



Skript zur Vorlesung
Datenbanksysteme II
Sommersemester 2006

Kapitel 2: Synchronisation

Vorlesung: Christian Böhm

Übungen: Elke Achtert, Peter Kunath, Alexey Pryakhin

Skript © 2006 Christian Böhm

<http://www.dbs.informatik.uni-muenchen.de/Lehre/DBSII>



Inhalt

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Inhalt

Datenbanksysteme II
Kapitel 2: Synchronisation

3

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Synchronisation (Concurrency Control)

Datenbanksysteme II
Kapitel 2: Synchronisation

4

- **Serielle Ausführung** von Transaktionen ist unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist (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



Anomalien im Mehrbenutzerbetrieb

- 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



Lost Updates

- Änderungen einer Transaktion können durch Änderungen anderer Transaktionen 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“;

- Mgl. Ablauf:

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

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



Dirty Read / Dirty Write

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

T1	T2
<pre>UPDATE Passagiere SET Gepäck = Gepäck+3; ROLLBACK;</pre>	<pre>UPDATE Passagiere SET Gepäck = Gepäck+5; COMMIT;</pre>

- Durch den Abbruch von T1 werden die geänderten Werte ungültig. T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*)
- Verstoß gegen ACID: Dieser Ablauf verursacht einen inkonsistenten DB-Zustand (*Consistency*) bzw. T2 muss zurückgesetzt werden (*Durability*)



Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Bsp.:
 - 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
- Mgl. Ablauf:

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

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl T1 den DB-Zustand nicht geändert hat → Verstoß gegen *Isolation*



Phantomproblem

- Ausprägung des nicht-reproduzierbaren Lesens, bei der Aggregatfunktionen 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
- Mgl. 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



Inhalt

1. Anomalien im Mehrbenutzerbetrieb

2. Serialisierbarkeit von Transaktionen

3. Sperrverfahren (Locking)

4. Behandlung von Verklemmungen

5. Synchronisation ohne Sperren



Schedules (1)

- Die nebenläufige Bearbeitung von Transaktionen geschieht für den Benutzer transparent, d.h. als ob die Transaktionen (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden
- Ein **Schedule** für eine Menge $\{T_1, \dots, T_n\}$ von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der T_i s entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.



Schedules (2)

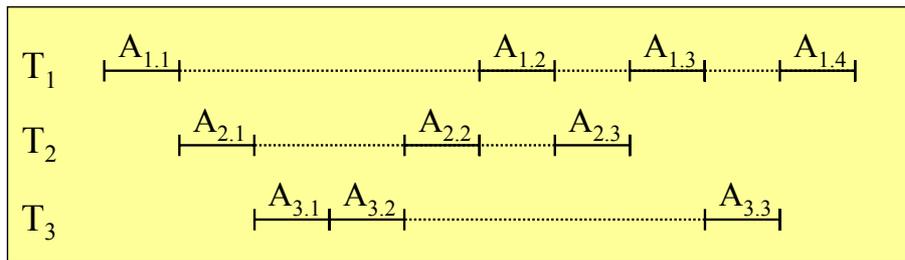
- Ein **serieller Schedule** ist ein Schedule S von $\{T_1, \dots, T_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden.
- Ein Schedule S von $\{T_1, \dots, T_n\}$ ist **serialisierbar**, wenn er dieselbe Wirkung hat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

Nur serialisierbare Schedules dürfen zugelassen werden!

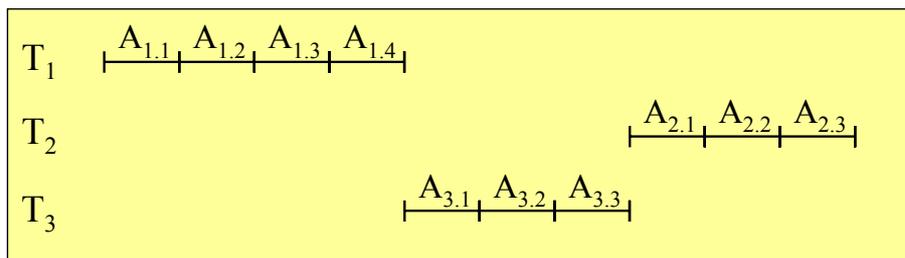


Beispiel serieller Schedule

- Beliebiger Schedule:



