

DIS'2010

-

5. Synchronisation – Algorithmen I

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Norbert Ritter

Datenbanken und Informationssysteme

vsis-www.informatik.uni-hamburg.de



© N. Ritter

Scheduling-Algorithmen – Scheduler (1)

■ Entwurf von Scheduling-Algorithmen (Scheduler)

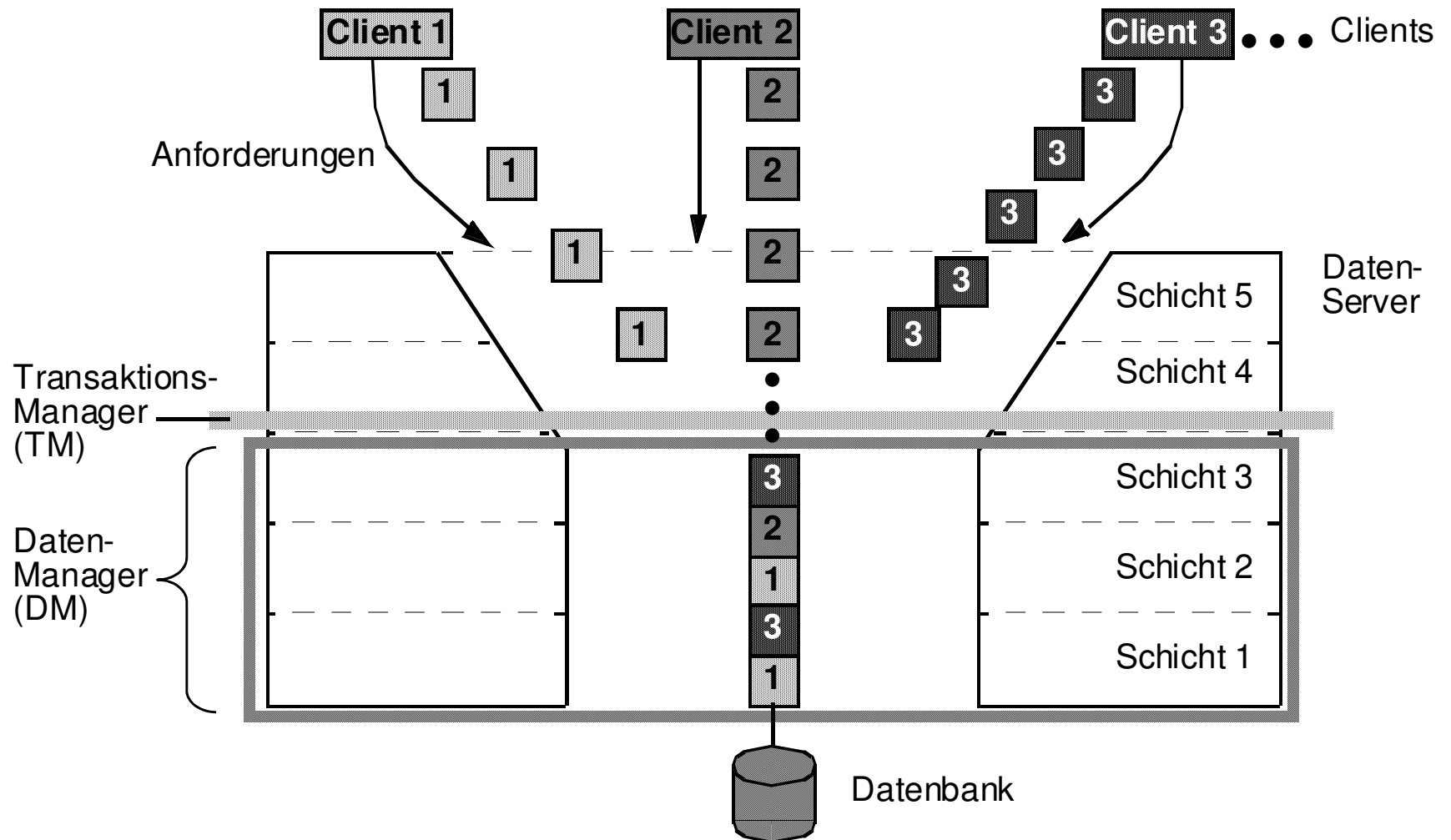
- Beschränkung auf Scheduler für konfliktserialisierbare Schedules
- vor allem: Richtlinien zum Entwurf von Scheduling-Protokollen und Verifikation gegebener Protokolle
- jedes Protokoll muss sicher (safe) sein, d.h., alle von ihm erzeugten Historien müssen in CSR sein
- Mächtigkeit des Protokolls (scheduling 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*

- $\text{Gen}(s)$ bezeichnet die Menge aller Schedules, die ein Scheduler S generieren kann. S heißt CSR-sicher, wenn $\text{Gen}(s) \subseteq \text{CSR}$

Scheduling-Algorithmen – Scheduler (2)

■ Allgemeiner Entwurf



Scheduling-Algorithmen – Scheduler(3)

- Allgemeiner Entwurf (Forts.)
 - Transaktions-Manager (TM)
 - empfängt Anforderungen und leitet die erforderlichen Schritte für die Synchronisation (concurrency control) und Recovery ein
 - ist typischerweise zwischen den Schichten des Datensystems und Zugriffssystems oder denen des Zugriffssystems und Speichersystems angeordnet
 - Schichten unterhalb des TM, auch Daten-Manager (DM) genannt, sind für den TM nicht relevant und können als eine „virtuelle“ System-Komponente aufgefasst werden

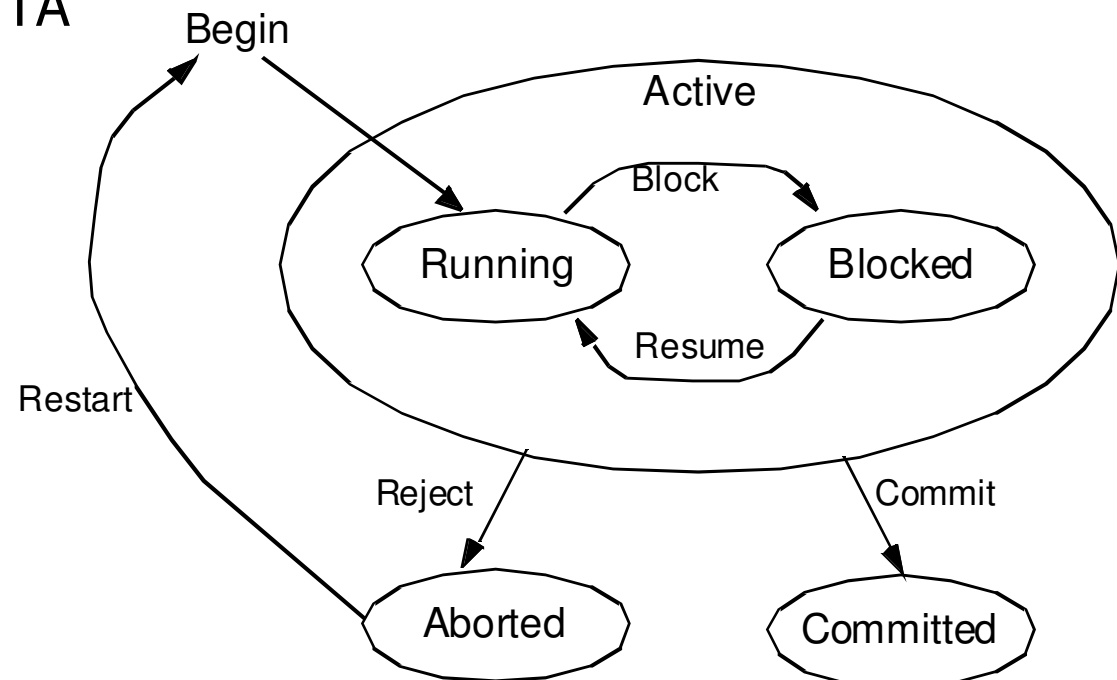
Scheduling-Algorithmen – Scheduler(4)

■ Allgemeiner Entwurf (Forts.)

• Dynamischer Ablauf

- TM verwaltet vor allem die Listen trans, commit, abort und active und eine Liste der ausführungsbereiten Schritte
- Scheduler erhält von TM einen willkürlichen Eingabe-Schedule und hat ihn zu einem serialisierbaren Ausgabe-Schedule zu transformieren
- TM schickt die TA-Schritte c_i und a_i an den Scheduler

• Zustände einer TA



Scheduling-Algorithmen – Scheduler(5)

■ Allgemeiner Entwurf (Forts.)

- 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 Transaktion
 - 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

Scheduling-Algorithmen – Scheduler(5)

■ Allgemeiner Entwurf (Forts.)

