3.2 Serialisierbarkeit von Transaktionen

Techniken zur Synchronisation

- Verwaltungsaufwand für Serialisierungsgraphen ist in der Praxis zu hoch. Deshalb: Andere Verfahren, die Serialisierbarkeit gewährleisten
- Pessimistische Ablaufsteuerung (Standardverfahren: Locking)
  - Konflikte werden vermieden, indem Transaktionen (typischerweise durch Sperren) blockiert werden
  - Nachteil: ggf. lange Wartezeiten
  - Vorteil: I.d.R. nur wenig Rücksetzungen aufgrund von Synchronisationsproblemen nötig
- Optimistische Ablaufsteuerung
  - Transaktionen werden im Konfliktfall zurückgesetzt
  - Transaktionen arbeiten bis zum COMMIT ungehindert. Anschließend erfolgt Prüfung (z.B. anhand von Zeitstempeln), ob Konflikt aufgetreten ist
  - Nur geeignet, falls Konflikte zwischen Schreibern eher selten auftreten

Techniken zur Synchronisation

- Im folgenden:
  - zunächst Sperrverfahren (***DAS*** Standardverfahren zur Synchronisierung)

    3.3 -> Grundlagen, verschiedene Varianten, etc.

    3.4 -> ein wesentliches Problem von Sperrverfahren: Deadlocks

  - Dann weitere pessimistische und optimistische Verfahren, die ohne Sperren auskommen

3. Synchronisation

3.3 Sperrverfahren (Locking)

Sperre (Lock)

- Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
- Anforderung einer Sperre durch *LOCK*, z.B. *L*(*x*) für *LOCK* auf Objekt *x*
- Freigabe durch *UNLOCK*, z.B. *U*(*x*) für *UNLOCK* von Objekt *x*
- *LOCK* / *UNLOCK* erfolgt atomar (also nicht unterbrechbar!)
- Sperrgranularität (Objekte, auf denen Sperren gesetzt werden):
  Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
- Sperrenverwalter führt Tabelle für aktuell gewährte Sperren

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.

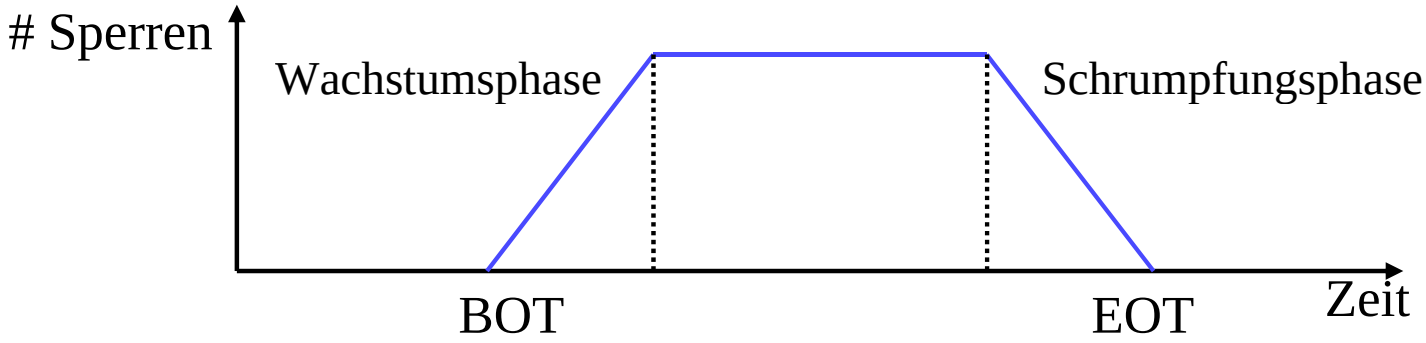
Bemerkungen

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

3.3 Sperrverfahren (Locking)

Zwei-Phasen-Sperrprotokoll (2PL)