- Serieller Schedule:



Kriterium für Serialisierbarkeit (1)

Mit Hilfe von Serialisierungsgraphen kann man prüfen, ob ein Schedule $\{T_1, \dots, T_n\}$ serialisierbar ist:

- 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

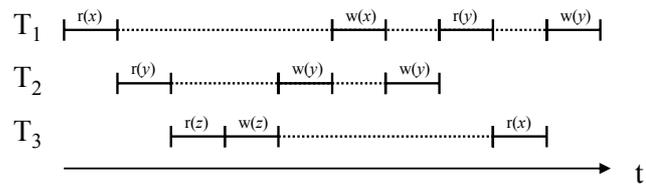
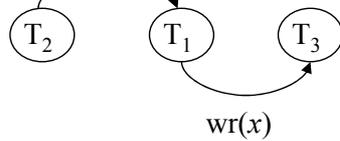
- $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr(x)$
- $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw(x)$
- $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww(x)$



Kriterium für Serialisierbarkeit (2)

- Ein Schedule ist serialisierbar, falls der Serialisierungsgraph **zyklenfrei** ist
- Einen zugehörigen seriellen Schedule erhält man durch topologisches Sortieren des Graphen
- Beispiel:
 $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$

$rw(y), wr(y), ww(y)$

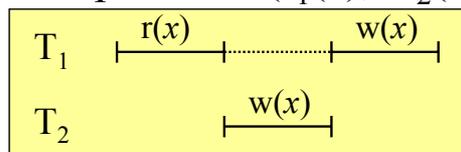


Serialisierungsreihenfolge: (T_2, T_1, T_3)

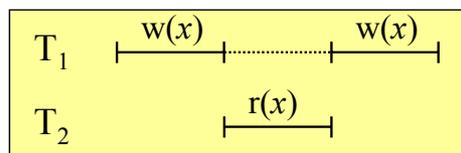


Beispiele nicht-serialisierbare Schedules

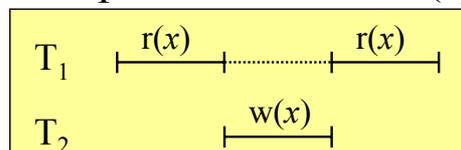
- Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



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



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





Konfliktäquivalenz von Schedules (1)

Zwei Schedules S_1 und S_2 heißen **konfliktäquivalent**, wenn

- i. S_1 und S_2 die gleichen Aktionsmengen besitzen, d.h. wenn beide Schedules dieselben Operationen ausführen.
- ii. S_1 und S_2 die gleichen Abhängigkeitsmengen besitzen, d.h. wenn in der Abhängigkeitsmenge von S_1 z.B. die Schreib-Lese-Abhängigkeit " $w_i(x)$ vor $r_j(x)$ " vorkommt, dann muss diese auch in der Abhängigkeitsmenge von S_2 vorkommen.



Konfliktäquivalenz von Schedules (2)

Beispiel:

$$S_1 = (r_1(x), r_1(y), r_2(x), w_2(x), w_1(x), w_1(y))$$

$$S_2 = (r_2(x), r_1(y), r_1(x), w_2(x), w_1(y), w_1(x))$$

$$S_3 = (r_1(x), r_1(y), r_2(x), w_1(x), w_2(x), w_1(y))$$

- Aktionsmengen von S_1 , S_2 und S_3 sind identisch
- Abhängigkeitsmengen:

$$A_{S_1} = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_2(x), w_1(x))\}$$

$$A_{S_2} = \{(r_2(x), w_1(x)), (r_1(x), w_2(x)), (w_2(x), w_1(x))\}$$

$$A_{S_3} = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x), w_2(x))\}$$

- Schedule S_1 und S_2 sind konfliktäquivalent
Schedule S_1 und S_3 , bzw. S_2 und S_3 sind nicht konfliktäquivalent



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
- **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



Inhalt

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Grundbegriffe (1)

- **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



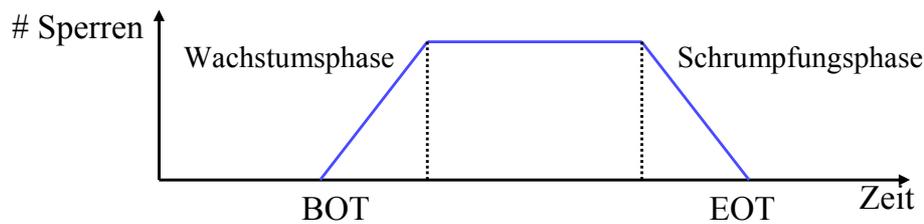
Grundbegriffe (2)