- Generischer Scheduler

scheduler ():

var newstep : step;

{state := initial_state;

repeat

on arrival (newstep) do

{update (state);

if test (state, newstep)

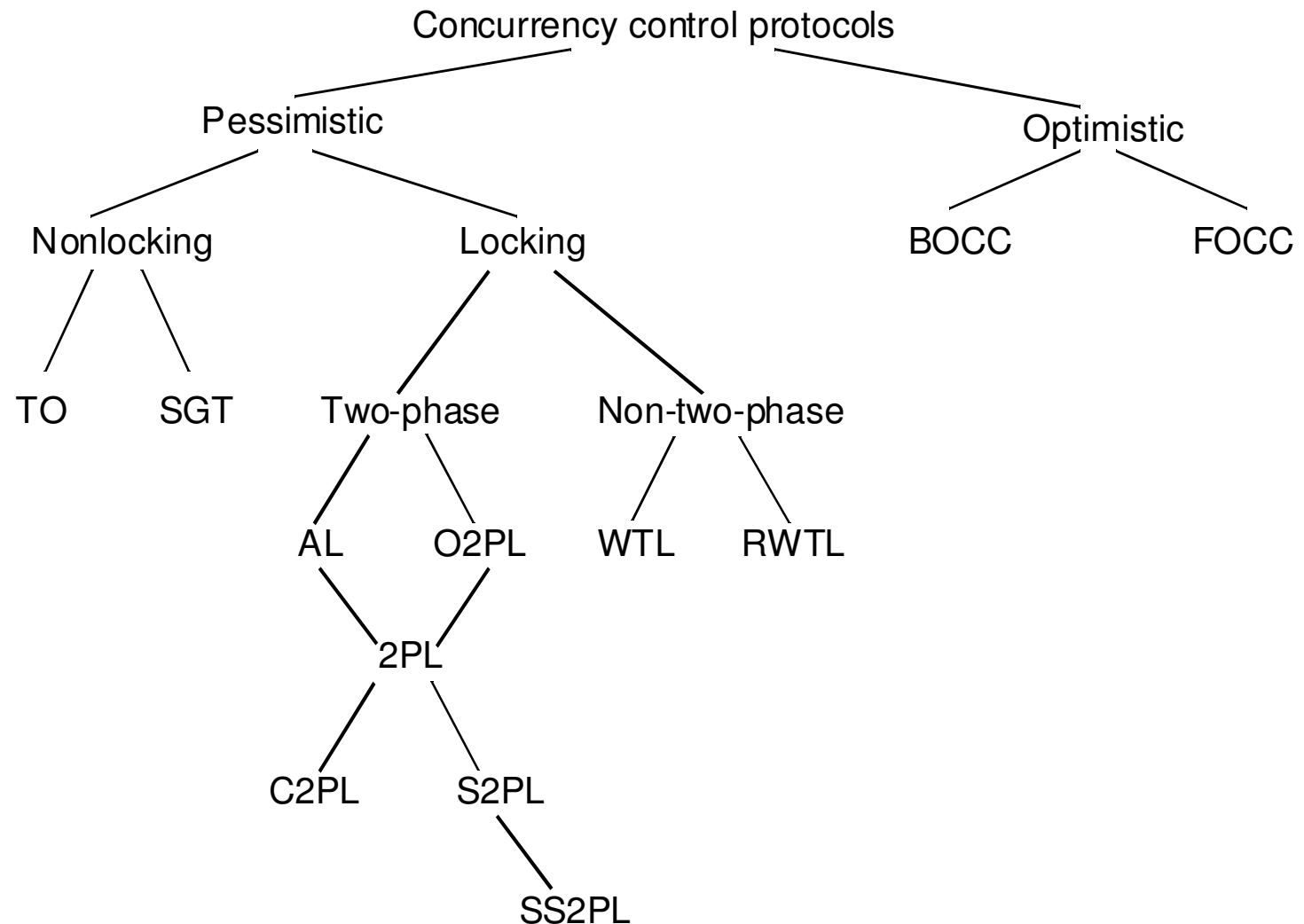
then output (newstep)

else block (newstep) or reject (newstep) }

forever };

Klassifikation von Protokollen (1)

■ Klassifikation von Protokollen



Klassifikation von Protokollen (2)

■ Klassifikation von Protokollen (Forts.)

- pessimistisch oder auch „konservativ“
 - vor allem: Sperrprotokolle; sie sind meist allen anderen Protokollen in ihrem Leistungsverhalten überlegen
 - einfach zu implementieren
 - erzeugen nur geringen Laufzeit-Aufwand
 - können für die Anwendung bei verschiedenen TA-Modellen generalisiert werden
 - sie können beim Seiten-Modell und beim Objekt-Modell angewendet werden
- optimistisch oder auch „aggressiv“
- hybrid: kombinieren Elemente von sperrenden und nicht-sperrenden Protokollen

Sperrprotokolle - Allgemeines (1)

■ Allgemeine Idee

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

■ Allgemeine Vorgehensweise

- Der 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

		Angeforderte Sperre	
Gehaltene Sperre		$rl_j(x)$	$wl_j(x)$
	$rl_i(x)$	+	-
	$wl_i(x)$	-	-

■ Allgemeine Sperregeln (locking well-formedness rules)

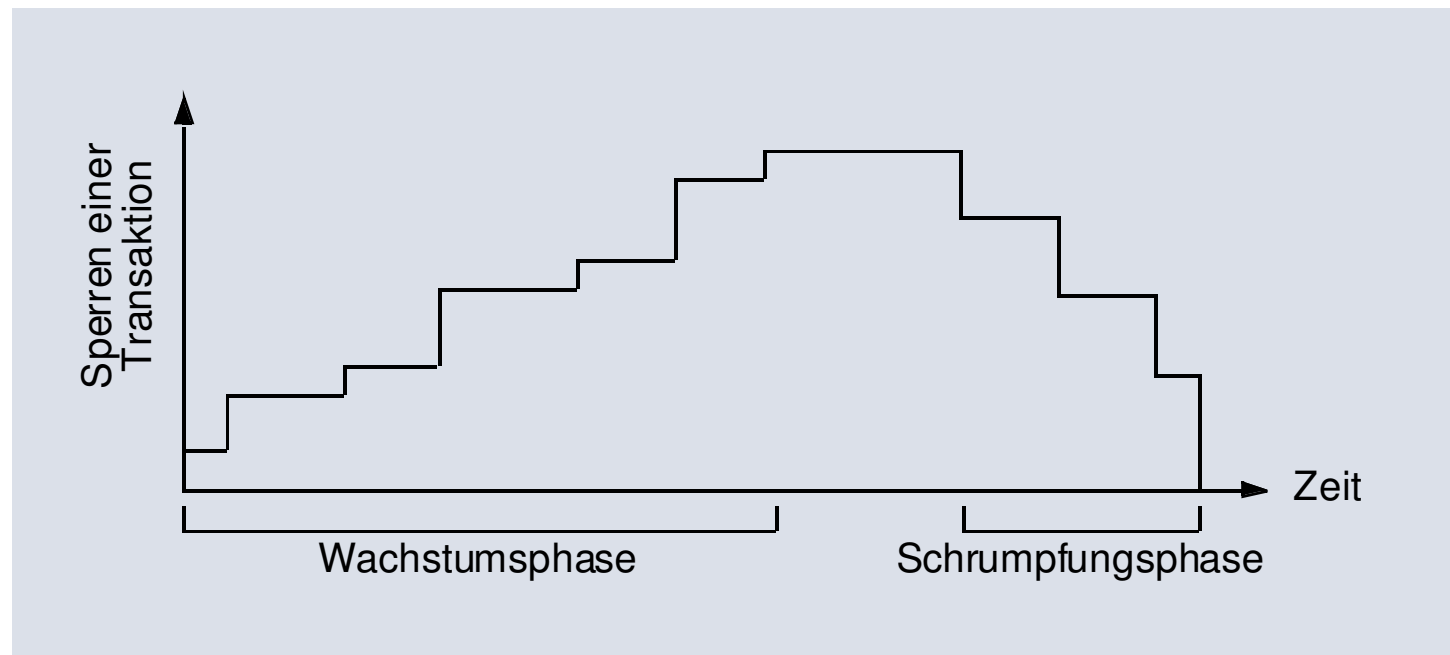
- LR1: Jeder Datenoperation $r_i(x)$ [$w_i(x)$] muss ein $rl_i(x)$ [$wl_i(x)$] vorausgehen und ein $ru_i(x)$ [$wu_i(x)$] folgen
- LR2: Es gibt höchstens ein $rl_i(x)$ und ein $wl_i(x)$ für jedes x und t_i
- LR3: Es ist kein $ru_i(.)$ oder $wu_i(.)$ redundant
- LR4: Wenn x durch t_i und t_j gesperrt ist,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

Sperrprotokolle – 2PL (3)

■ Definition **2PL**

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

■ Ausgabe eines 2PL-Schedulers



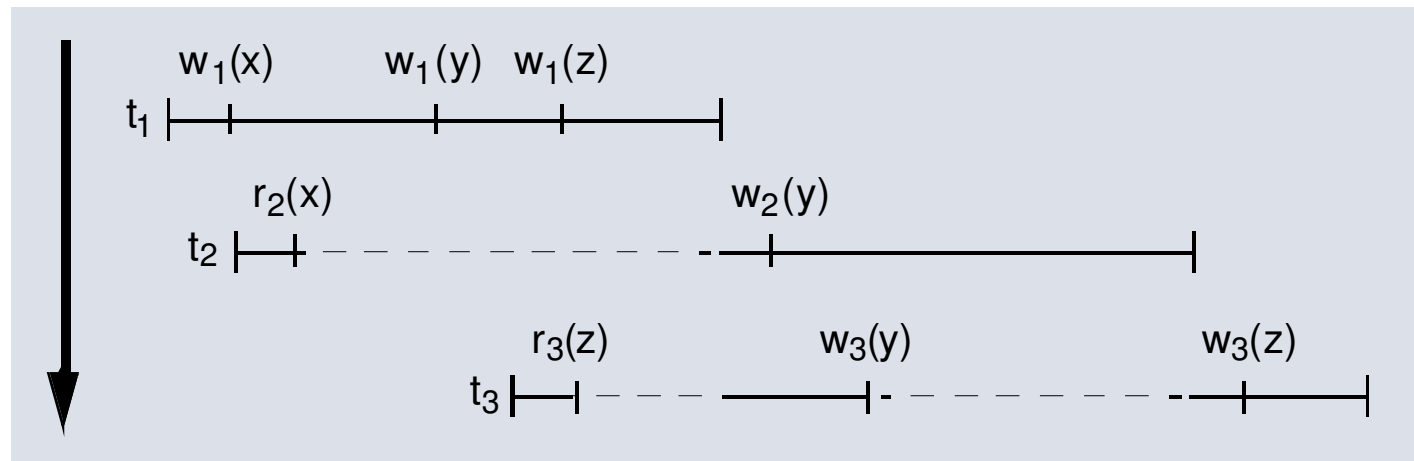
Sperrprotokolle – 2PL (4)

■ Beispiel

- Eingabe-Schedule

- $s = w_1(x) \ r_2(x) \ w_1(y) \ w_1(z) \ r_3(z) \ c_1 \ w_2(y) \ w_3(y) \ c_2 \ w_3(z) \ c_3$

