Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Eigenschaften bzgl. Recovery

Was passiert wenn eine Transaktion t_i zurückgesetzt wird?

- Andere Transaktionen t_j dürfen davon nicht betroffen sein.

Weitere Klassen werden betrachtet:

- Rücksetzbare (Recoverable) Historien
- Historien ohne kaskadierendes Zurücksetzen

Schreib-/Leseabhängigkeiten

Kritisch für lokale Rücksetzbarkeit sind Schreib-/Leseabhängigkeiten:

$$w_j(x) \ \ldots\ldots \ r_i(x)$$

Definition: t_i liest von t_j in Historie s, wenn gilt

1. t_j schreibt mindestens ein Datum x, das t_i nachfolgend liest, d.h.

$$w_j(x) <_s r_i(x)$$

2. t_j wird (zumindest) nicht vor dem Lesevorgang von t_i zurückgesetzt:

$$a_j \not<_s r_i(x)$$

3. Alle anderen zwischenzeitlichen Schreibvorgänge auf x durch andere Transaktionen t_k werden vor dem Lesen durch t_i zurückgesetzt: Wenn $w_j(x) <_s w_k(x) <_s r_i(x)$, dann $a_k <_s r_i(x)$

$$s \ = \ \ldots w_j(x) \ldots\ldots \ w_k(x) \ldots\ldots \ a_k \ \ldots\ldots \ r_i(x)$$

Rücksetzbare Historie

Definition: Rücksetzbare Historie

Eine Historie s heißt rücksetzbar (recoverable, RC), falls für alle t_i, t_j, $i \neq j$ gilt: falls t_i von t_j liest, dann muss t_j vor t_i ihr Commit ausführen:

$$c_j <_s c_i$$

$$s = \ldots\ldots w_j(x) \ldots\ldots r_i(x) \ldots\ldots w_i(y) \ldots\ldots c_j \ldots\ldots a_i (\text{oder } c_i)$$

Beispiel: Dirty Read

$$s = w_1(x)\ r_2(x)\ c_2\ a_1$$

T_2 liest von T_1, aber kein c_1 vor c_2, also ist s nicht rücksetzbar!

Historie ohne kaskadierendes Rücksetzen

Kaskadierendes Rücksetzen

- abort verursacht Folge von weiteren aborts
- dies beeinträchtigt die Leistungsfähigkeit des Systems

Definition

Eine Historie **vermeidet kaskadierendes Rücksetzen** (avoids cascading aborts, ACA), wenn $c_j <_s r_i(x)$ gilt, wann immer t_i von t_j liest.

D.h., **Änderungen dürfen erst nach Commit freigegeben werden!**

Strikte Historien

Ist folgende Historie unproblematisch?

$$s \;=\; r_1(y)\; r_2(z)\; w_1(y)\; w_2(y)\; w_1(x)\; a_1\; r_2(x)\; c_2$$

- s ist serialisierbar: $G(s) = t_1 \rightarrow t_2$
- s ist ACA, da t_2 nicht von t_1 liest
- Problem: Abort-Behandlung von t_1 wird kompliziert

Definition: Strikte Historien

Eine Historie s ist **strikt**, wenn für je zwei Transaktionen t_i und t_j gilt: Wenn $w_j(x) <_s o_i(x)$ (mit $o_i = r_i$ oder $o_i = w_i$), dann muss gelten $c_j <_s o_i(x)$ oder $a_j <_s o_i(x)$

Zusammenfassung

- Bei ungeschützten und konkurrierenden Zugriffen von Lesern und Schreibern auf gemeinsame Daten können Anomalien entstehen.

Korrektheitskriterium der Synchronisation: Serialisierbarkeit

Gleicher DB-Zustand, gleiche Ausgabewerte wie bei seriellem Ablaufplan

Zusammenfassung (2)

- FSR erfüllt nicht einmal Minimalbedingungen
- VSR ist nicht monoton und Testen der VSR-Eigenschaft ist NP-vollständig!
- Im Gegensatz zu FSR und VSR ist CSR (Konflikt-Serialisierbarkeit) für praktische Anwendungen die wichtigste. Sie ist effizient überprüfbar.

  Es gilt: CSR $\subset$ VSR $\subset$ FSR
- Konfliktoperationen: Kritisch sind Operationen verschiedener Transaktionen auf denselben Daten, wenn diese Operationen nicht reihenfolgeunabhängig sind.
- Serialisierbarkeitstheorem: Sei s eine Historie; dann gilt $s \in$ CSR gdw $G(s)$ azyklisch
- Verschärfung durch OCSR und COCSR
- Weitere Klassen bzgl. Eigenschaften bei Recovery: ACA, RC, Strikt

Und nun?