- **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

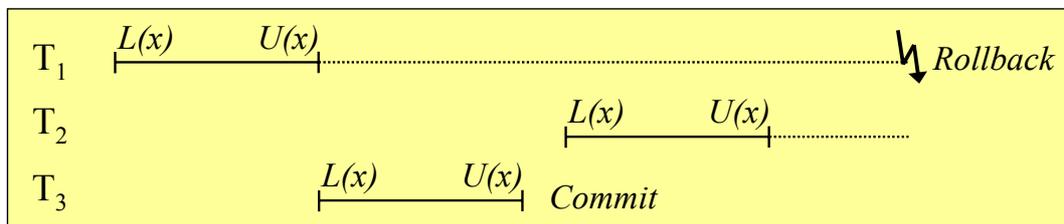


- Serialisierbarkeit ist gewährleistet, da Serialisierungsgraphen keine Zyklen enthalten können.



Striktes Zwei-Phasen-Sperrprotokoll (1)

- Problem des einfachen 2PL: Gefahr des kaskadierenden Rücksetzens im Fehlerfall
- Beispiel:

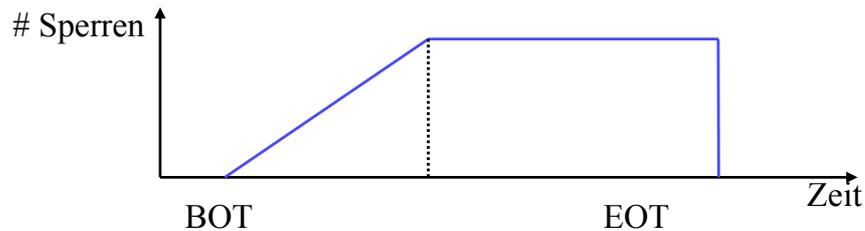


- Transaktion T_1 wird nach $U(x)$ zurückgesetzt
- Transaktion T_2 hat “schmutzig” gelesen und muss auch zurückgesetzt werden
- Sogar die abgeschlossene Transaktion T_3 muss zurückgesetzt werden → eklatanter Verstoß gegen die Dauerhaftigkeit (ACID) des *COMMIT*!



Striktes Zwei-Phasen-Sperrprotokoll (2)

- 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

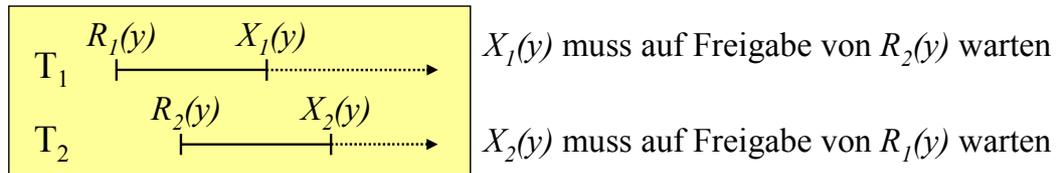
- Bisher: kein paralleles Lesen oder Schreiben möglich
- Jetzt: Parallelität unter Lesern erlaubt
- 2 Arten von Sperren
 - Lesesperren oder **R-Sperren** (*read locks*)
 - Schreibsperren oder **X-Sperren** (*exclusive locks*)
- Verträglichkeit der Sperrentypen

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



RUX-Sperrverfahren (1)

- 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 (2)

- Verträglichkeit der Sperrentypen

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

- Kein Verhungern möglich, da spätere Leser keinen Vorrang haben
- Keine Konversionsverklebung auf demselben Objekt möglich
- Verklebungen 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

Sperrobjekte	Granularität	Aufwand	Konfliktrate
DB	grob	gering	hoch
DB-Segment	fein	hoch	gering
Tabelle			
Tupel			

- Tradeoff
 - Geringe Konfliktrate ermöglicht hohen Parallelitätsgrad
 - Feine Granularität verursacht hohen Verwaltungsaufwand
- Lösung: variable Granularität durch hierarchische Sperren
- Kommerzielle DBS unterstützen zumeist 2-stufige Objekthierarchie, z.B. Segment-Seite oder Tabelle-Tupel