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



- $wl_1(x) \ w_1(x) \ wl_1(y) \ w_1(y) \ wl_1(z) \ w_1(z) \ wu_1(x) \ rl_2(x) \ r_2(x) \ wu_1(y) \ wu_1(z) \ c_1 \ rl_3(z) \ r_3(z) \ wl_2(y) \ w_2(y) \ wu_2(y) \ ru_2(x) \ c_2 \ wl_3(y) \ w_3(y) \ wl_3(z) \ w_3(z) \ wu_3(z) \ wu_3(y) \ c_3$

Sperrprotokolle – 2PL (5)

■ Theorem

- Ein 2PL-Scheduler ist CSR-sicher, genauer: $\text{Gen}(2\text{PL}) \subset \text{CSR}$

■ Beispiel

- $s = w_1(x) \ r_2(x) \ c_2 \ r_3(y) \ c_3 \ w_1(y) \ c_1$
- $s \approx_c t_3 \ t_1 \ t_2 \in \text{CSR}$
- aber $s \notin \text{Gen}(2\text{PL})$, da
 - $wu_1(x) < rl_2(x)$ und $ru_3(y) < wl_1(y)$,
(Kompatibilitätsanforderung)
 - $rl_2(x) < r_2(x)$ und $r_3(y) < ru_3(y)$,
(Wohlgeformtheitsregeln)
 - und $r_2(x) < r_3(y)$
(aus dem Schedule)
 - würde (über Reihenfolgebeschränkungen und Transitivität)
 $wu_1(x) < wl_1(y)$ implizieren, was der 2PL-Eigenschaft widerspricht

Sperrprotokolle – 2PL (6)

■ Verfeinerung

- Das Beispiel zeigt: die Tatsache, dass eine Historie von einem 2PL-Scheduler erzeugt wurde, ist eine hinreichende, aber keine notwendige Bedingung für CSR
- dies lässt sich auf OCSR verfeinern, wie folgt

■ Theorem: $\text{Gen}(2\text{PL}) \subset \text{OCSR}$ Ordnungserhaltender Konfliktserial. Schedule(?)

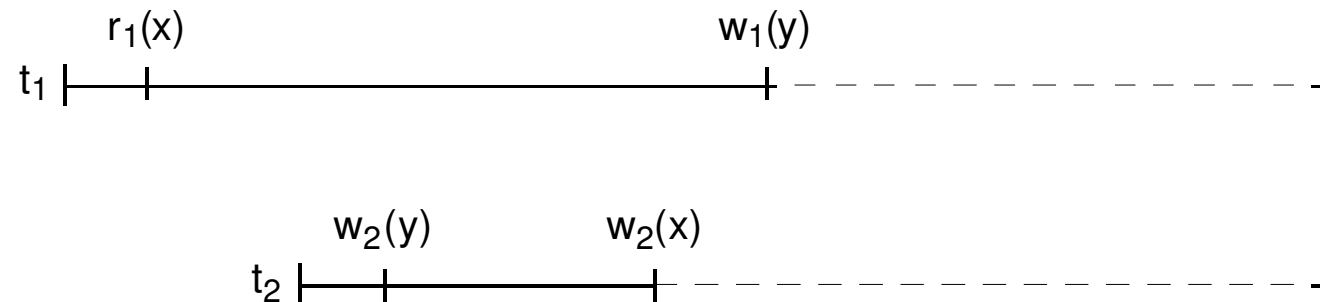
■ Beispiel

- $s = w_1(x) r_2(x) r_3(y) r_2(z) w_1(y) c_3 c_1 c_2 \in \text{CSR}$
- s fällt in die Klasse OCSR, aber nicht in $\text{Gen}(2\text{PL})$, (da es in s kein Paar von strikt sequentiellen TA gibt, ist OCSR-Bedingung automatisch erfüllt)

Sperrprotokolle – Deadlocks (1)

■ Deadlocks

- werden verursacht durch zyklisches Warten auf Sperren
- beispielsweise in Zusammenhang mit Sperrkonversionen (beispielsweise bezeichnet man das spätere Anheben (Upgrade) des Sperrmodus als Sperrkonversion)
- Beispiel



Sperrprotokolle – Deadlocks (2)

■ Deadlock-Erkennung

- Aufbau eines dynamischen Waits-for-Graph (WFG) mit aktiven TA 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)

Sperrprotokolle – Deadlocks (3)

■ Deadlock-Auflösung

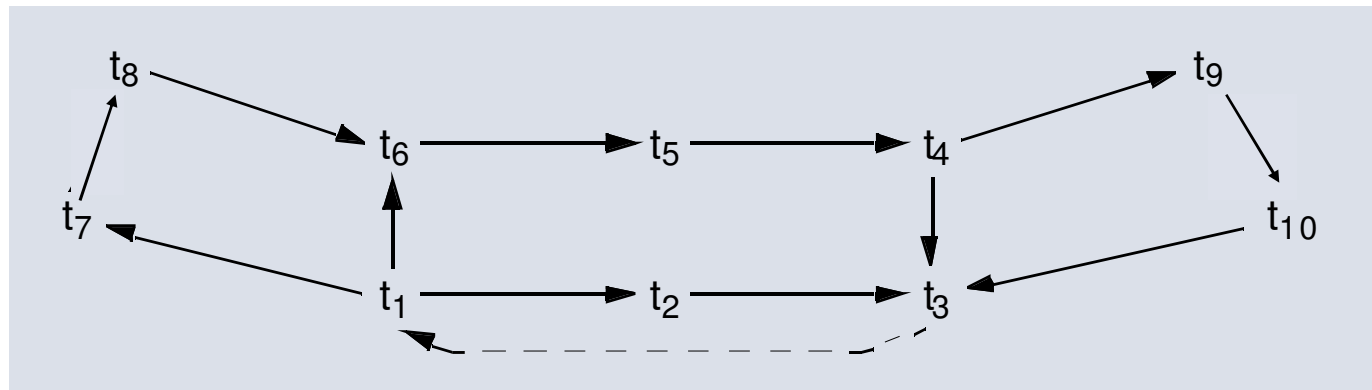
- Wähle eine TA in einem WFG-Zyklus aus
- Setze diese TA zurück
- Wiederhole diese Schritte, bis keine Zyklen mehr gefunden werden

■ Mögliche Strategien zur Bestimmung von „Opfern“

1. Zuletzt blockierte TA
2. Zufällige TA
3. Jüngste TA
4. Minimale Anzahl von Sperren
5. Minimale Arbeit (geringster Ressourcen-Verbrauch, z.B. CPU-Zeit)
6. Meiste Zyklen
7. Meiste Kanten

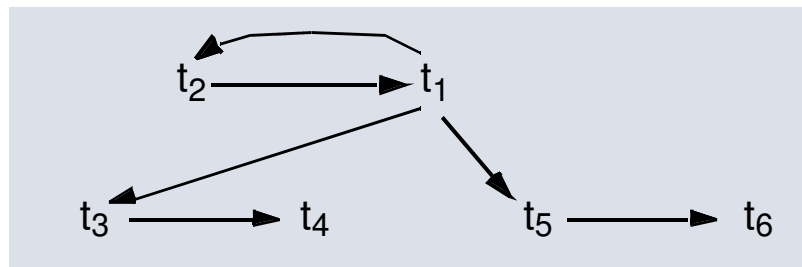
Sperrprotokolle – Deadlocks (4)

■ Beispiel



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

■ Beispiel



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

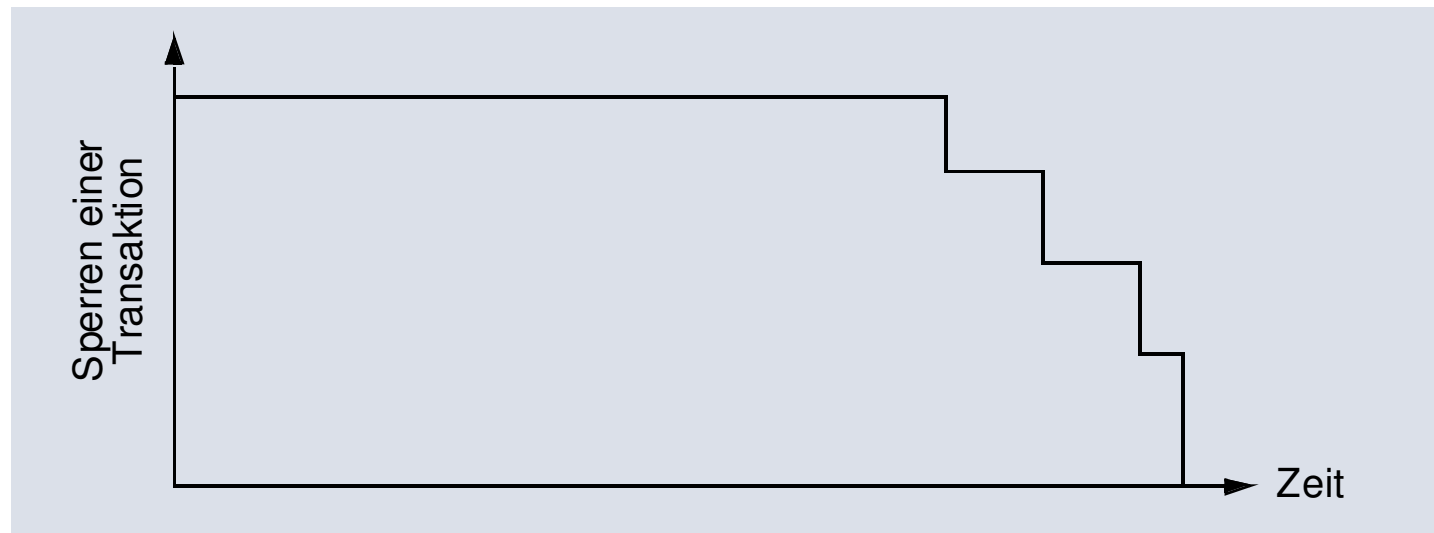
Sperrprotokolle – Deadlocks (5)