- Scheduler beschreiben durch Protokolle, wie die einzelnen TA verzahnt ausgeführt werden.
- Optimistische oder pessimistische Scheduler
- Z.B. sperrbasiert (2PL) oder durch Zeitstempel
- Je nach Scheduler-Ansatz liegen dann die erzeugten Schedules in einer Klasse von Schedules, wie: 2PL erzeugt Schedules in CSR

Synchronisation - Übersicht

Entwurf des Schedulers

Klassifikation von Synchronisationsalgorithmen

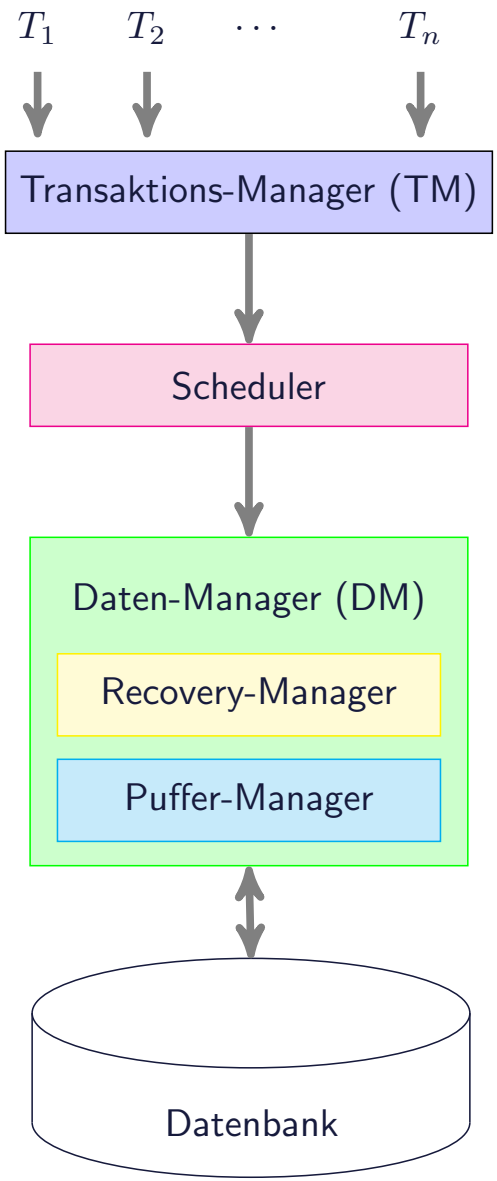
Sperrprotokolle

- Sperrregeln
- Zweiphasige Sperrprotokolle (2PL, C2P, S2PL, SS2PL)
- Deadlocks und ihre Behandlung

Nicht-sperrende Protokolle

- Zeitstempel-Verfahren

Der Datenbank-Scheduler



- TM: Informationen über TA, welche Schritte als nächstes ausgeführt werden können.
- Scheduler: Bekommt Schedules vom TM und muss diese in einen serialisierbaren Schedule umwandeln.
- Welcher verarbeitet wird von Daten-Manager (DM)

Scheduler

Entwurf des Schedulers

- Beschränkung auf Scheduler für **konfliktserialisierbare Schedules**
- vor allem: Richtlinien zum Entwurf von Scheduling-Protokollen und Verifikation gegebener Protokolle
- jedes Protokoll muss sicher (safe) sein, d.h., alle von ihm erzeugten Historien müssen in CSR sein
- Mächtigkeit des Protokolls (scheduling power): Kann es die vollständige Klasse CSR erzeugen oder nur eine echte Teilmenge davon?
- **Scheduling Power** ist ein Maß für den Parallelitätsgrad, den ein Scheduler nutzen kann.

Definition: CSR-Sicherheit

$Gen(S)$ bezeichnet die Menge aller Schedules, die ein Scheduler S erzeugen kann. S heißt CSR-sicher, wenn $Gen(S) \subseteq CSR$

Generischer Scheduler

```
var newstep : step:
{ state := initial_state;
  repeat
    on arrival (newstep) do
        update (state);
        if test (state, newstep)
        then output (newstep)
        else block (newstep) or reject (newstep) }
  forever };
```

Generischer Scheduler (2)

Scheduler-Aktionen

- **Ausgabe:** eine r, w, c oder a Eingabe wird direkt an das Ende des Ausgabe-Schedules geschrieben
- **Zurückweisung (reject):** auf eine r oder w Eingabe erkennt der Scheduler, dass die Ausführung dieses Schrittes die Serialisierbarkeit des Ausgabe-Schedules verhindern würde und initiiert mit der Zurückweisung den Abbruch a der entsprechenden TA.
- **Blockierung (block):** auf eine r oder w Eingabe erkennt der Scheduler, dass eine sofortige Ausführung des Schrittes die Serialisierbarkeit des Ausgabe-Schedules verhindern würde, eine spätere Ausführung jedoch noch möglich ist.
- DM führt die Schritte in der vom Scheduler vorgegebenen Reihenfolge aus.

Klassifikation von Protokollen (2)

Pessimistisch oder auch "konservativ"

- vor allem Sperrprotokolle
- einfach zu implementieren

optimistisch oder auch "aggressiv"

hybrid

- kombinieren Elemente von sperrenden und nicht-sperrenden Protokollen

Sperrprotokolle - Allgemeines

Allgemeine Idee

- **Zugriff auf gemeinsam genutzte Daten wird durch Sperren synchronisiert**
- hier: ausschließlich konzeptionelle Sichtweise und gleichförmige Granulate wie Seiten (keine Implementierungstechnik, keine multiplen Granulate usw.)

Allgemeine Vorgehensweise

- Scheduler fordert für die betreffende TA für jeden ihrer Schritte eine Sperre an
- Jede Sperre wird in einem spezifischen Modus angefordert (read oder write)
- Falls das Datenelement noch nicht in einem unverträglichen Modus gesperrt ist, wird die Sperre gewährt; sonst ergibt sich ein Sperrkonflikt und die TA wird blockiert, bis die Sperre freigegeben wird.