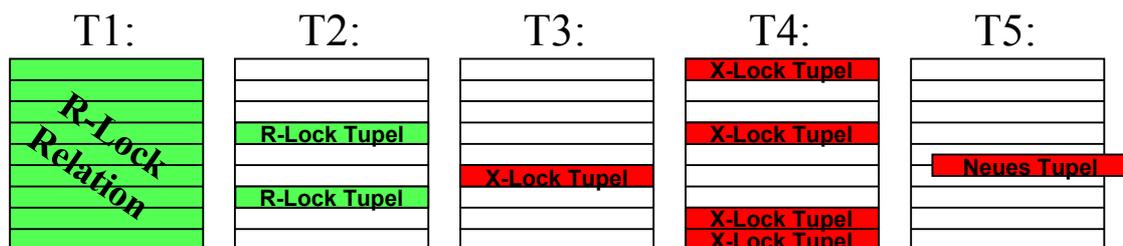


Hierarchische Sperrverfahren

- Vorgehensweise bei hierarchischen Sperrverfahren:
 - Anwendung eines beliebigen Sperrprotokolls (z.B. RX) auf der fein-granularen Ebene (z.B. Tupel)
 - Anwendung eines speziellen Protokolls (RIX) auf der grob-granularen Ebene (z.B. Relation)
- 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 Sperren: 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



Hierarchisches Konzept: Intentionssperren

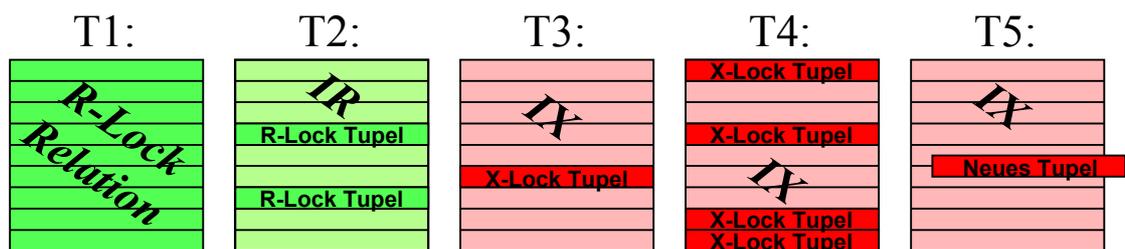
- *IR-Sperre (intention read)*: auf feinerer Granularitätsstufe existiert (mindestens) eine *R-Sperre*
- *IX-Sperre (intention exclusive)*: auf feinerer Stufe *X-Lock*
- *RIX-Sperre (R-Sperre + IX-Sperre)*: volle Lesesperre und feinere Schreibsperre (sonst zu große Behinderung)
- **Verträglichkeit der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



Beispiel



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



Mehrversionen-Sperren: RAC (1)

- 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 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 (2)

- 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>	+	-	-

- 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 (Abhängigkeitsbeziehungen 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
- Verschiedene Konzepte für Konsistenzstufen
 - Definition über Sperrentypen (“Konsistenzstufen” nach Jim Gray):
 - Definition über Anomalien (“Isolation Levels” in SQL92)



Konsistenzstufen nach J. Gray (1)

- 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

- **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



Konsistenzstufen nach J. Gray (2)

- **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)
- **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)



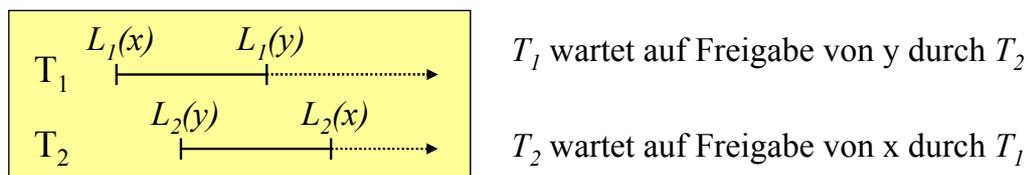
Inhalt

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Verklemmung (Deadlock)

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



- 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
 - TAs sperren Objekte in beliebiger Reihenfolge
 - TAs warten auf Sperrenfreigabe durch andere TAs, ohne selbst Sperren freizugeben