- Prinzip der Deadlock-Verhütung
 - Blockierungen (lock waits) werden eingeschränkt, so dass stets ein azyklischer WFG gewährleistet werden kann
- Strategien zur Deadlock-Verhütung (t_i fordert jeweils Sperre an)
 - Wait-Die: sobald t_i und t_j in Konflikt geraten: wenn t_i vor t_j gestartet ist (älter ist), dann wait(t_i), sonst restart(t_i)
(TA kann nur von jüngeren blockiert werden)
 - Wound-Wait: sobald t_i und t_j in Konflikt geraten: wenn t_i vor t_j gestartet wurde, 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)
 - Immediate Restart: sobald t_i und t_j in Konflikt geraten: restart(t_i)
 - Running Priority: sobald t_i und t_j in Konflikt geraten: wenn t_j selbst blockiert ist, dann restart(t_j) sonst wait(t_i)
 - Timeout: Wenn Timer ausläuft, wird t zurückgesetzt in der Annahme, dass t in einen Deadlock involviert ist!
 - **Konservative Ansätze**: TA, die zurückgesetzt wird, ist nicht notwendigerweise in einen Deadlock involviert

Sperrprotokolle – Preclaiming

■ Definition *Konservatives 2PL*

- Unter *statischem* oder *konservativem* 2PL (C2PL) fordert jede TA alle Sperren an, bevor sie den ersten Read- oder Write-Schritt ausführt (*Preclaiming*)



Nachteil:
unrealistisch
selten alle Fakten
apriori bekannt

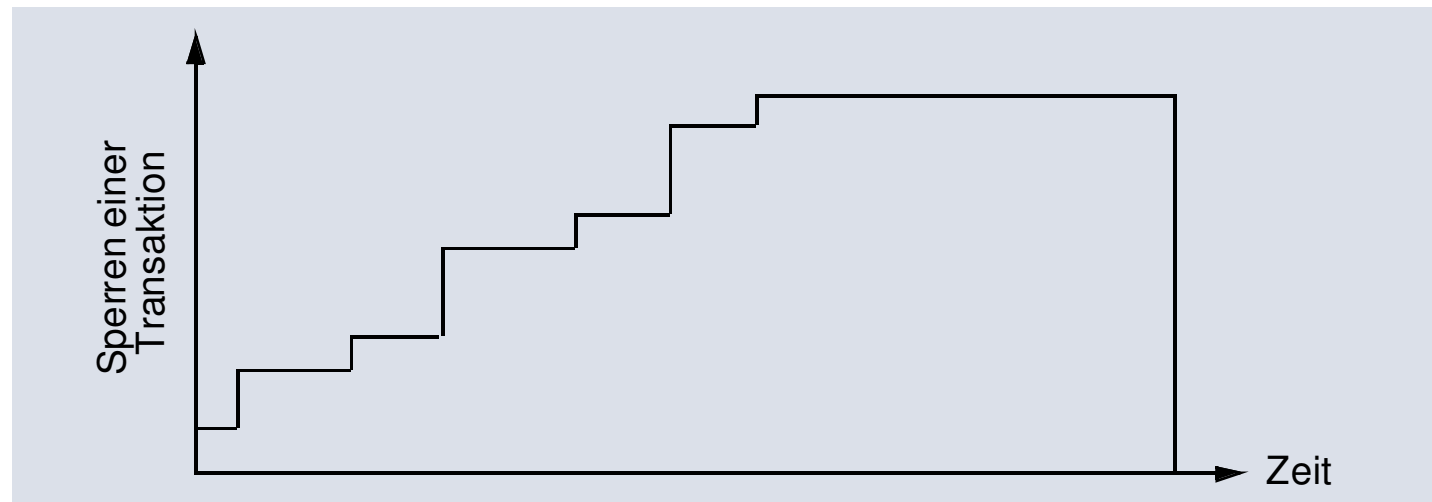
- C2PL vermeidet Deadlocks:
atomares Erwerben von Sperren
⇒ blockierte Transaktionen halten keine Sperren

Sperrprotokolle – S2PL

■ Definition *Striktes 2PL*

- unter striktem 2PL (S2PL) werden *alle exklusiven* Sperren (wl) einer TA bis zu ihrer Terminierung gehalten
- wird in praktischen Implementierungen am häufigsten eingesetzt

Vorteil:
verhindert cascadierende
"Abstürze"



- S2PL vermeidet *kaskadierende Abbrüche*

Sperrprotokolle – SS2PL

- Definition *Starkes 2PL*
 - unter starkem 2PL (strong 2PL, SS2PL) werden *alle Sperren* einer TA (wl, rl) bis zu ihrer *Terminierung* gehalten
- Theorem: $\text{Gen}(\text{SS2PL}) \subset \text{Gen}(\text{S2PL}) \subset \text{Gen}(\text{2PL})$
- Theorem: $\text{Gen}(\text{SS2PL}) \subset \text{COCSR}$
 - Erinnerung: Eine Historie bewahrt die *Commit-Reihenfolge* gdw die Reihenfolge der Commits der einer Serialisierungsreihenfolge entspricht
 - wird im Kontext verteilter Systeme ausgenutzt

mit SS2PL ist es mit mehreren DBMS möglich auch global (!) zu serialisieren

Timestamp Ordering (1)

- Diskussion von einigen nicht-sperrenden Protokollen
 - Sie garantieren die Sicherheit ihrer Ausgabe-Schedules ohne die Nutzung von Sperren
 - Einsatz vor allem in hybriden Protokollen
- Timestamp Ordering
 - 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:
$$p_i(x) <_s q_j(x) \text{ gdw } ts(t_i) < ts(t_j)$$
- Theorem: $\text{Gen(TO)} \subseteq \text{CSR}$

Timestamp Ordering (2)

■ Wer zu spät kommt, ...

- Operation $p_i(x)$ ist zu spät, wenn sie ankommt, nachdem der Scheduler schon die Konfliktoperation $q_j(x)$ ausgegeben hat und $i \neq j$, $ts(t_j) > ts(t_i)$ gilt
- TO-Regel muss vom Scheduler erzwungen werden: Wenn $p_i(x)$ zu spät kommt, ist restart (t_i) erforderlich

■ BTO-Protokoll (*Basic Timestamp Ordering*)

- BTO-Scheduler hat zwei Zeitstempel für jedes Datenelement zu halten
 - $\max-r(x) = \max\{ ts(t_j) \mid r_j(x) \text{ wurde ausgegeben} \} ; j = 1 \dots n$
 - $\max-w(x) = \max\{ ts(t_j) \mid w_j(x) \text{ wurde ausgegeben} \} ; j = 1 \dots n$
- Operation $p_i(x)$ wird mit $\max-q(x)$ für jedes in Konflikt stehende q verglichen
 - Wenn $ts(t_i) < \max-q(x)$, dann weise $p_i(x)$ zurück ($\text{abort}(t_i)$)
 - Sonst gebe $p_i(x)$ aus und setze $\max-p(x)$ auf $ts(t_i)$, wenn $ts(t_i) > \max-p(x)$

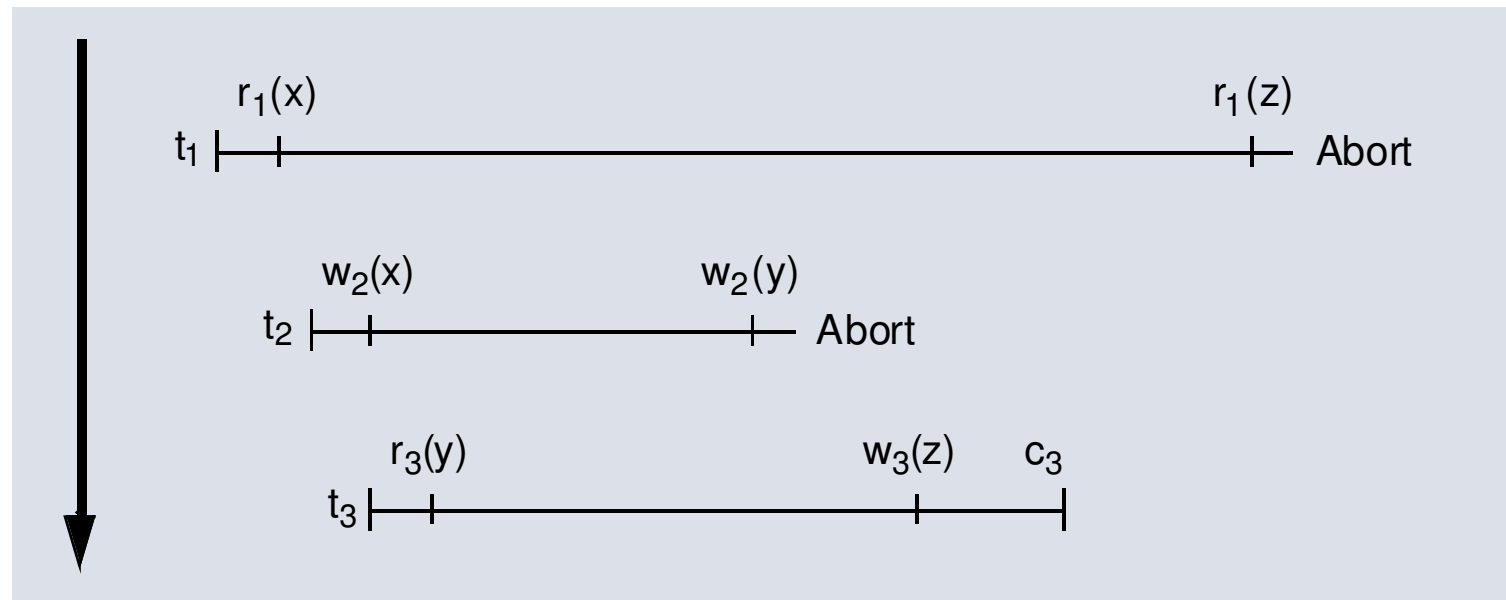
Timestamp Ordering (3)

■ BTO-Scheduler

- muss sicherstellen, dass DM seine Ausgaben in Scheduler-Reihenfolge verarbeitet (sonst potenziell Verletzung der zentralen Regel)
- führt deshalb „Handshake“ mit DM nach jeder Operation durch