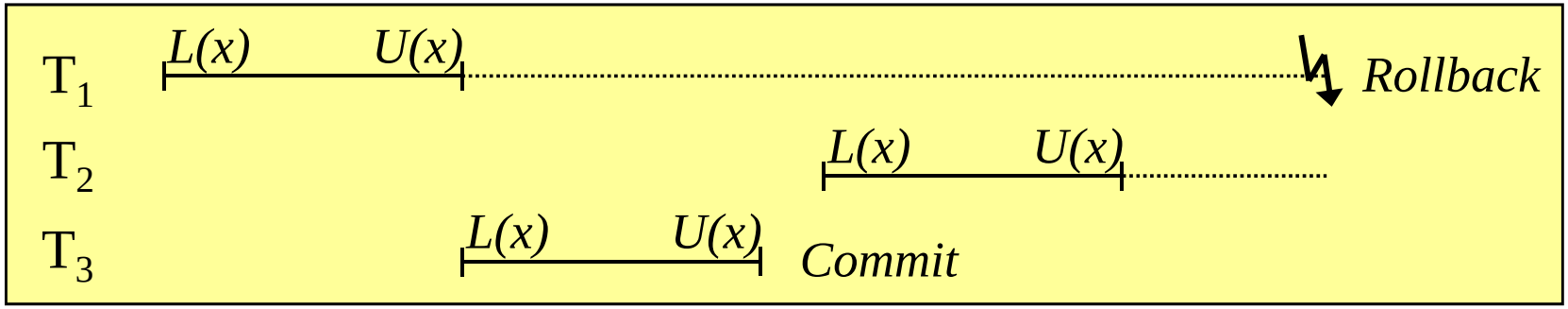
- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- Merkmal: keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
  - Wachstumsphase: Anforderungen der Sperren
  - Schrumpfungsphase: Freigabe der Sperren



3.3 Sperrverfahren (Locking)

Zwei-Phasen-Sperrprotokoll (2PL)

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

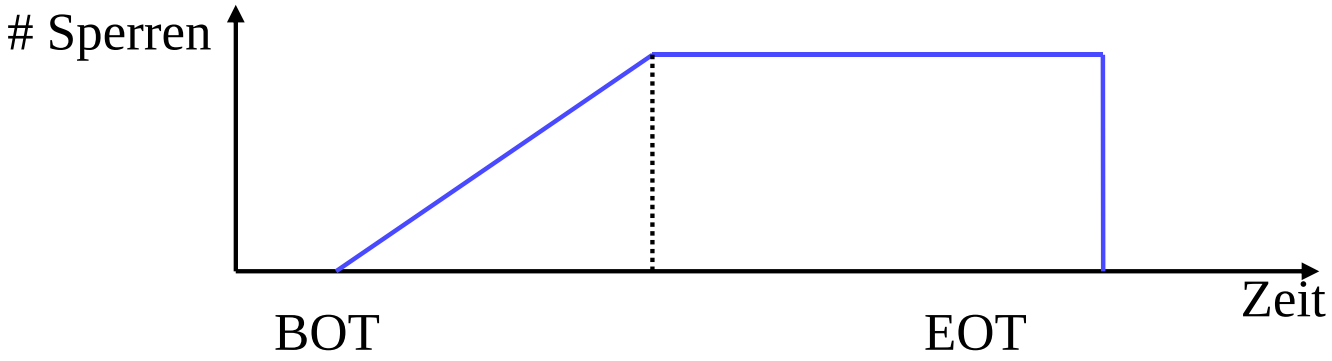


- Transaktion T_1 wird nach *U*(*x*) zurückgesetzt
- T_2 hat "schmutzig" gelesen und muss zurückgesetzt werden
- Sogar T_3 muss zurückgesetzt werden

  → Verstoß gegen die Dauerhaftigkeit (ACID) des *COMMIT*!