Erkennen und Auflösen von Deadlocks (1)

- **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



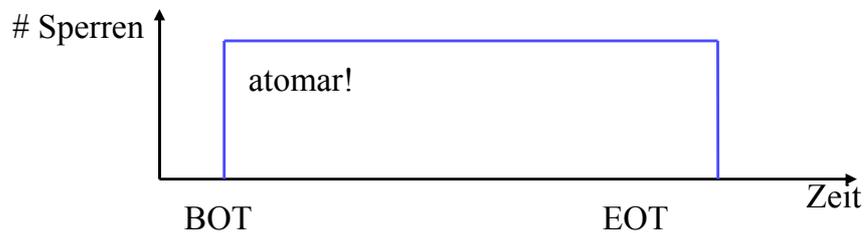
Erkennen und Auflösen von Deadlocks (2)

- 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 durch Preclaiming (1)

- **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 durch Preclaiming (2)

- **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 durch Zeitstempel (1)

• Konzept

- Jeder Transaktion T_i wird zu Beginn ein Zeitstempel $TS(T_i)$ zugeordnet (*Time Stamp*)
- 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* und *wait-die*

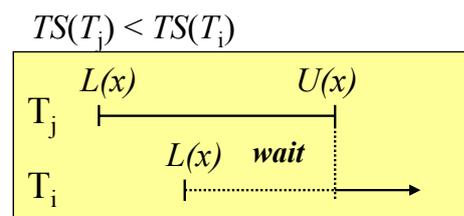
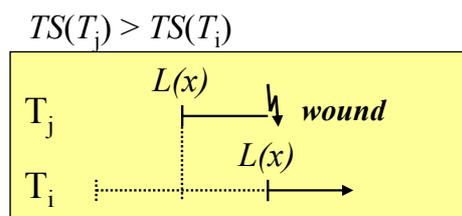


Vermeidung von Deadlocks durch Zeitstempel (2)

• **wound-wait:**

T_i fordere 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

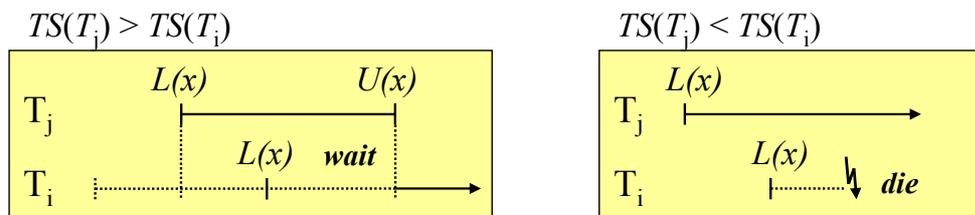


Vermeidung von Deadlocks durch Zeitstempel (3)

- **wait-die:**

T_i fordere 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

- **Verhalten:**

- es treten keine Deadlocks mehr auf
- Gelegentlich werden TAs zurückgesetzt, ohne dass tatsächlich ein Deadlock aufgetreten wäre



Inhalt

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Synchronisation ohne Sperren

- **Synchronisation mit Sperren**
 - *pessimistische* Annahme: Konflikte sind möglich und 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 (1)

- 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 (2)

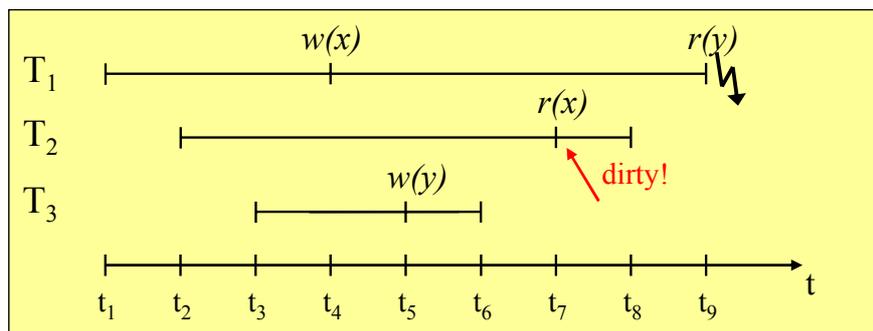
- Prüfungen beim **Schreibzugriff** von T_i auf ein Objekt O :

