert haben
- Algorithmus

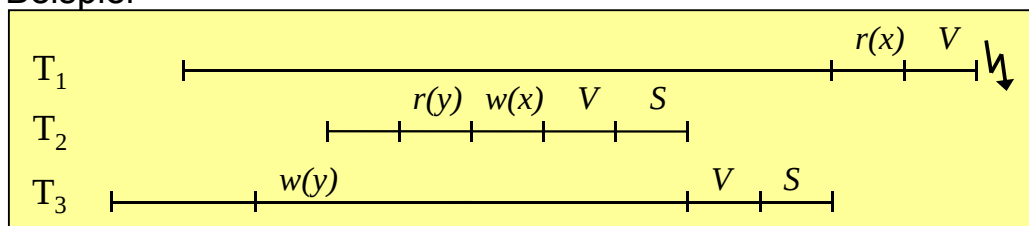
```
VALID := true;  
for (alle während Ausführung von  $T_i$  beendeten  $T_j$ ) do  
    if  $RS(T_i) \cap WS(T_j) \neq \emptyset$  then VALID := false;  
end;  
if VALID then Schreibphase( $T_i$ ); Commit ( $T_i$ );  
           else Rollback( $T_i$ ); // Nothing to do
```

85

3.5 Synchronisation ohne Sperren

BOCC (cont.)

- Beispiel



- Ablauf
 - T_2 wird erfolgr. validiert, da es noch keine validierten TAs gibt
 - T_3 wird erfolgreich validiert, 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.

86

3.5 Synchronisation ohne Sperren

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

87

3.5 Synchronisation ohne Sperren

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 Schreibphase( $T_i$ ) ; commit ( $T_i$ );
    else löse Konflikt auf;
```

88

3.5 Synchronisation ohne Sperren

FOCC (cont.)

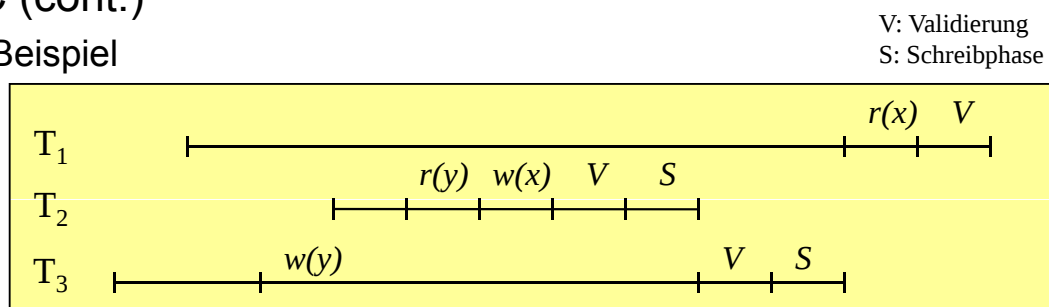
- Bewertung
 - Validierung muss nur von ändernden Transaktionen durchgeführt werden.
 - die überflüssigen Rücksetzungen von BOCC werden vermieden
 - überflüssige Rücksetzungen wegen der vorgegebenen Serialisierungs-Reihenfolge sind weiterhin möglich
 - mehr Freiheiten bei der **Konfliktauflösung**: beliebige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

89

3.5 Synchronisation ohne Sperren

FOCC (cont.)

- Beispiel



- Ablauf:
 - T₂ wird erfolgr. validiert, da x noch von keiner TA gelesen wurde
 - T₃ wird erfolgreich validiert, da y von keiner (noch) laufenden TA gelesen wurde
 - T₁ ist eine reine Lese-TA und muss nicht validiert werden
 - Hätte T₂ das Objekt y auch geändert, so wäre der Konflikt mit T₃ bei der Validierung von T₂ erkannt worden, und eine der beiden TAs hätte abgebrochen werden müssen

90

3.5 Synchronisation ohne Sperren

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

91

3.5 Synchronisation ohne Sperren

- **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, ...)
- **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

92