■ Beispiel

- $s = r_1(x) \ w_2(x) \ r_3(y) \ w_2(y) \ c_2 \ w_3(z) \ c_3 \ r_1(z) \ c_1$



- $r_1(x) \ w_2(x) \ r_3(y) \ a_2 \ w_3(z) \ c_3 \ a_1$

Timestamp Ordering (4)

■ Beobachtung

- Wenn ein BTO-Scheduler neue Operationen in einer Reihenfolge empfängt, die stark von der Zeitstempelreihenfolge abweicht, sind möglicherweise viele TA zurückzusetzen
- konservative Variante durch künstliches Blockieren: $o_i(x)$ mit „hohem“ Zeitstempelwert wird eine zeitlang zurückgehalten, bis hoffentlich alle in Konflikt stehenden Operationen „pünktlich“ eingetroffen sind

Serialization Graph Testing (1)

- Erinnerung: CSR wird erreicht, wenn der Konfliktgraph G stets azyklisch gehalten wird
- SGT-Protokoll: für jede empfangene Operation $p_i(x)$
 1. Erzeuge einen neuen Knoten für TA t_i im Graph, wenn $p_i(x)$ die erste Operation von t_i ist
 2. Füge Kanten (t_j, t_i) ein für jedes $q_j(x) <_s p_i(x)$, das in Konflikt mit $p_i(x)$ ($i \neq j$) steht
 3. Wenn der Graph zyklisch geworden ist, setze t_i zurück (und entferne sie aus dem Graph), sonst gebe $p_i(x)$ zur Verarbeitung aus
- Theorem: $\text{Gen}(\text{SGT}) = \text{CSR}$

Serialization Graph Testing (2)

■ Löschen von Knoten

kompliziert, deshalb trotzdem
noch Sperrprotokolle
Hier müssten Lese- +
Schreib-zugriffe auf Objekte
gespeichert werden

-> Konflikte können
über commits hinaus
gehen (eingehende
Kanten)

- Löschregel: Ein Knoten t_i im Graph G kann gelöscht werden, wenn t_i terminiert ist und er ein Quellknoten ist (d.h., er hat **keine Eingangskanten**)
- Vorzeitiges Löschen von Knoten würde die Zyklenerkennung unmöglich machen
- Halten von Read- und Write-Sets von bereits abgeschlossenen TA erforderlich
- Deshalb ist SGT ungeeignet für praktische Implementierungen!

Vorteil: mehr parallel, aber aufwändiger

Optimistische Verfahren (1)

■ Motivation

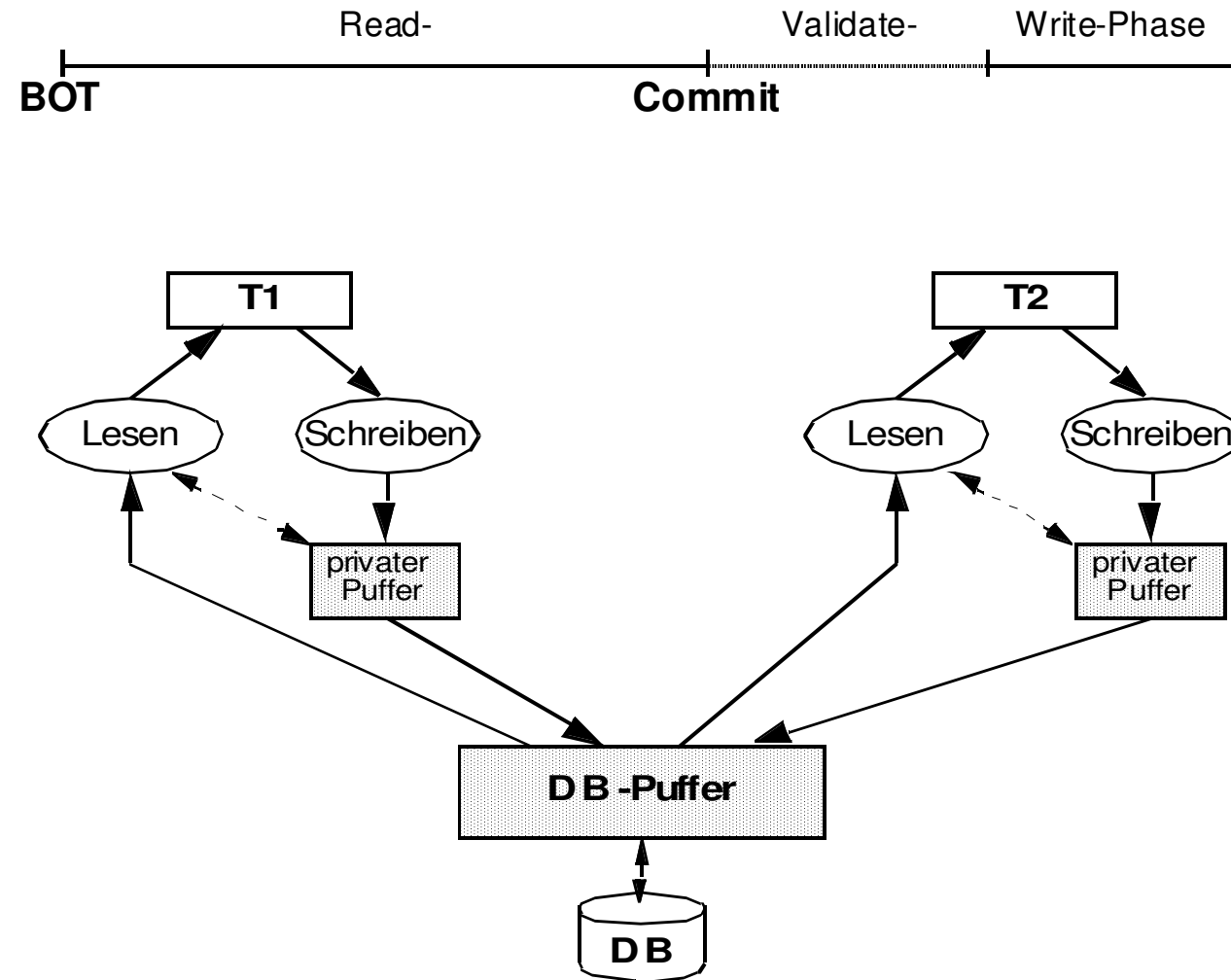
- In manchen Anwendungen benötigt man fast nur Lesezugriff
- Konflikte sind selten
- 2PL-Aufwand erscheint deshalb unangemessen hoch

■ 3 Phasen einer TA

- *Read-Phase*: Führe TA aus, kapsle dabei Write-Operationen in einem privaten Workspace
- *Validate-Phase (Certifier)*: Wenn t_i Commit ausführt, teste mit Hilfe von Read-Sets RS und Write-Sets WS, ob der Schedule in CSR bleibt, wenn t_i abgeschlossen wird
- *Write-Phase*: Nach erfolgreicher Validierung wird der (geänderte) Workspace-Inhalt in die DB (DB-Puffer) eingebracht (deferred writes), sonst wird t_i zurückgesetzt (Workspace wird aufgegeben)

Optimistische Verfahren (2)

■ Illustration

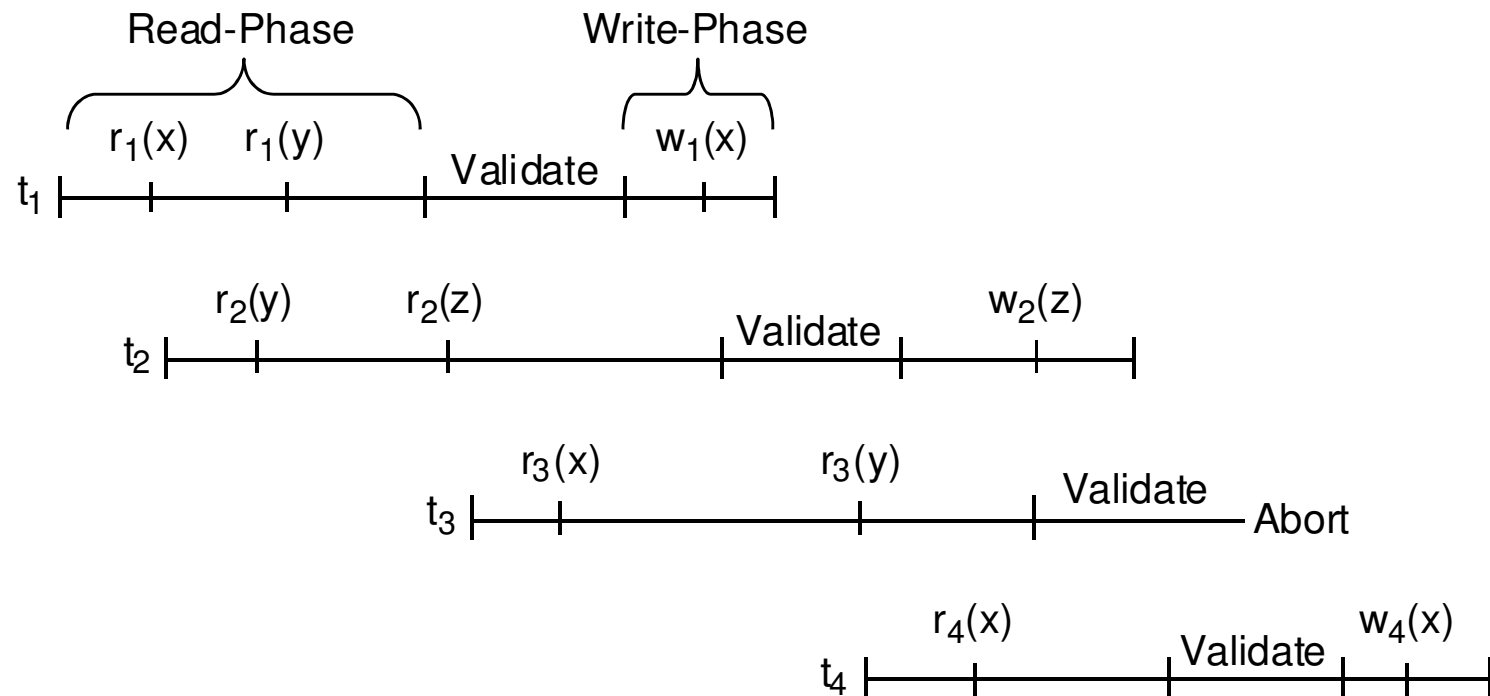


Optimistische Verfahren (4)