Sperrprotokolle - Allgemeines (2)

Kompatibilität und neuer Modus

angeforderte Sperre

aktueller Modus des Objekts x

	NL	R	X
rl(x)	+	+	-
wl(x)	+	-	-

neuer Modus des Objekts x

	NL	R	X
rl(x)	R	R	-
wl(x)	X	-	-

Sperrregeln

Allgemeine Sperrregeln (locking well-formedness rules)

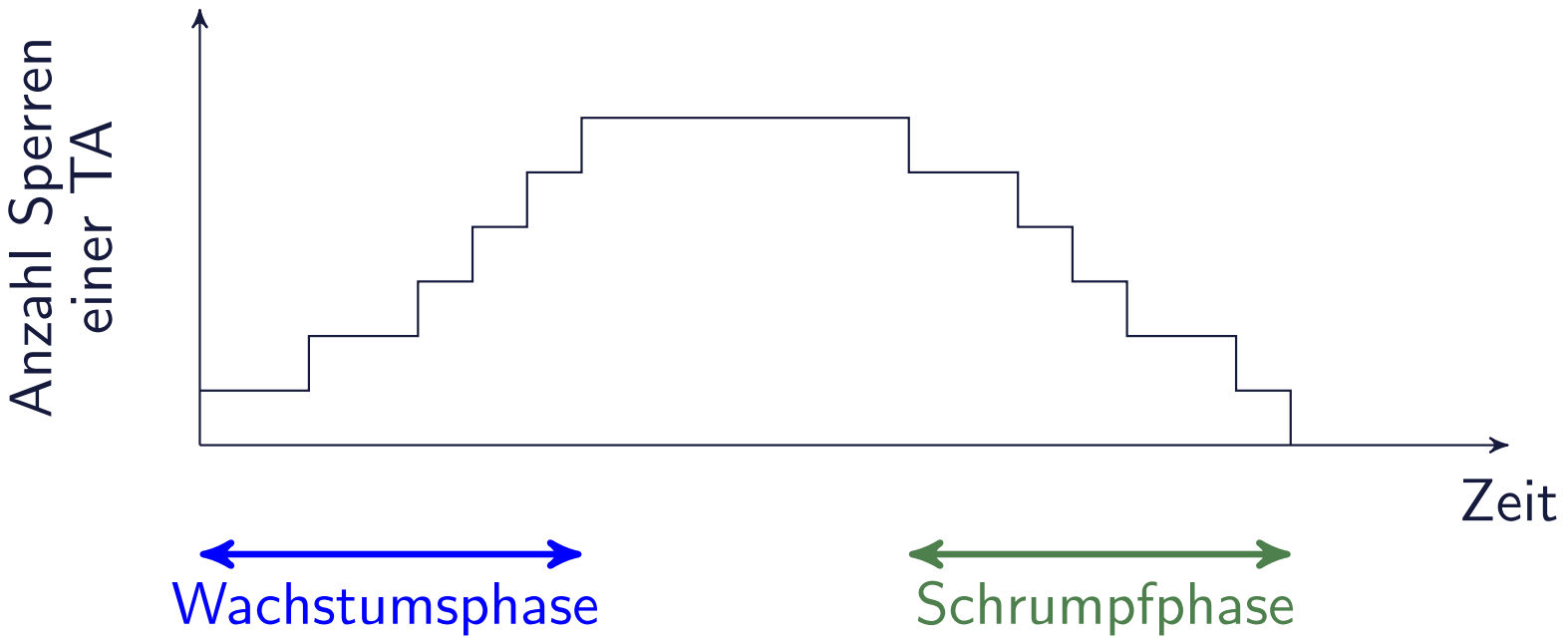
- LR1: Jeder Datenoperation $r_i(x)$ $[w_i(x)]$ muss ein $rl_i(x)$ $[wl_i(x)]$ (Sperre) vorausgehen und ein $ru_i(x)$ $[wu_i(x)]$ (Freigabe) folgen.
- LR2: Es gibt höchstens ein $rl_i(x)$ und ein $wl_i(x)$ für jedes x und t_i
- LR3: Es ist kein $ru_i(.)$ oder $wu_i(.)$ redundant
- LR4: Wenn x durch t_i und t_j gesperrt ist, dann sind diese Sperren kompatibel.

Anmerkungen: Reihenfolge der Freigaben spielt keine Rolle. Wenn sowohl Lesesperre auf Schreibsperre vorhanden, so reicht Freigabe der Schreibsperre (Lesesperre dann implizit aufgehoben).

