Gleiche Wirkung auf den DB-Zustand

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

  Konflikt-Äquivalenz

- Idee: Wenn in S1 eine Transaktion T1 z.B. einen Wert liest, den T2 geschrieben hat, dann muss das auch in S2 so sein.
- Wir sprechen hier von einer Schreib-Lese-Abhängigkeit (bzw. Konflikt) zwischen T2 und T1 (in Schedule S1)

3 Arten von Abhängigkeiten

Sei *S* ein Schedule.

Wir sprechen von einer…

1. Schreib-Lese-Abhängigkeit von $T_i \rightarrow T_j$ wenn ein Obj. x existiert, so dass in S $w_i(x)$ vor $r_j(x)$ kommt:
   $wr_{i,j}(x)$
2. Lese-Schreib-Abhängigkeit von $T_i \rightarrow T_j$ wenn ein Obj. x existiert, so dass in S $r_i(x)$ vor $w_j(x)$ kommt:
   $rw_{i,j}(x)$
3. Schreib-Schreib-Abhängigkeit von $T_i \rightarrow T_j$ wenn ein Obj. x existiert, so dass in S $w_i(x)$ vor $w_j(x)$ kommt:
   $ww_{i,j}(x)$

Konfliktäquivalenz von Schedules (1)

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

i. S_1 und S_2 die gleichen Transaktions- und 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.

Zwei konflikt-äquivalente Schedules haben die gleiche Wirkung auf den Datenbank-Inhalt. (Gilt die Umkehrung?)

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:

  $A_{S1} = \{(r_1(x),w_2(x)),\ (r_2(x),w_1(x)),(w_2(x),w_1(x))\}$

  $A_{S2} = \{(r_2(x),w_1(x)),\ (r_1(x),w_2(x)),(w_2(x),w_1(x))\}$

  $A_{S3} = \{(r_1(x),w_2(x)),\ (r_2(x),w_1(x)),(w_1(x),w_2(x))\}$

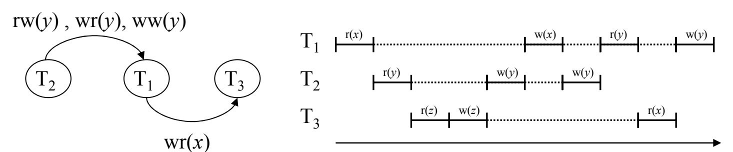
→ Schedule S_1 und S_2 sind konfliktäquivalent
Schedule S_1 und S_3, bzw. S_2 und S_3 sind nicht konfliktäquivalent

Serialisierungs-Graph (1)

- Mit Hilfe eines **Serialisierungs-Graphen** (auch: **Abhängigkeits-Graphen**) kann man prüfen, ob ein Schedule von $\{T_1, \ldots, T_n\}$ serialisierbar ist (d.h. ob ein konflikt-äquivalenter serieller Schedule existiert)
- Die beteiligten Transaktionen $\{T_1, \ldots, T_n\}$ 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.

Serialisierungs-Graph (2)

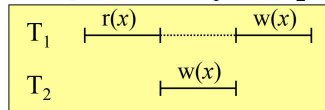
- Ein Schedule ist serialisierbar, falls der Serialisierungs-graph **zyklenfrei** ist
- Einen zugehörigen konfliktäquivalenten seriellen Schedule erhält man durch topologisches Sortieren des Graphen
- Es kann i.a. mehrere serielle Schdules geben.
- Beispiel:
  $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y),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

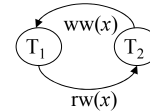


Serialisierungsreihenfolge: (T_2, T_1, T_3)

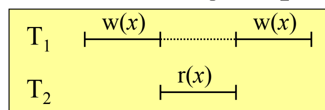
Beispiele nicht-serialisierbare Schedules

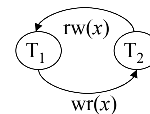
- Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



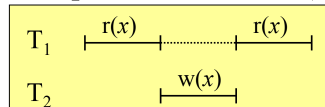


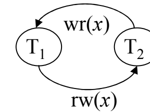
- Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$





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





Rücksetzbare Schedules (1)

- Zusätzlich zur Serialisierbarkeit besteht die Anforderung, dass Transaktionen jederzeit (z.B. auf eigenen Wunsch) rücksetzbar sein müssen
- **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

Rücksetzbare Schedules (2)

- Auch rücksetzbare Schedules können eine Lawine weiterer Rollbacks in Gang setzen:

Schritt	T_1	T_2	T_3	T_4	T_5
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_5(D)$	
8.					$r_5(D)$
9.	$abort_1$				

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

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)