- Backward-Oriented Optimistic CC
 - TA-Validierung und -Schreibphase wird als *kritischer Abschnitt* ausgeführt: keine andere t_k kann ihre *val-write-Phase* beginnen
 - BOCC-Validierung von t_j : Vergleiche t_j mit allen vorher abgeschlossenen t_i . Akzeptiere t_j nur, wenn eine der beiden Bedingungen gilt:
 - t_i war abgeschlossen, bevor t_j gestartet wurde
 - $RS(t_j) \cap WS(t_i) = \emptyset$ und t_i wurde vor t_j validiert
- Lemma
 - Sei G ein DAG. Wenn ein neuer Knoten K in G derart hinzugefügt wird, dass keine Kanten von K ausgehen, dann ist der resultierende Graph immer noch ein DAG.
- Theorem: $\text{Gen(BOCC)} \subseteq \text{CSR}$

Optimistische Verfahren (3)

■ BOCC-Beispiel



Optimistische Verfahren (4)

- Forward-Oriented Optimistic CC
 - TA-Validierung wird als *starker kritischer Abschnitt* ausgeführt: während t_i in ihrer val-write-Phase ist, kann keine andere t_k einen Schritt ausführen
 - FOCC-Validierung von t_j : Vergleiche t_j mit allen aktiven t_i (die sich in ihrer Lesephase befinden müssen). Akzeptiere t_j nur, wenn $WS(t_j) \cap RS^*(t_i) = \emptyset$, wobei $RS^*(t_i)$ der momentane Read-Set von t_i ist
- Theorem: $\text{Gen(FOCC)} \subseteq \text{CSR}$
- FOCC garantiert sogar COCSR

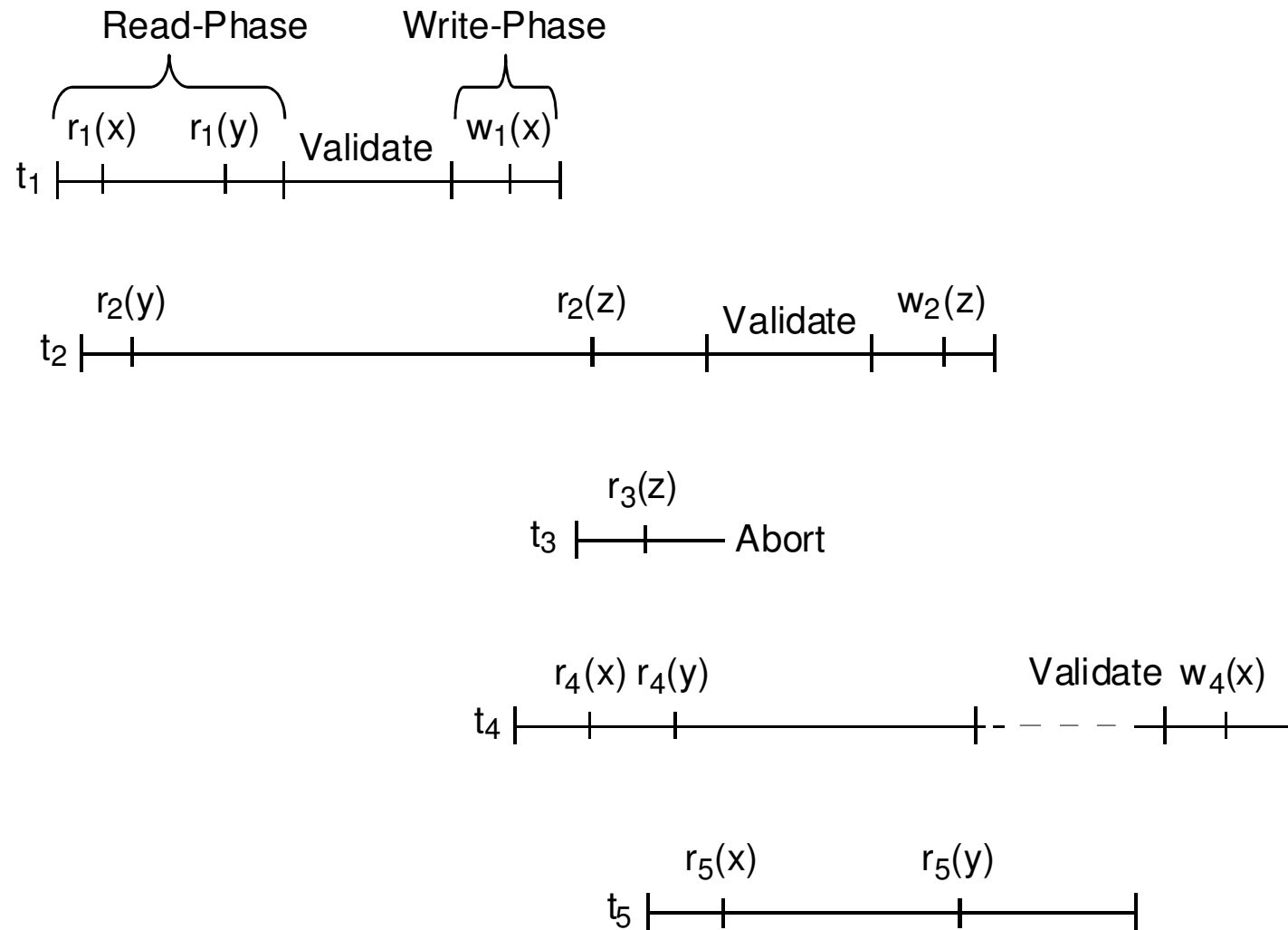
Optimistische Verfahren (5)

■ Bemerkungen

- FOCC ist viel flexibler als BOCC: Bei nicht erfolgreicher Validierung von t_j gibt es 3 Optionen:
 - setze t_j zurück
 - setze eine (oder mehrere) von den aktiven t_i zurück, für die $RS^*(t_i)$ mit $WS(t_j)$ überlappt
 - warte und wiederhole die Validierung von t_j später
- Read-only-TA brauchen überhaupt nicht zu validieren

Optimistische Verfahren (6)

■ FOCC-Beispiel



Zusammenfassung

- Wichtigstes Korrektheitskriterium der Synchronisation: Konfliktserialisierbarkeit
- Realisierung der Synchronisation durch Sperrverfahren
 - Sperren stellen während des laufenden Betriebs sicher, dass der resultierende Schedule serialisierbar bleibt.
 - Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt.
 - S2PL ist das flexibelste und robusteste Protokoll und wird am häufigsten in der Praxis eingesetzt.
 - Sperrverfahren sind pessimistisch und universell einsetzbar.
- SGT ist leistungsfähiger, verursacht aber mehr Aufwand.
- FOCC kann für spezifische Arbeitslasten attraktiv sein.
- Hybride Protokolle sind möglich, aber sie sind nicht-trivial.

Sperrprotokolle – Ordered Sharing (1)

■ Ordered Sharing

- Ziel

- Generalisierung von 2PL mit weniger Restriktionen als 2PL
- Generierung einer größeren Teilklasse von CSR

- Erinnerung

- $s = w_1(x) \ r_2(x) \ r_3(y) \ c_3 \ r_2(z) \ c_2 \ w_1(y) \ c_1$
 - $s \notin \text{Gen}(2\text{PL})$ wegen Schreib-Lese-Konflikt auf x
 - aber: $s \in \text{OCSR}$
- Höhere Parallelität durch gleichzeitige (geordnete) Vergabe von in Konflikt stehenden Sperren

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Ordered Sharing (2)

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

■ Neue Kompatibilität

- Zwei in Konflikt stehende Sperren auf demselben Datenelement können gleichzeitig von verschiedenen TA gehalten werden, solange die Sperranforderungen und die entsprechenden Datenoperationen in derselben Reihenfolge ausgeführt werden

■ Notation: Ordered Sharing

- $pl_i(x) \rightarrow ql_j(x)$ impliziert $pl_i(x) <_s ql_j(x)$ und $p_i(x) <_s q_j(x)$

■ Beispiel

- $s_1 = w_1(x) \ r_2(x) \ r_3(y) \ c_3 \ w_1(y) \ c_1 \ w_2(z) \ c_2$
- Scheduler-Ausgabe mit LT_2 :
 - $wl_1(x) \ w_1(x) \ rl_2(x) \ r_2(x) \ rl_3(y) \ r_3(y) \ ru_3(y) \ c_3 \ wl_1(y) \ w_1(y) \ wu_1(x) \ wu_1(y) \ c_1 \ wl_2(z) \ w_2(z) \ ru_2(x) \ wu_2(z) \ c_2$
- Sperrtabellen LT_5 , LT_7 oder LT_8 würden dieselbe Historie erlauben

Sperrprotokolle – Ordered Sharing (3)

■ Beispiel (Forts.)

LT_1	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	-	-

LT_2	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	\rightarrow
$wl_j(x)$	-	-

LT_3	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	\rightarrow	-

LT_4	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	-	\rightarrow

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Ordered Sharing (4)

■ Beispiel (Forts.)

LT_5	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	\rightarrow
$wl_j(x)$	\rightarrow	-

LT_6	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	\rightarrow	\rightarrow

LT_7	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	\rightarrow
$wl_j(x)$	-	\rightarrow

LT_8	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	\rightarrow
$wl_j(x)$	\rightarrow	\rightarrow

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Ordered Sharing (5)

■ Beispiel (Forts.)