Striktes Zwei-Phasen-Sperrprotokoll

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



3.3 Sperrverfahren (Locking)

Erhöhung des Parallelisierungsgrads

- Bisher: Objekt ist entweder gesperrt oder zur Bearbeitung frei
  => kein paralleles Lesen oder Schreiben möglich
- ABER: Parallelität unter Lesern könnte man eigentlich erlauben
- Dazu 2 Arten von Sperren
  - Lesesperren oder R-Sperren (read locks)
  - Schreibsperren oder X-Sperren (exclusive locks)

3.3 Sperrverfahren (Locking)

RX-Sperrverfahren

- R- und X-Sperren
- Parallelität unter Lesern erlaubt
- Verträglichkeit der Sperrentypen (siehe Tabelle rechts)

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

Serialisierungsreihenfolge bei RX

- RX-Sperrverfahren meist in Verbindung mit striktem 2PL um nur kaskadenfreie rücksetzbare Schedules zu erhalten
- Zur Erinnerung: 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.

3.3 Sperrverfahren (Locking)

- Beispiel (Serialisierungsreihenfolge bei RX):
  - Situation:
    - T_1 schreibt ein Objekt *x*
    - Danach möchte T_2 Objekt *x* lesen
  - Folge:
    - T_2 muss auf das *COMMIT* von T_1 warten, d.h. der serielle Schedule enthält T_1 vor T_2.
    - Da T_2 wartet, kommen auch alle weiteren Operationen erst nach dem *COMMIT* von T_1.
  - Achtung:

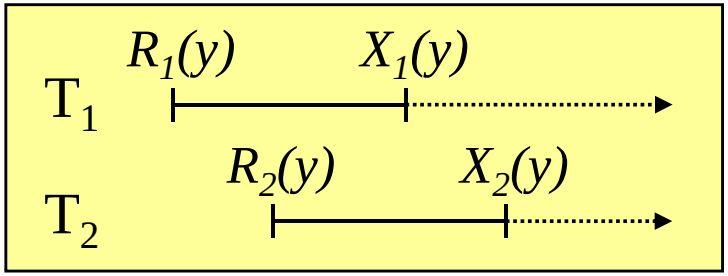
    Grundsätzlich sind zwar auch Abhängigkeiten von T_2 nach T_1 denkbar (z.B. auf einem Objekt *y*), diese würden aber zu einer ***Verklemmung*** (***Deadlock***, gegenseitiges Warten) führen.

Deadlocks (die Erste …)

- Größtes Problem von Sperren: zwei TAs warten wechselseitig auf die Freigabe der jeweils anderen (bei 2PL führt das offensichtlich zu einem Deadlock)
- Zwei Arte:
  - Deadlock bzgl. unterschiedlichen Objekten:

    z.B. T_1 hält *X*-Sperre auf *x* und fordert *X*-Sperre auf *y* an

    T_2 hält *X*-Sperre auf *y* und fordert *X*-Sperre auf *x* an
  - Deadlock bzgl. einem einzigen Objekt durch Sperrenkonversion (Umwandlung einer R- in eine X-Sperre)



$X_1(y)$ muss auf Freigabe von $R_2(y)$ warten

$X_2(y)$ muss auf Freigabe von $R_1(y)$ warten

Update-Sperren

- Eine dritte Sperrenart: Update-Sperren

  => RUX-Verfahren bzw. RAX-Verfahren

  - *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)
- Lösung der Deadlockgefahr durch Sperrkonversionen (Deadlock bzgl. einem einzigen Objekt)

RUX-Sperrverfahren

- Verträglichkeit der Sperrentypen

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

- Kein Verhungern möglich, da spätere Leser keinen Vorrang haben
- Keine Konversionsverklemmung bzgl. einem einzigen Objekt
- Deadlocks bzgl. verschiedener Objekte bleiben weiterhin möglich