2PL: Two Phase Locking

Definition: 2PL

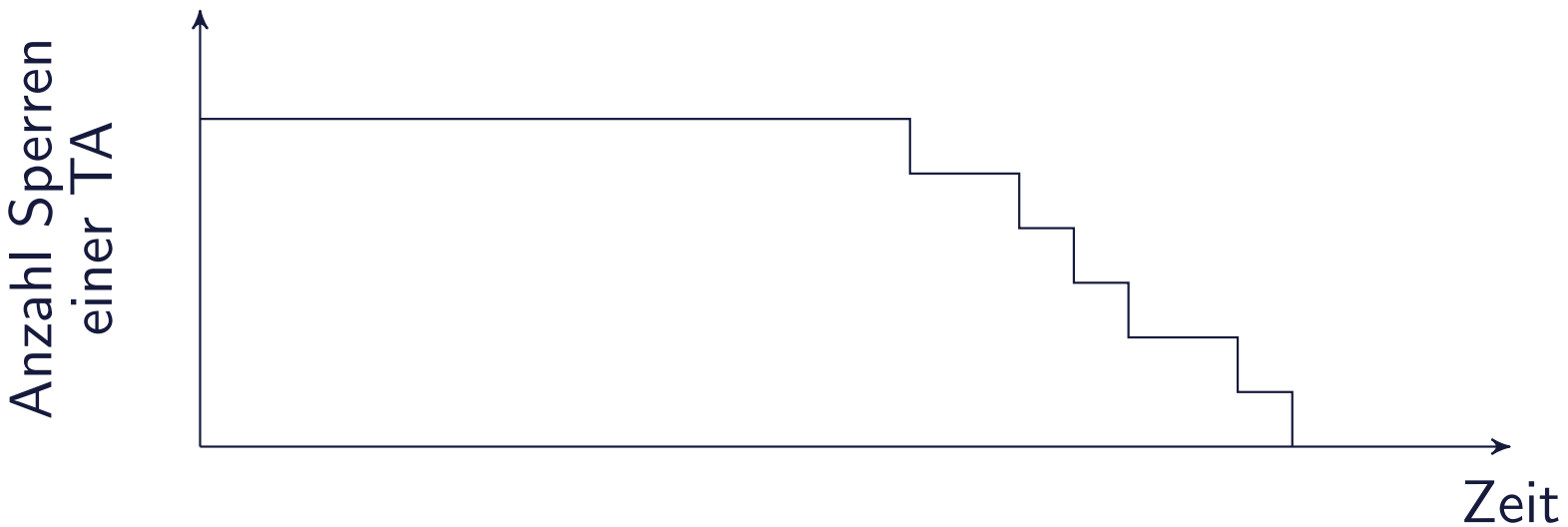
Ein Sperrprotokoll ist **zweiphasig** (2PL), wenn für jeden (Ausgabe-) Schedule s und jede TA $t_i \in trans(s)$ kein ql_i Schritt dem ersten ou_i Schritt folgt $(o, q \in \{r,w\})$.



Konservatives 2PL (C2PL)

Definition: Konservatives 2PL

Unter konservativem 2PL (C2PL) fordert jede TA alle Sperren an, bevor sie den erste Read- oder Write-Schritt ausführt (**Preclaiming**).
In der Praxis nur eingeschränkt anwendbar, da alle Sperren schon zu Beginn der TA bekannt sein müssen.

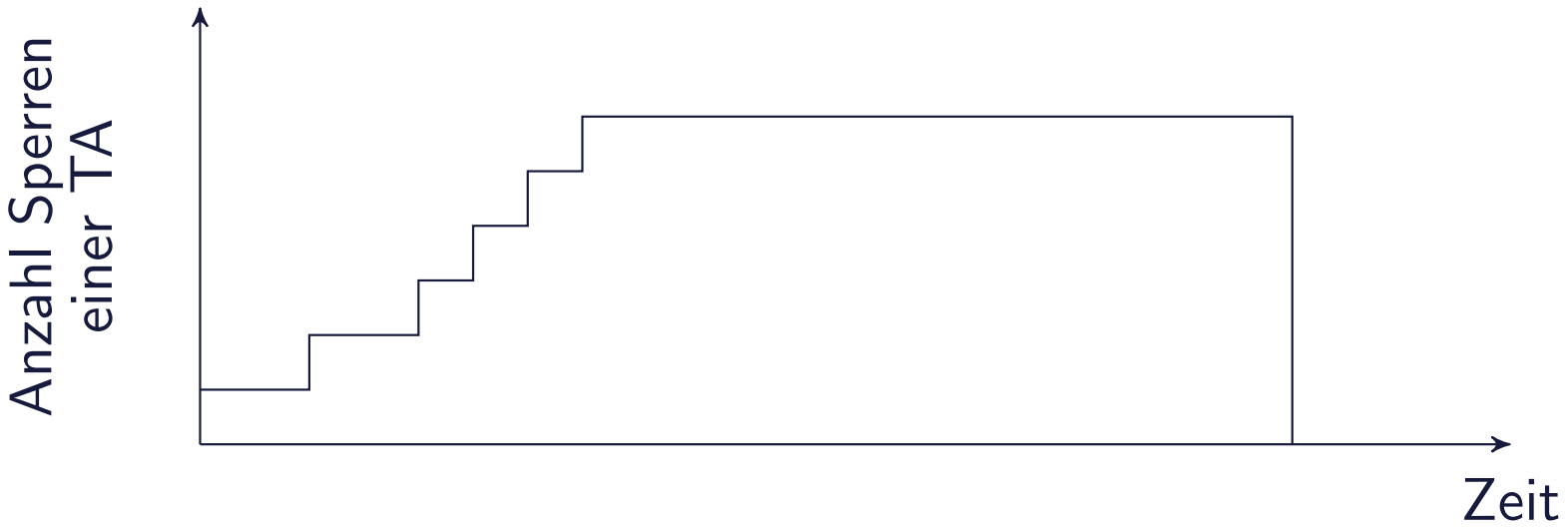


Striktes 2PL (S2PL) + Starkes 2PL (SS2PL)

Definition: Striktes 2PL

Unter striktem 2PL (S2PL) werden **alle exklusiven Sperren** (wl) einer TA bis zur ihrer Terminierung gehalten.

Wird in praktischen Implementierungen am häufigsten eingesetzt.