- $s_2 = r_1(x) w_2(x) r_3(y) c_3 w_1(y) c_1 w_2(z) c_2$
LT₃, LT₅, LT₆ oder LT₈ würden s_2 erlauben
- $s_3 = w_1(x) w_2(x) r_3(y) c_3 w_1(y) c_1 w_2(z) c_2$
LT₄, LT₆, LT₇ oder LT₈ würden s_3 erlauben

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Ordered Sharing (6)

- Zusätzliche Sperregeln erforderlich

- OS1 (Erwerb von Sperren)

- Angenommen $pl_i(x) \rightarrow ql_j(x)$ ist erlaubt:
Wenn $pl_i(x) <_s ql_j(x)$, dann muss $p_i(x) <_s q_j(x)$ gelten
- Allein nicht ausreichend für CSR

- Gegenbeispiel

- $s = wl_1(x) w_1(x) wl_2(x) w_2(x) wl_2(y) w_2(y) wu_2(x) wu_2(y) c_2$
 $wl_1(y) w_1(y) wu_1(x) wu_1(y) c_1$
- s erfüllt OS1 und LR1 - LR4
- s ist zweiphasig
- aber $s \notin \text{CSR}$ wegen ‚unterschiedlich gerichteten‘ Schreib-Schreib-Konflikten auf x und y

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Ordered Sharing (7)

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

■ OS2 (Freigabe von Sperren)

- Wenn $pl_i(x) \rightarrow ql_j(x)$ und t_i noch keine Sperre freigegeben hat, ist t_j reihenfolgeabhängig (*order dependent*) von t_i . Wenn solch ein t_i existiert, dann ist t_j *on hold* (im Haltezustand). Während eine TA *on hold* ist, darf sie keine Sperren freigeben.

■ Neue Familie von Sperrprotokollen unter Nutzung von

- Sperrregeln LR1 - LR4
- Regeln OS1 und OS2
- Zweiphasigkeit
- eine der acht Kompatibilitätstabellen $LT_1 - LT_8$

■ Im Falle der Nutzung von LT_8 heißt resultierendes Protokoll O2PL

Sperrprotokolle – Ordered Sharing (8)

■ Beispiel

- O2PL-Scheduler erhält Eingabe
- $s = r_1(x) w_2(x) r_3(y) w_2(y) c_2 w_3(z) c_3 r_1(z) c_1$
- und erzeugt Ausgabe

Scheduler

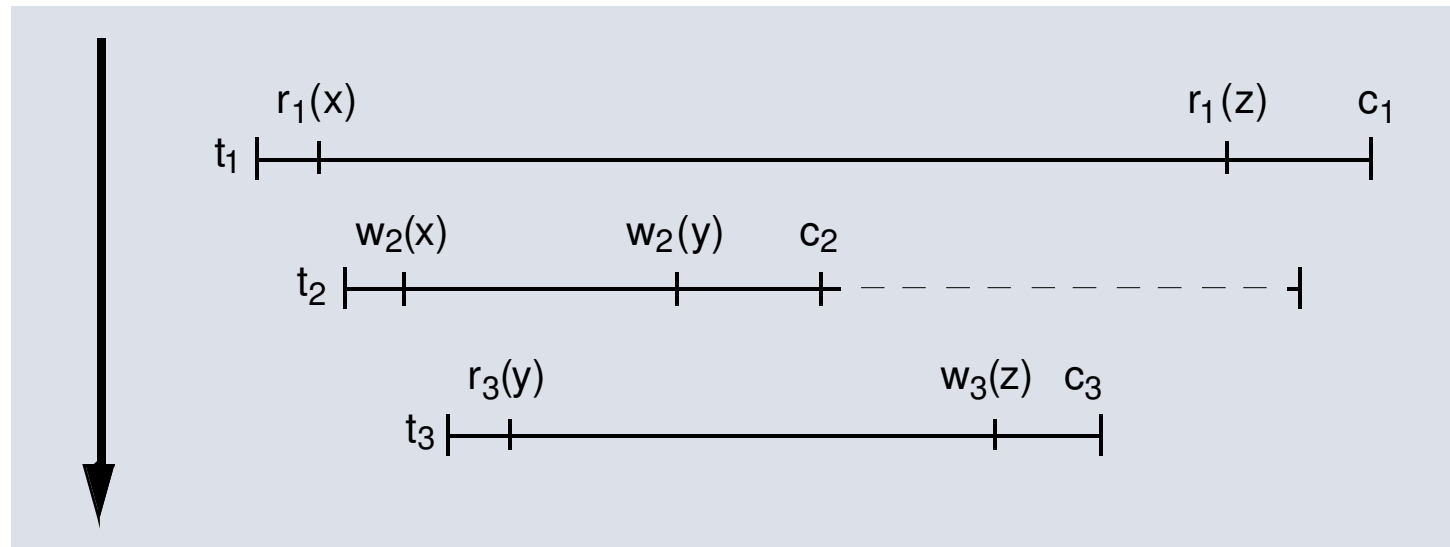
Klassifikation

Locking

Non-Locking

Optimistic

Hybrid



- $s' = rl_1(x) r_1(x) wl_2(x) w_2(x) rl_3(y) r_3(y) wl_2(y) w_2(y) wl_3(z) w_3(z) ru_3(y) wu_3(z) c_3 rl_1(z) r_1(z) ru_1(x) ru_1(z) wu_2(x) wu_2(y) c_2 c_1$

Sperrprotokolle – Ordered Sharing (9)

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

- Theorem (Sicherheit von Ordered Sharing):
 - $\text{Gen}(\text{LT}_i)$, $1 \leq i \leq 8$, bezeichne die Menge der Historien, die durch ein Sperrprotokoll bei Einsatz der Sperrtabelle LT_i erzeugt werden. Dann gilt $\text{Gen}(\text{LT}_i) \subseteq \text{CSR}$
- Theorem: $\text{Gen}(\text{O2PL}) \subseteq \text{OCSR}$
- Theorem (Mächtigkeit von O2PL): $\text{OCSR} \subseteq \text{Gen}(\text{O2PL})$
- Korollar: $\text{Gen}(\text{O2PL}) = \text{OCSR}$
- O2PL ist theoretisch 2PL überlegen, aber es erfordert Laufzeitaufwand (Buchhaltung), wobei der Gewinn möglicherweise diesen Aufwand nicht lohnt.

Sperrprotokolle – Altruistische Sperren (1)

■ Ziel

- Erweiterung des 2PL-Protokolls, um bei langen TA
 - ernsthafte Leistungsprobleme zu vermeiden
 - die Parallelität auf gemeinsamen Daten zu erhöhen und
 - trotzdem automatisch Serialisierbarkeit zu gewährleisten
- Zusätzliche Information der langen TA hilft dem Scheduler, vorzeitig den Zugriff auf gesperrte Daten zu ermöglichen

■ Beispiel

- $D = \{a, b, c, d, e, f, g\}$
- t_1 greift auf a bis g in alphabetischer Reihenfolge zu
- Zugriffsanforderungen paralleler TA:
 - t_2 : a und b
 - t_3 : c und e
 - t_4 : f und g

Scheduler

Klassifikation

Locking

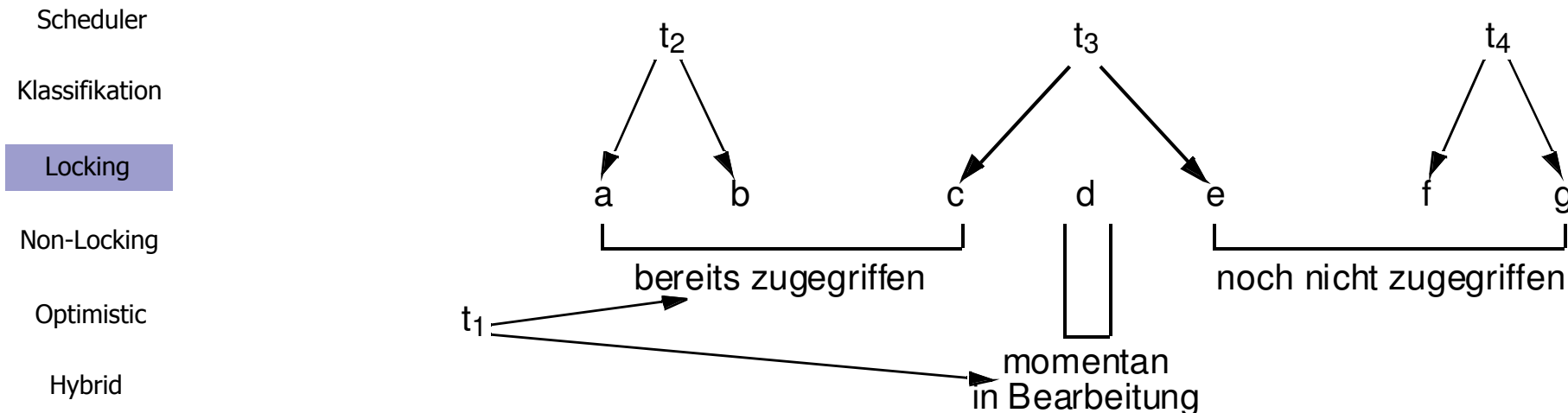
Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Altruistische Sperren (2)

- Beispiel (Forts.)
 - Momentaufnahme



- t₂ könnte vom Scheduler zugelassen werden, wenn
 - t₁ einen Hinweis gibt, dass a und b nicht mehr benötigt werden
 - das Zugriffsprofil von t₂ bekannt ist
- t₂ würde in der Serialisierungsreihenfolge nach t₁ erscheinen

Sperrprotokolle – Altruistische Sperren (3)

■ Idee des Altruistischen Sperrens (AL)

- AL benutzt neben *lock* und *unlock* eine dritte Zugriffskontrolloperation *donate*
- $d_i(x)$ bedeutet, dass t_i seine Sperre auf x anderen TAs ‚schenkt‘ (*donate*)
- die TA, die Sperren verschenkt, kann weiterhin Sperren erwerben, bleibt aber zweiphasig bezüglich der *unlock*-Operationen

■ Beispiel mit Verschenken von Sperren (*lock donation*):