Datenbanksysteme II
Kapitel 2: Synchronisation

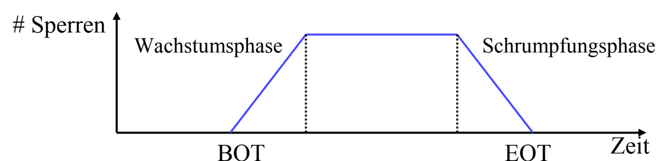
- **Sperre (Lock)**
  - Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
  - Anforderung einer Sperre durch *LOCK*, Freigabe durch *UNLOCK*
  - *LOCK* / *UNLOCK* erfolgt atomar
  - Sperrgranularität: Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
  - Sperrenverwalter führt 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)

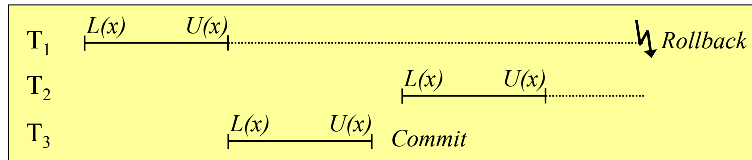
- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- **Merkmal**: keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
  - *Wachstumsphase*: Anforderungen der Sperren
  - *Schrumpfungsphase*: Freigabe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



- Serialisierbarkeit ist gewährleistet, da Serialisierungs-graphen keine Zyklen enthalten können.

Striktes Zwei-Phasen-Sperrprotokoll (1)

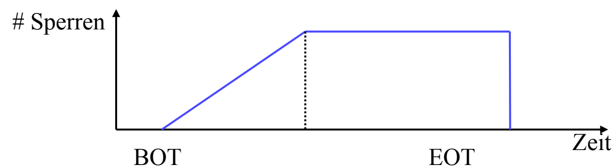
- Problem des einfachen 2PL: Gefahr des kaskadierenden Rücksetzens im Fehlerfall (bzw. sogar Nicht-Rücksetzbar)
- Beispiel:



  – 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

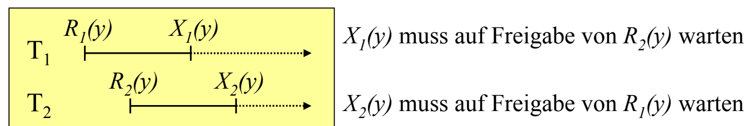
		bestehende Sperre	
		R	***X***
angeforderte Sperre	***R***	+	-
	X	-	-

Serialisierungsreihenfolge bei RX

- 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.
- Beispiel: Wenn T1 ein Objekt *x* schreibt und danach T2 *x* lesen möchte, so muss T2 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.
- 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.
- Durch striktes 2PL nur kaskadenfreie rücksetzb. Schedules

RUX-Sperrverfahren (1)

- 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)

RUX-Sperrverfahren (2)

- Verträglichkeit der Sperrentypen

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

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

RAX-Sperrverfahren

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

angeforderte Sperre \ *bestehende Sperre*	***R***	***A***	***X***
R	+	+	-
A	+	-	-
X	-	-	-

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

Hierarchische Sperrverfahren

- Sperrgranularität bestimmt Parallelität / Aufwand

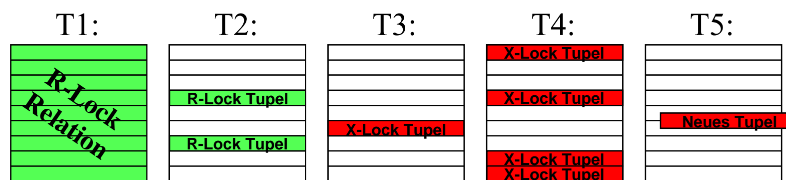
Sperrobjekte	**Granularität**	**Aufwand**	**Konfliktrate**
DB DB-Segment Tabelle Tupel	grob … fein	gering … hoch	hoch … gering

- 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)
  - Zusätzlich 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

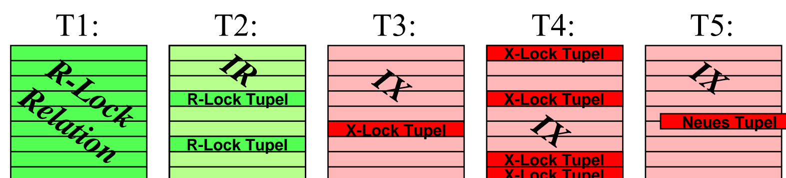
Hierarchisches Konzept: Intentionssperren

- *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:

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

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

Beispiel



angeforderte Sperre \ *bestehende Sperre*	**R**	**X**	**IR**	**IX**	**RIX**
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 $A \rightarrow C$
- *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)

Datenbanksysteme II
Kapitel 2: Synchronisation

- Verträglichkeit der Sperrentypen

angeforderte Sperre \ *bestehende Sperre*	***R***	***A***	***C***
R	+	+	+
A	+	-	-
C	+	-	-

- 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 (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)