RAX-Sperrverfahren

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

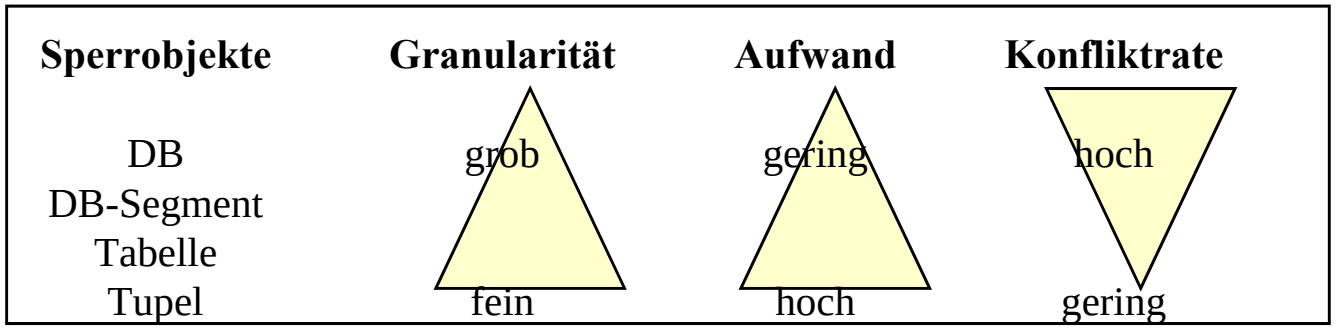
angeforderte Sperre \ *bestehende Sperre*	***R***	***A***	***X***
R	+	+	-
A	+	-	-
X	-	-	-

- Beim Konvertierungswunsch $A \rightarrow X$ Verhungern möglich

  (Warten bis alle *R*-Sperren aufgehoben sind, weitere *R*-Sperren aber jederzeit möglich)
- Trade-Off zwischen höherer Parallelität und Verhungern

3.3 Sperrverfahren (Locking)

Hierarchische Sperrverfahren



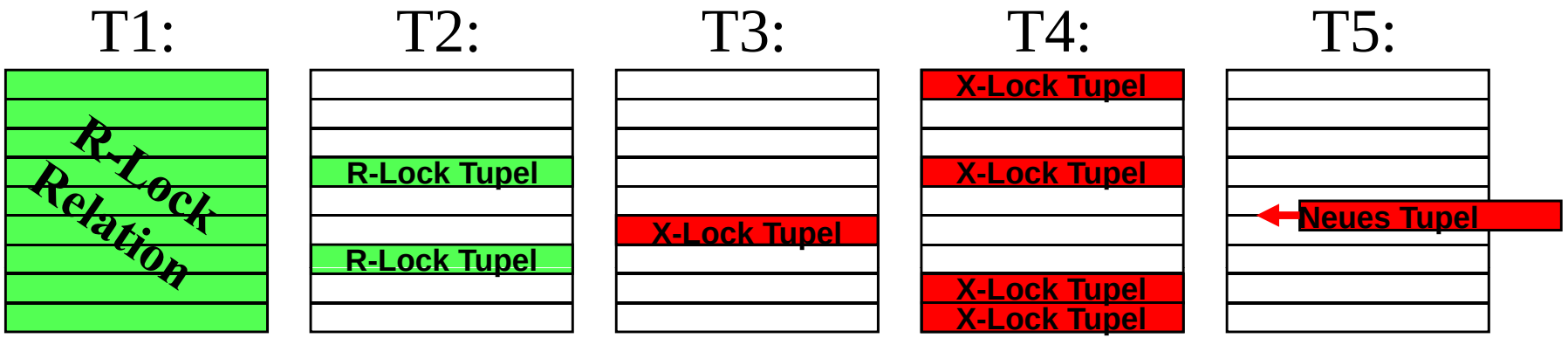
- Trade-Off
  Geringe Konfliktrate
  => hohe Parallelität
  => hoher Verwaltungsaufwand
- Lösung: variable Granularität durch hierarchische Sperren
- Kommerzielle DBS unterstützen zumeist 2-stufige Objekthierarchie, z.B. Segment-Seite oder Tabelle-Tupel
- Vorgehensweise bei hierarchischen Sperrverfahren: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

Hierarchische Sperrenverfahren (RIX)

- 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.