Definition: Starkes 2PL

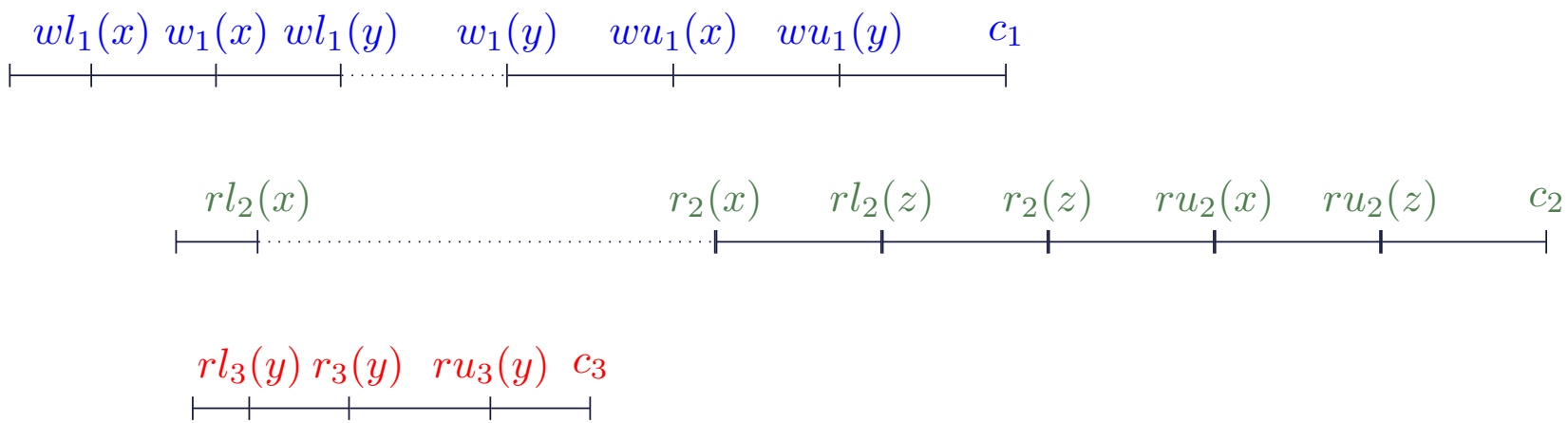
Unter starkem 2PL (strong 2PL, SS2PL) werden **alle Sperren** (wl, rl) einer TA bis zur ihrer Terminierung gehalten.

Sperrprotokoll 2PL

Beispiel: Eingabe-Schedule

$$s_1 = w_1(x)\ r_2(x)\ r_3(y)\ r_2(z)\ w_1(y)\ c_3\ c_1\ c_2$$

2PL Scheduler transformiert s_1 z.B. in folgende Ausgabe-Historie:



Theorem

Ein 2PL-Scheduler ist CSR-sicher, d.h., $Gen(\text{2PL}) \subset CSR$

Beispiel

- $s_2 = w_1(x)\ r_2(x)\ c_2\ r_3(y) c_3\ w_1(y)\ c_1$
- $s_2 \approx_c t_3\ t_1\ t_2\ \in CSR$

aber $s_2 \notin Gen(\text{2PL})$

- s_2 kann nicht in $Gen(\text{2PL})$ sein, da der Scheduler überprüft:
- $wu_1(x) < rl_2(x)$ (durch Warten) und $ru_3(y) < wl_1(y)$ (Kompatibilitätsregel)
- $rl_2(x) < r_2(x)$ und $r_3(y) < ru_3(y)$ (Wohlgeformtheitsregel)
- $r_2(x) < r_3(y)$ (aus dem Schedule)
- c_2 vor c_3 würde $wu_1(x) < wl_1(y)$ implizieren, was der 2PL-Eigenschaft (Zweiphasigkeit) widerspricht.

Beispiel ...

$$s_2 = w_1(x)\ r_2(x)\ c_2\ r_3(y)c_3\ w_1(y)\ c_1$$

s_2 wird durch 2PL-Scheduler transformiert zu

$$wl_1(x)\ w_1(x)\ rl_2(x)^*\ rl_3(y)\ r_3(y)\ ru_3(y)\ c_3\ wl_1(y)\ w_1(y)$$

$$wu_1(x)\ rl_2(x)^*\ r_2(x)\ wu_1(y)\ c_1\ ru_2(x)\ c_2$$

also zu

$$s_2' = w_1(x)\ r_3(y)\ c_3\ w_1(y)\ r_2(x)\ c_1\ c_2$$

***)Anmerkung:** $rl_2(x)$ bedeutet hier: Anfrage auf Lock auf x von TA 2, später folgt dann erst die Gewährung ($rl_2(x)$) der Sperre.

2PL, S2PL, SS2PL

Verfeinerung

- Das Beispiel zeigt: eine von einem 2PL-Scheduler erzeugte Historie ist eine hinreichende, aber keine notwendige Bedingung für CSR
- Dies lässt sich auf OCSR verfeinern.

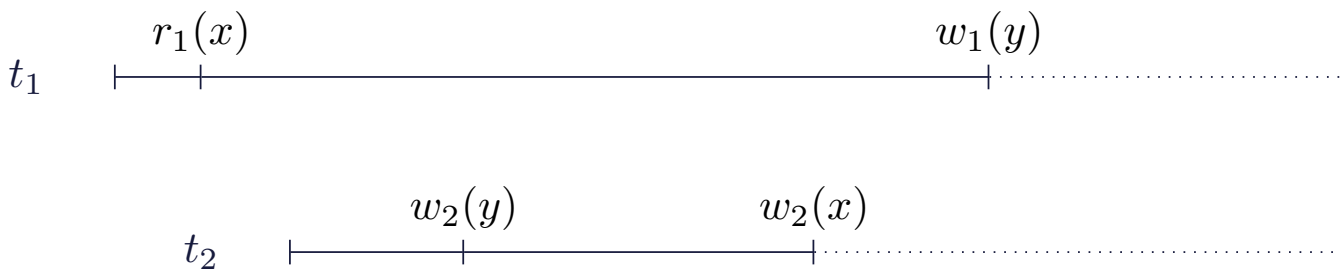
Theorem: $Gen(\textbf{2PL}) \subset OCSR$