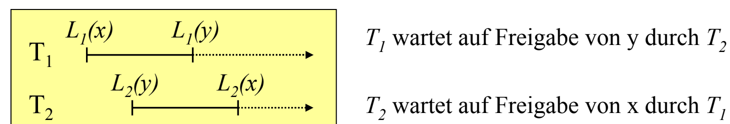
4. Behandlung von Verklemmungen

5. Synchronisation ohne Sperren

Datenbanksysteme II
Kapitel 2: Synchronisation

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)

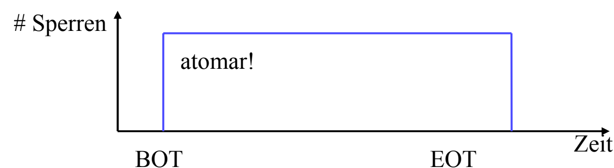
- **Time-Out Strategie**
  - Falls eine TA innerhalb einer Zeiteinheit *t* keinen Fortschritt macht, wird sie als verklemmt betrachtet und zurückgesetzt
  - *t* zu klein: TAs werden u. U. beim Warten auf Ressourcen abgebrochen
  - *t* zu groß: Verklemmungszustände werden zu lange geduldet
- **Wartegraphen**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

Deadlock-Vermeidung durch Ordnung

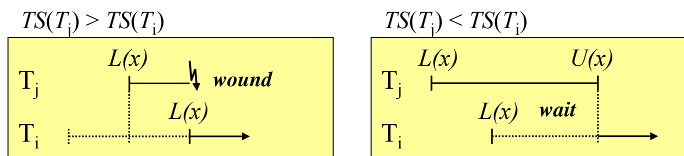
- Auf den Datenbank-Objekten wird eine totale Ordnung definiert, z.B. Relation R1 < R2 < R3 < …
- Annahme: Es gibt nur eine Sperren-Art (X).
- Es wird festgelegt, dass Sperren nur in aufsteigender Reihenfolge (bezüglich dieser Ordnung) vergeben werden (ggf. werden nicht benötigte Objekte mit gesperrt).
- Damit ist gegenseitiges Warten nicht mehr möglich.
- Aber das Szenario ist ähnlich restriktiv wie Preclaiming.
- Für Spezial-Anwendungen ist die Definition einer Ordnung auf den DB-Objekten durchaus denkbar.

Vermeidung von Deadlocks durch Zeitstempel (1)

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

Vermeidung von Deadlocks durch Zeitstempel (2)

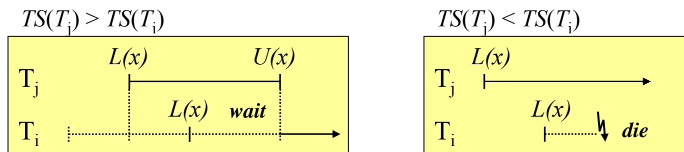
- **wound-wait**:
  T_i fordert Sperre $L(x)$ an.
  - Jüngere TA T_j , d.h. $TS(T_j) > TS(T_i)$, hält bereits Sperre auf x:
    => T_i läuft weiter, jüngere TA T_j wird zurückgesetzt (***wound***)
  - Ältere TA T_j, d.h. $TS(T_j) < TS(T_i)$, hält bereits Sperre auf x:
    => T_i wartet auf Freigabe der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



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

Vermeidung von Deadlocks durch Zeitstempel (3)

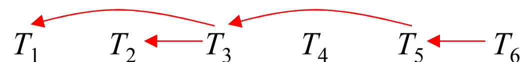
- **wait-die**:
  T_i fordert Sperre $L(x)$ an.
  - Jüngere TA T_j , d.h. $TS(T_j) > TS(T_i)$, hält bereits Sperre auf x:
    => T_i wartet auf Freigabe der Sperre durch jüngere TA T_j (***wait***)
  - Ältere TA T_j, d.h. $TS(T_j) < TS(T_i)$, hält bereits Sperre auf x:
    => T_i wird zurückgesetzt (***die***),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



→ ältere TAs müssen zunehmend mehr warten

Deadlock-Freiheit bei Wound-Wait

- Die Zeitstempel („Alter der Transaktion“) definieren eine strikte, totale Ordnung auf den Transaktionen:
  - $TS(T_1) < TS(T_2) < TS(T_3) < \ldots < TS(T_n)$
- Bei der Wound-Wait-Strategie warten jüngere auf ältere Transaktionen, aber nie umgekehrt:
  - T_i wartet auf T_j => $TS(T_j) < TS(T_i)$
- 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

Serialisierbarkeit bei Wound-Wait

- 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-Sprerrprotokoll
- Der „äquivalente serielle Schedule“ ist aber nicht beliebig: Nur der nach aufsteigenden Zeitstempeln geordnete serielle Schedule wird überprüft
- Serialisierungsreihenfolge definiert durch BOT
- Rücksetzungen wesentlich häufiger als nötig

- WAIT-DIE: Analog
  - Pfeile im Wartegraphen nie von rechts nach links
  - Äquivalenter serieller Schedule gleich