3.3 Sperrverfahren (Locking)

- Intuition von RIX



- 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

3.3 Sperrverfahren (Locking)

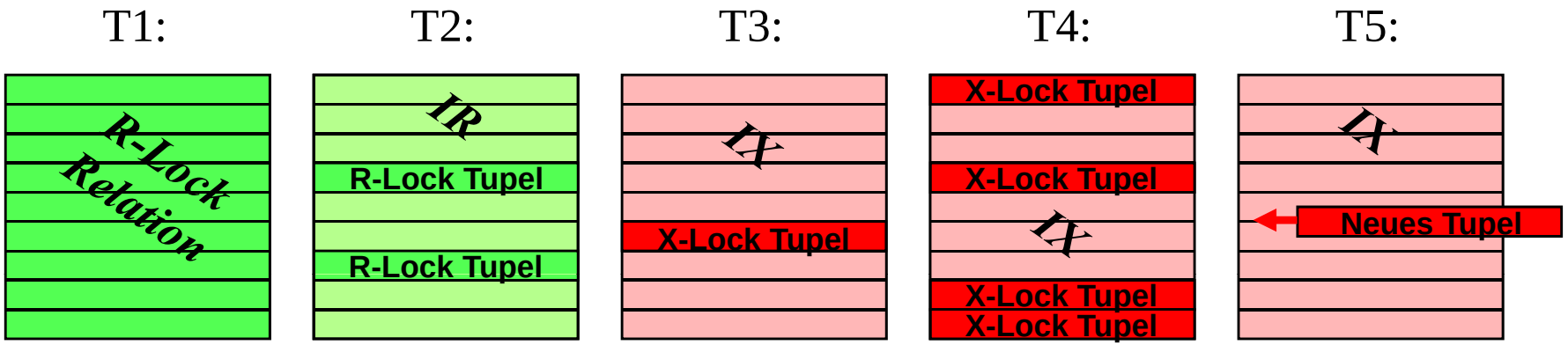
- Umsetzung: ***Intentionssperren***
  - IR-Sperre (intention read): auf feinerer Granularitätsstufe existiert (mindestens) eine R-Sperre
  - IX-Sperre (intention exclusive): auf feinerer Stufe X-Lock
  - RIX-Sperre (R-Sperre + IX-Sperre): volle Lesesperre und feinere Schreibsperre (sonst zu große Behinderung)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

3.3 Sperrverfahren (Locking)

- Beispiel



bestehende Sperre

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

Mehrkonversions Sperren (RAC)

- Prinzip

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)

Konsistenzstufen nach J. Gray

- Definition über die Dauer der Sperren:
  - lange Sperren: werden bis EOT gehalten (=> striktes 2PL)
  - kurze Sperren: werden nicht bis EOT gehalten