Theorem: $Gen(\textbf{SS2PL}) \subset Gen(\textbf{S2PL}) \subset Gen(\textbf{2PL})$

Theorem: $Gen(\textbf{SS2PL}) \subset COCSR$

Deadlocks

- werden verursacht durch zyklisches Warten auf Sperren
- Beispiel



Deadlock-Erkennung

Aufbau eines dynamischen Wait-for-Graph (WfG) mit aktiven TAs als Knoten und Wartebeziehungen als Kanten: Eine Kante von t_i nach t_j ergibt sich, wenn t_i auf eine von t_j gehaltene Sperre wartet.

Testen des WfG zur Zyklenerkennung

- kontinuierlich (bei jedem Blockieren)
- periodisch (z.B. einmal pro Sekunde)

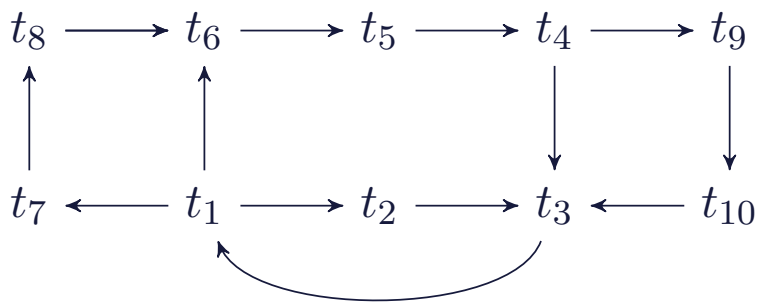
Deadlock Auflösung

- Wähle eine TA in einem WfG-Zyklus aus
- Setze diese TA zurück
- Wiederhole diese Schritte, bis keine Zyklen mehr gefunden werden

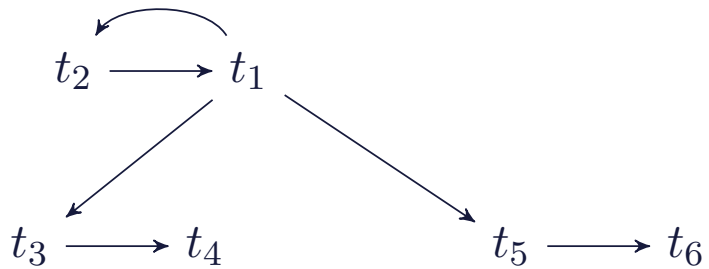
Mögliche Strategien zur Bestimmung von "Opfern"

- Zuletzt blockierte TA
- Zufällige TA
- Jüngste TA
- Minimale Anzahl von Sperren
- Minimale Arbeit (geringster Ressourcen-Verbrauch, z.B. CPU-Zeit)
- Meiste Zyklen
- Meiste Kanten

Deadlock Auflösung - Beispiele



Meiste-Zyklen-Strategie würde t_1 (oder t_3) auswählen, um alle 5 Zyklen aufzubrechen.



Meiste-Kanten-Strategie würde t_1 auswählen, um 4 Kanten zu entfernen.

Strategien zur Deadlock-Vermeidung

Idee: Blockierungen (lock waits) werden eingeschränkt, so dass **stets** ein azyklischer WfG gewährleistet werden kann.

Strategien

- **Wait-Die:** Sobald Sperranforderung von t_i zu Konflikt mit t_j führt: Wenn t_i vor t_j gestartet ist (also älter ist), dann **wait**(t_i), sonst **restart**(t_i).
  "TA wird nur von jüngeren blockiert"
- **Wound-Wait:** Sobald t_i mit t_j in Konflikt gerät: Wenn t_i vor t_j, dann **restart**(t_j) sonst **wait**(t_i)
  "TA kann nur von älteren blockiert werden und TA kann den Abbruch von jüngeren, mit denen sie in Konflikt gerät, verursachen"

Weitere Strategien zur Deadlock-Vermeidung

- **Immediate Restart:** Sobald t_i mit t_j in Konflikt gerät: **restart**(t_i)
- **Running Priority:** Sobald t_i mit t_j in Konflikt gerät: wenn t_j selbst blockiert ist, dann **restart**(t_j) sonst **wait**(t_i)
- **Konservative Ansätze:** TA, die zurückgesetzt wird, ist nicht notwendigerweise in einen Deadlock involviert.
- **Timeout:** Wenn Timer ausläuft, wird t zurückgesetzt in der Annahme, dass t in einen Deadlock involviert ist.