- Falls $TS(T_i) < readTS(O)$:
 T_i ist älter als TA, die O gerade 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: $writeTS(O) = TS(T_i)$



Zeitstempel statt Sperren (3)

- 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



Zeitstempel statt Sperren (4)

- **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 (5)

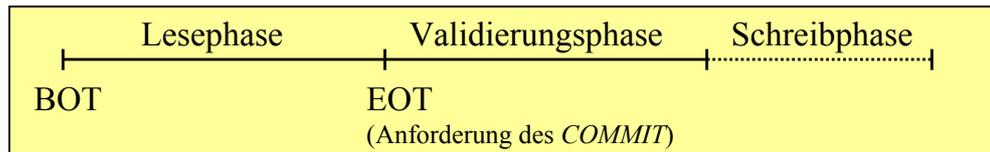
- **Nachteil für lange TAs**
 - Rücksetzgefahr steigt mit Dauer der TA
 - Verhungern von TAs durch wiederholtes Zurücksetzen wird nicht verhindert
- **Bewertung**
 - Verwaltung der Objektmarken ist sehr aufwändig und nicht feingranularer als auf Seitenebene praktikabel
 - Zeitstempel müssen für jedes Objekt verwaltet werden, während Sperren nur bei Zugriff auf Objekte angelegt werden



Optimistische Synchronisation (1)

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

- **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



Optimistische Synchronisation (2)

- **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)
 - 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
- **Bemerkungen**
 - Serialisierungsreihenfolge ist durch Validierungsreihenfolge gegeben
 - Validierung und Schreiben muss ununterbrechbar durchgeführt werden



BOCC (1)

- **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 Schreipphase( $T_i$ ); Commit ( $T_i$ );
else Rollback( $T_i$ ); // Nothing to do

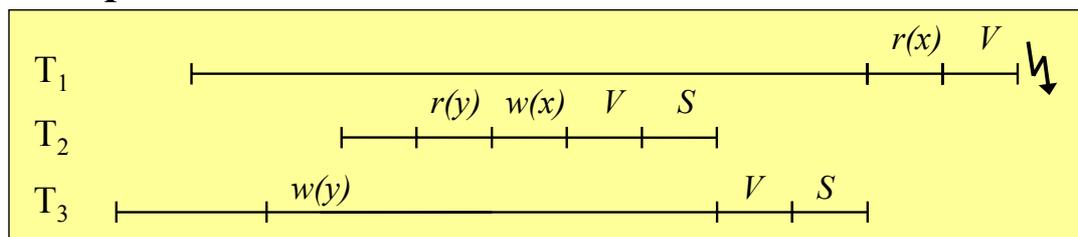
```



BOCC (2)

• Beispiel

V: Validierung
S: Schreibphase



• Ablauf

- T_2 wird erfolgreich 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.



BOCC (3)

- **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 (1)

- **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 (2)

- **Bewertung**

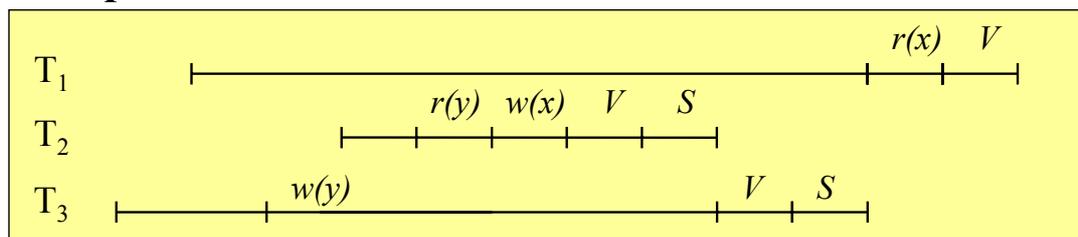
- Validierung muss nur von ändernden Transaktionen durchgeführt werden.
- die überflüssigen Rücksetzungen von BOCC werden vermieden
- überflüssige Rücksetzungen wegen der vorgegebenen Serialisierungs-Reihenfolge sind weiterhin möglich
- mehr Freiheiten bei der **Konfliktauflösung**: beliebige TA kann 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.



FOCC (3)

- **Beispiel**

V: Validierung
S: Schreibphase



- **Ablauf:**

- T₂ wird erfolgreich 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



Abschließende Bemerkungen

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