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

Konsistenzstufen nach J. Gray

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

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

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>
```

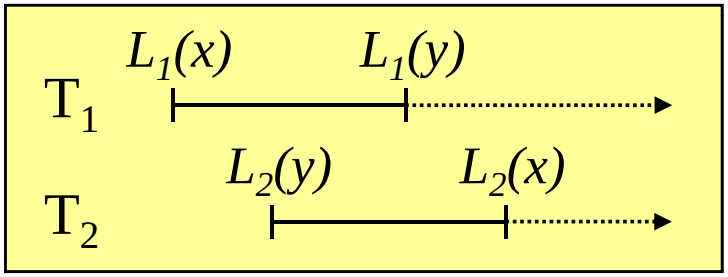
(Default `<level>` ist `SERIALIZABLE`)

3. Synchronisation

3.4 Behandlung von Verklemmungen

Verklemmung (Deadlock)

- Zwei Transaktionen warten gegenseitig auf die Freigabe einer Sperre (L = *LOCK*)
- Beispiel: Deadlock bzgl. zwei Objekten: $L_1(x)$, $L_2(y)$, $L_1(y)$, $L_2(x)$



T_1 wartet auf Freigabe von y durch T_2

T_2 wartet auf Freigabe von x durch T_1

- Wie wir gesehen haben können (je nach Sperrentyp und Sperrverträglichkeiten) auch Deadlocks bzgl. demselben Objekt entstehen (meist durch Sperrkonversionen)

3.4 Behandlung von Verklemmungen

Voraussetzungen für das Auftreten einer Verklemmung

(vgl. Betriebssysteme: Prozess-Synchronisation)

- 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

=> Letztlich alles Eigenschaften, die gewünscht sind und die wir daher auch garantieren wollen (und i.Ü. dem 2PL entsprechen)

Erkennen von Deadlocks

- Erkennen von Deadlocks über ***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
- Sattdessen: Heuristiken wie die ***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

Auflösung von Deadlocks

- Verletze eine der Voraussetzungen für das Auftreten von Deadlocks

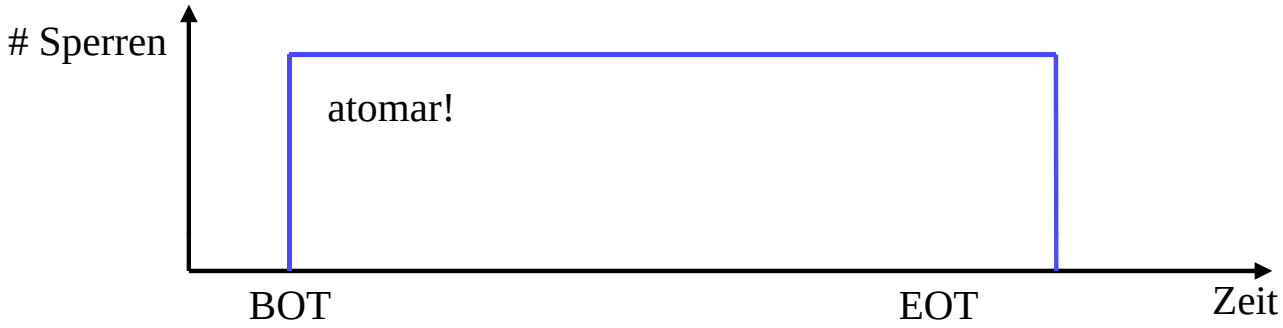
  Siehe Folie 60:

  - 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
- Nicht erwünscht, daher bleibt nur: ***Rücksetzen beteiligter TAs***
- Strategien:
  - 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 des Verhungerns (Starvation) von TAs: Setze früher bereits zurückgesetzte TAs möglichst nicht noch einmal zurück

3.4 Behandlung von Verklemmungen