Zeitstempelverfahren

Grundsätzliche Idee

- Transaktion (TA) bekommt bei BOT einen systemweit eindeutigen Zeitstempel; er legt die Serialisierbarkeitsreihenfolge fest.
- TA hinerlässt den Wert ihres Zeitstempels auf jedem Objekt o_i auf das sie zugreift

Timestamp Ordering (TO)

- Jeder TA t_i wird ein eindeutiger Zeitstempel $ts(t_i)$ zugewiesen
- Zentrale TO-Regel: Wenn $p_i(x)$ und $q_j(x)$ in Konflikt stehen, dann muss für jeden Schedule s folgendes gelten:

$$p_i(x) <_s q_j(x) \text{ gdw } ts(t_i) < ts(t_j)$$

Zeitstempelverfahren (2)

Theorem: $Gen(TO) \subseteq CSR$

Prinzipielle Arbeitsweise

- Vergabe von aufsteigend eindeutigen TA IDs (Zeitstempel ts der TA)
- "Stempeln" des Objektes x bei Zugriffen t_i: $TS(x) := ts(t_i)$

Read und Write Timestamps

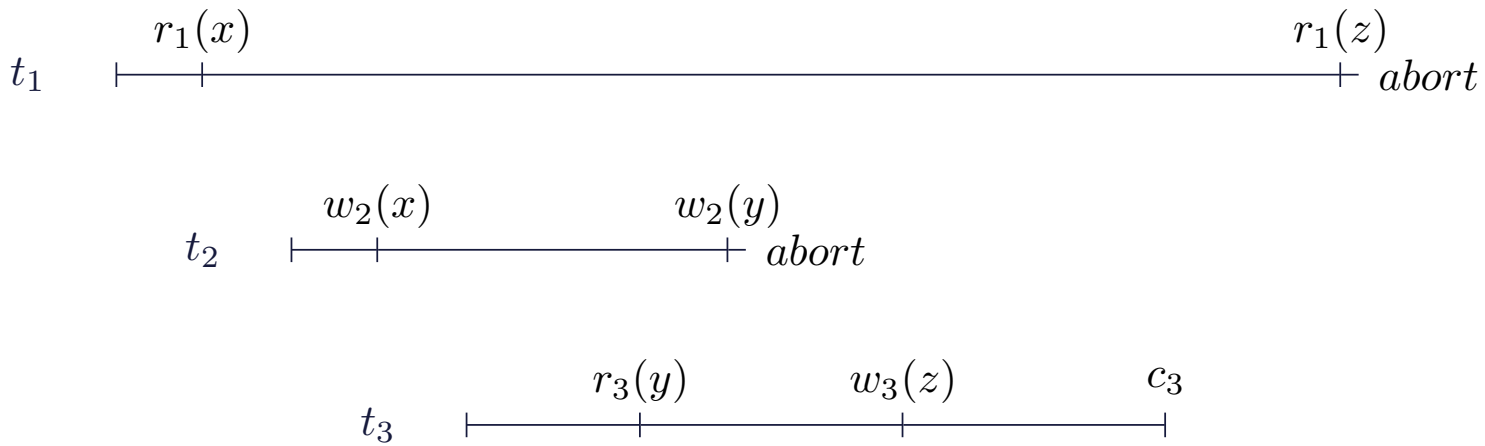
Der TO Scheduler muss für jedes Objekt x die beiden folgenden Zeitstempel speichern:

1. **max-r-scheduled(x)**: der Wert des größten Zeitstempels einer Leseoperation auf x
2. **max-w-scheduled(x)**: den Wert des größten Zeitstempels einer Schreiboperation auf x

Vorgehensweise TO Scheduler

Wenn eine Operation $p_i(x)$ im Scheduler ankommt, wird $ts(t_i)$ verglichen mit max-q-scheduled(x) für jede Operation q, die mit p in Konflikt ist.

- **Falls $ts(t_i) <$ max-q-scheduled(x): $p_i(x)$ wird zurückgewiesen (rejected)**
- **Ansonsten:** $p_i(x)$ wird erlaubt (und an Datenmanager geschickt) und max-p-scheduled(x) wird auf den Zeitstempel $ts(t_i)$ gesetzt falls $ts(t_i) >$ max-p-scheduled(x)



Regeln und Thomas' Write Rule

Abk. $i := ts(t_i)$

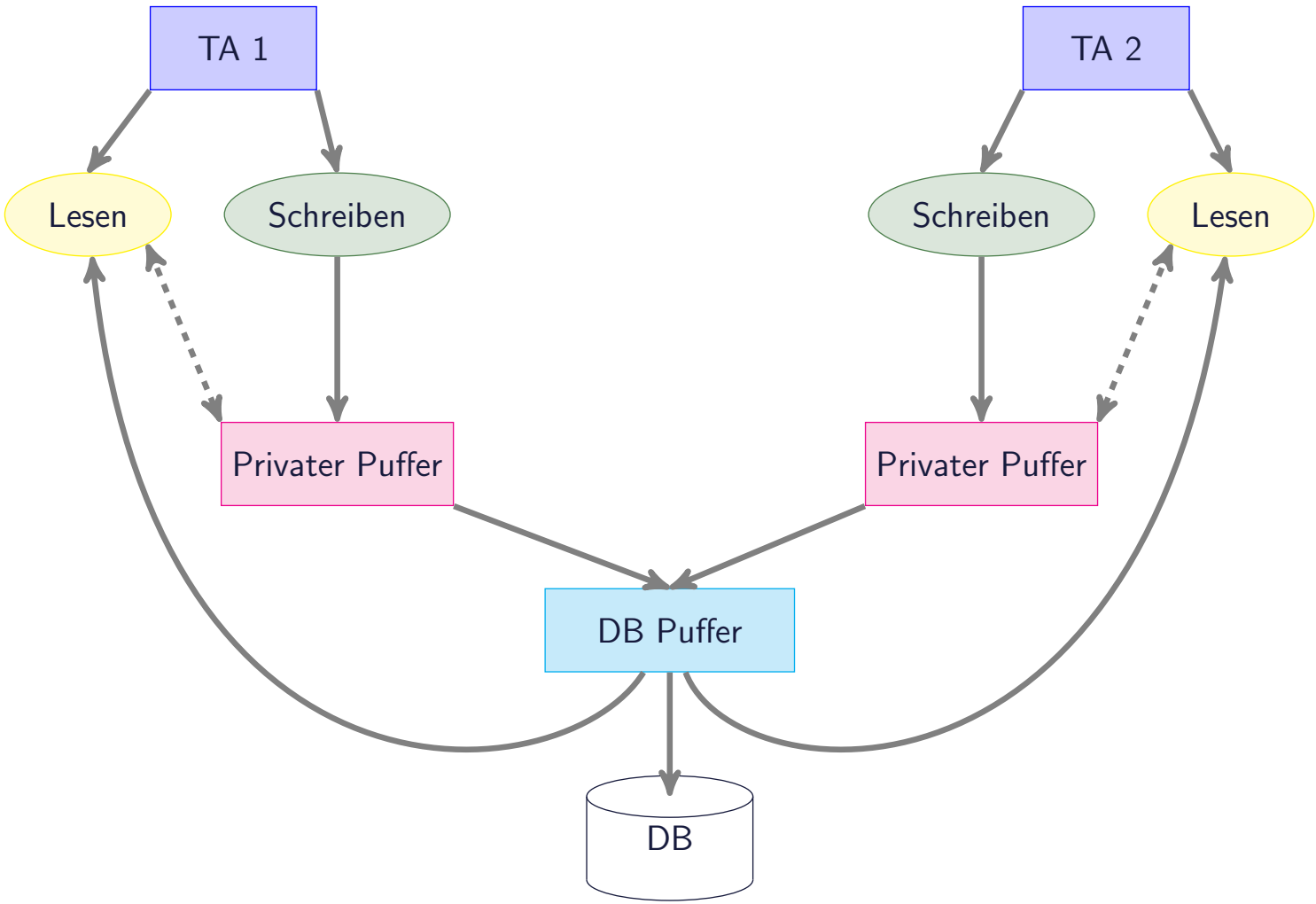
R1	$r_i \wedge (i \geq$ max-w$(x))$	if max-r$(x) < i$ then max-r$(x) := i$; Lesen
R2	$w_i \wedge (i \geq$ max-r$(x)) \wedge (i \geq$ max-w$(x))$	max-w$(x) := i$; Schreiben
R3	$w_i \wedge (i \geq$ max-r$(x)) \wedge (i <$ max-w$(x))$	Ignoriere Schreiben(*)
R4	$w_i \wedge (i <$ max-r$(x))$	Zurücksetzen!
R5	$r_i \wedge (i <$ max-w$(x))$	Zurücksetzen!

(*) Thomas' Write Rule

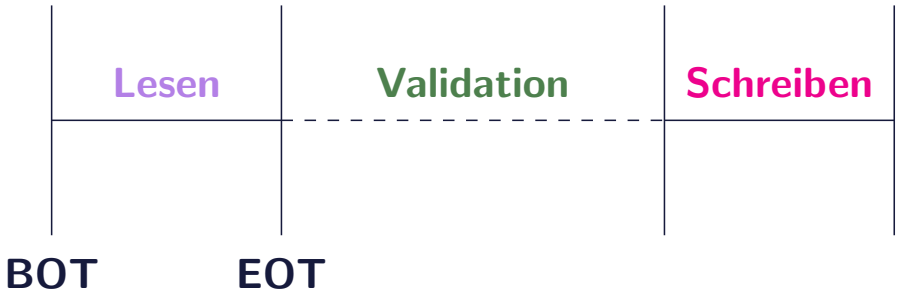
- Optimierung/Reduzierung von Schreibkonflikten
- Idee: Last-write-wins
- Gegeben TA t_i mit Zeitstempel $ts(t_i)$
- Falls $ts(t_i) \geq$ max-r-scheduled(x) und $ts(t_i) <$ max-w-scheduled(x), dann ignoriere Schreiben $w_i(x)$.

Optimistic Concurrency Control (OCC)

Idee: Transaktionen laufen erstmal getrennt voneinander mittels privatem Puffer ab. Anschließend wird validiert ob es Konflikte zwischen den TAs gibt.



Optimistic Concurrency Control (OCC)



Lesephase

- eigentliche TA-Verarbeitung
- Änderungen einer Transaktion werden in privatem Puffer durchgeführt

Schreibphase

- nur bei positiver Validierung
- Lese-Transaktion ist ohne Zusatzaufwand beendet
- Schreib-Transaktionen schreiben hinreichende Log-Information und propagieren Änderungen

Optimistic Concurrency Control (OCC)

Validierungsphase

- Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer parallel laufenden Transaktionen passiert.
- Konfliktauflösung durch Zurücksetzen von Transaktionen

OCC: Validierungs Prinzipien BOCC und FOCC

Backward Oriented (BOCC)

Validierung gegenüber bereits beendeten (Änderungs-) TAs

Forward Oriented (FOCC)

Validierung gegenüber laufenden TA

Details: Theo Härder: Observations on optimistic concurrency control schemes. Inf. Syst. 9(2): 111-120 (1984)
`http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/publications/1984/Hae84.InformationSystems.pdf`

Sonstiges: Überblick

Weitere Ansätze/Prinzipien

- **RUX**
- **Hierarchische Sperrverfahren**
- **MVCC: Multi-Version-Concurrency-Control**
- **Snapshot Isolation**