Vermeidung von Deadlocks

- 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
  - Nachteile:
    - benötigte Sperren sind bei BOT typischerweise noch nicht bekannt, z.B. bei interaktiven TAs, Fallunterscheidungenm in TAs, dyn. Bestimmung der gesperrten Objekte
    - z. T. Abhilfe durch Sperren einer Obermenge der tatsächlich benötigten Objekte, ***ABER***: unnötige Ressourcenbelegung + Einschränkung der Parallelität

3.4 Behandlung von Verklemmungen

- Ordnung der Datenbank-Objekte
  - 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).
  - 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.
  - Beispiel:
    - TA fordert R1 an => R1 wird gesperrt
    - TA fordert R3 an => R2 und R3 werden gesperrt (R1 bleibt wegen 2PL weiterhin gesperrt => TA hat Sperre auf R1, R2, R3 !!!)

3.4 Behandlung von Verklemmungen

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

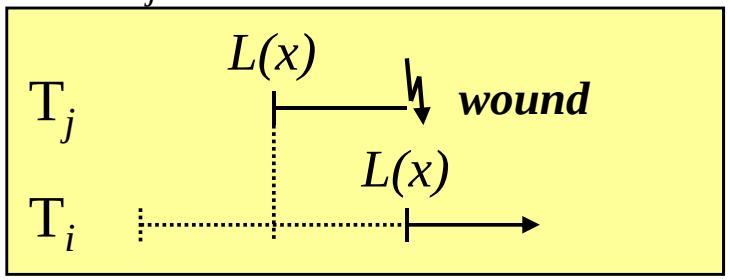
3.4 Behandlung von Verklemmungen

- ***wound-wait***

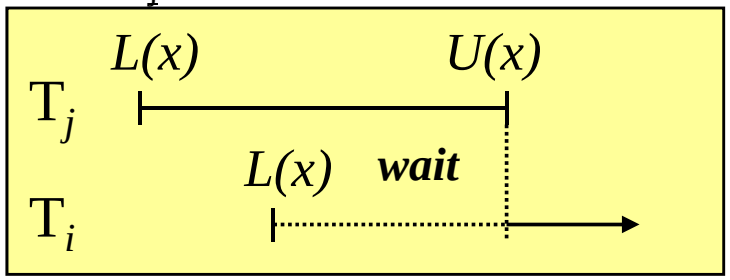
  T_i fordert Sperre $L(x)$ an.

  - 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***)

$TS(T_j) > TS(T_i)$



$TS(T_j) < TS(T_i)$



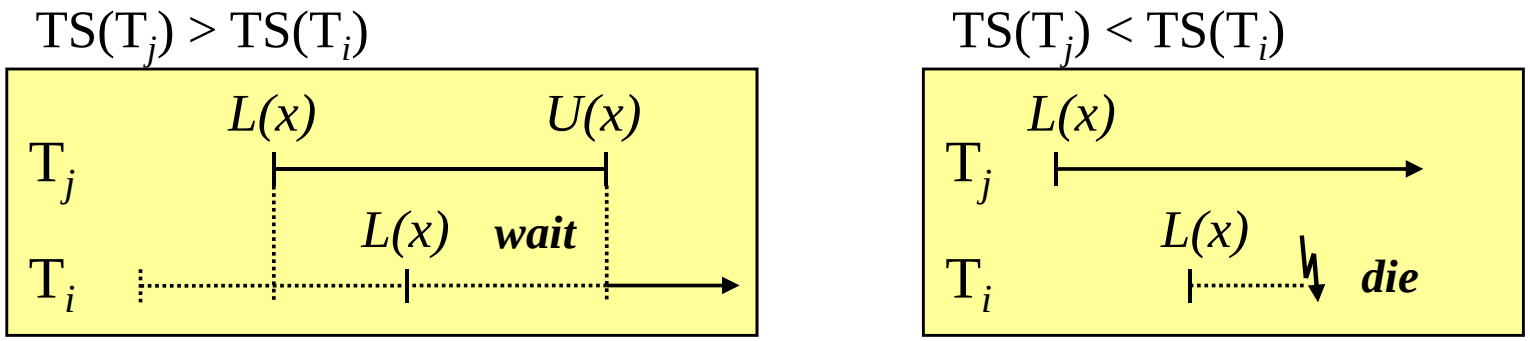
- → ältere TAs „bahnen“ sich ihren Weg durch das System

3.4 Behandlung von Verklemmungen

- ***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

3.4 Behandlung von Verklemmungen

- Eigenschaften: Wound-Wait ist ***Deadlock-frei***
  - Die Zeitstempel („Alter der Transaktion“) definieren eine strikte, totale Ordnung auf den Transaktionen:

    $TS(T_1) < TS(T_2) < TS(T_3) < \ldots < TS(T_n)$
  - 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
- 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 2PL
  - Rücksetzungen wesentlich häufiger als nötig
- Wait-Die: Analog (Pfeile im Wartegraphen nie von rechts nach links)

3. Synchronisation