- $wl_1(a) \ w_1(a) \ d_1(a) \ rl_2(a) \ r_2(a) \ wl_1(b) \ w_1(b) \ d_1(b) \ rl_2(b) \ r_2(b) \ wl_1(c) \ w_1(c) \ . \ . \ ru_2(a) \ ru_2(b) \ . \ . \ wu_1(a) \ wu_1(b) \ wu_1(c) \ . \ . \ .$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Altruistische Sperren (4)

■ AL-Regeln

- AL1: Sobald t_i eine Sperre auf x verschenkt hat, kann sie nicht mehr auf x zugreifen
- AL2: Nachdem t_i eine Sperre auf x verschenkt hat, muss t_i irgendwann auch ein *unlock* auf x durchführen
- AL3: t_i und t_j können nur dann gleichzeitig in Konflikt stehende Sperren auf x halten, wenn t_i seine Sperre auf x verschenkt hat
- AL4: Wenn TA t_j der TA t_i *verpflichtet* ist, muss sie vollständig *im Sog von* t_i bleiben, bis t_i mit der Freigabe von Sperren begonnen hat

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Altruistische Sperren (5)

■ Definition *Sog und Verpflichtung*

1. $p_j(x)$ von TA t_j *ist im Sog von (in the wake of)* TA t_i ($i \neq j$) in s , wenn $d_i(x) \in \text{op}(s)$ und $d_i(x) <_s p_j(x) <_s \text{ou}_i(x)$ für irgendeine Operation $o_i(x)$ von t_i gilt.
2. TA t_j ist im Sog von t_i , wenn irgendeine Operation von t_j im Sog von t_i ist. t_j ist *vollständig im Sog von (completely in the wake of)* t_i , wenn alle ihre Operationen im Sog von t_i sind.
3. TA t_j ist TA t_i *verpflichtet (indebted)* in einem Schedule s , wenn es $o_i(x)$, $d_i(x)$, $p_j(x) \in \text{op}(s)$ gibt, so dass $p_j(x)$ im Sog von t_i ist und entweder $o_i(x)$ und $p_j(x)$ in Konflikt sind oder es eine Operation $q_k(x)$ mit $d_i(x) <_s q_k(x) <_s p_j(x)$ gibt, die in Konflikt mit $o_i(x)$ und $p_j(x)$ ist.

- Der Begriff Verpflichtung führt eine Bedingung ein, die erfüllt sein muss, wenn t_j in den Sog von t_i kommt

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Altruistische Sperren (6)

■ AL-Protokoll befolgt

1. Sperregeln LR1 - LR4
2. Zweiphasigkeit
3. Schenkungsmöglichkeiten (*donations*) und
4. die Regeln AL1 - AL4

■ Beispiel

- $s = rl_1(a) \ r_1(a) \ d_1(a) \ wl_3(a) \ w_3(a) \ wu_3(a) \ c_3 \ rl_2(a) \ r_2(a) \ wl_2(b) \ ru_2(a) \ w_2(b) \ wu_2(b) \ c_2 \ rl_1(b) \ r_1(b) \ ru_1(a) \ ru_1(b) \ c_1$
- Indirekter Konflikt zwischen t_1 und t_2 über t_3 : $r_1(a) \ w_3(a) \ r_2(a)$
außerdem: $w_2(b) \ r_1(b) \Rightarrow s \notin \text{CSR}$
- durch AL4 verboten

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Sperrprotokolle – Altruistische Sperren (7)

■ Korrigiertes Beispiel mit AL1 - AL4

- $s = rl_1(a) r_1(a) d_1(a) wl_3(a) w_3(a) wu_3(a) c_3 rl_2(a) r_2(a) rl_1(b) r_1(b) ru_1(a) ru_1(b) c_1 wl_2(b) ru_2(a) w_2(b) wu_2(b) c_2$

- $r_1(a) w_3(a) r_2(a)$ und $r_1(b) w_2(b)$
- durch AL zugelassen
(t_2 teils im Sog von t_1 , teils nach Freigabe von Sperren durch t_1)

■ Theorem: $\text{Gen}(2\text{PL}) \subset \text{Gen}(\text{AL})$

■ Theorem: $\text{Gen}(\text{AL}) \subset \text{CSR}$

■ Beispiel

- $s = r_1(x) r_2(z) r_3(z) w_2(x) c_2 w_3(y) c_3 r_1(y) r_1(z) c_1$
- die Inklusion ist echt: $s \in \text{CSR}$, aber $s \notin \text{Gen}(\text{AL})$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

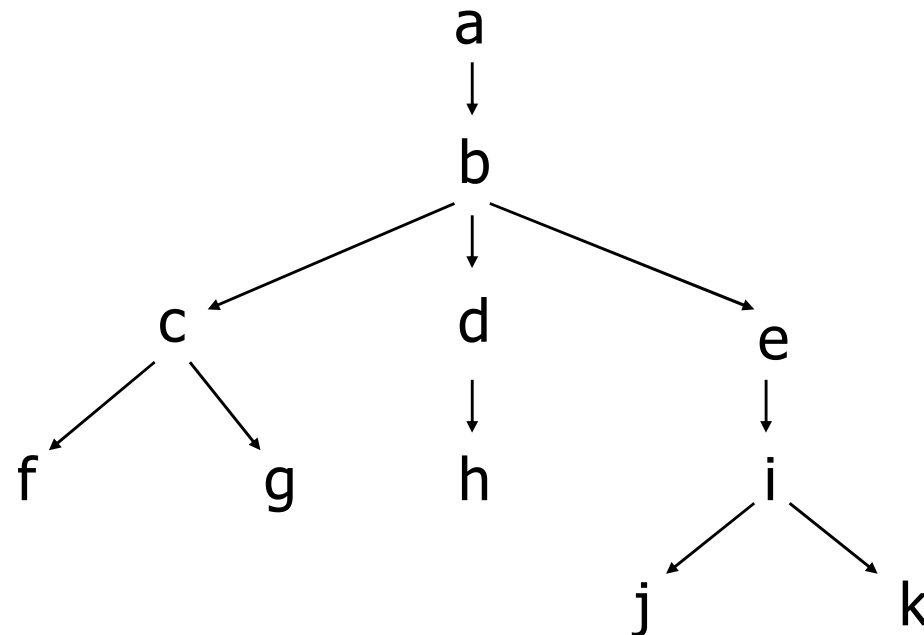
Nicht-2PL-Sperrprotokolle (1)

■ Achtung

- nur für sehr spezielle Einsatzbereiche
- *access patterns*

■ Write-Only-Tree-Locking (WTL)

- TAs führen ausschließlich Writes aus
- Baum-Organisation von Datenobjekten (*access pattern*)
- Beispiel



Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Nicht-2PL-Sperrprotokolle (2)

■ WTL (Forts.)

- Durch *access-pattern* vorgegebene Zugriffsreihenfolge ersetzt die 2-Phasen-Eigenschaft
- Protokol: LR1 – LR4 und (zusätzlich)
 - WTL1: für jeden Knoten x des Baumes gilt, dass $wl_i(x)$ nur dann gesetzt werden kann, wenn t_i bereits eine Schreibsperre auf dem Vaterknoten von x hält
 - WTL2: nach einem $wu_i(x)$ ist (auf demselben Datenobjekt x) kein weiteres $wl_i(x)$ erlaubt
- Beispiel (siehe Baum auf vorhergehender Folie)
 - $t = w(d) w(i) w(k)$
 - $wl(a) wl(b) wu(a) wl(d) wl(e) wu(b) \underline{w(d)} wu(d) wl(i) wu(e) \underline{w(i)} wl(k) wu(i) \underline{w(k)} wu(k)$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Nicht-2PL-Sperrprotokolle (3)

■ WTL (Forts.)

- Lemma

- Sperrt t_i x vor t_j , dann gilt für jeden Nachfolger (im Baum) von x , der von t_i und t_j gesperrt wird, dass er von t_i vor t_j gesperrt wird

- Theorem: $\text{Gen(WTL)} \subseteq \text{CSR}$

- Theorem: WTL ist Deadlock-frei

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Nicht-2PL-Sperrprotokolle (4)

■ Read/Write Tree Locking (RWTL)

- Betrachte TAs mit Lese-Operationen

- $t_1 = r_1(a) r_1(b) w_1(a) w_1(b) r_1(e) r_1(i) c_1$

- $t_2 = r_2(a) r_2(b) r_2(e) r_2(i) w_2(i) c_2$

- Anwendung von WTL mit zusätzlichen *shared read locks* könnte zu folgendem Schedule führen

- $rl_1(a) rl_1(b) r_1(a) r_1(b) wl_1(a) w_1(a) wl_1(b) ul_1(a) rl_2(a) r_2(a)$
 $w_1(b) rl_1(e) ul_1(b) rl_2(b) r_2(b) ul_2(a) rl_2(e) rl_2(i) ul_2(b) r_2(e)$
 $r_1(e) r_2(i) wl_2(i) w_2(i) wl_2(k) ul_2(e) ul_2(i) rl_1(i) ul_1(e) r_1(i) \dots$

- der nicht serialisierbar ist; betrachte beispielsweise Konflikte $(w_1(a), r_2(a))$ und $(w_2(i), r_1(i))$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Nicht-2PL-Sperrprotokolle (5)

■ RWTL (Forts.)

- Betrachte zu TA t *read set* $RS(t)$ und *write set* $WS(t)$
- $RS(t)$ bildet Komponenten C_i von jeweils verbundenen Datenobjekten im Baum
- Ein *pitfall* von t ist eine Menge der Form $C_i \cup \{x \in WS(t) \mid x \text{ ist Sohn oder Vater eines } y \in C_i\}$, $1 \leq i \leq m$
- Beispiel
 - Betrachte t mit $RS(t) = \{f, i, g\}$ und $WS(t) = \{c, l, j, k, o\}$
 - $C_1 = \{f, i\}$, $C_2 = \{g\}$
 - *pitfalls*: $pf_1 = \{c, f, i, l, j\}$, $pf_2 = \{g, c, k\}$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Nicht-2PL-Sperrprotokolle (6)

- RWTL (Forts.)
 - Beispiel (Forts.)

Scheduler

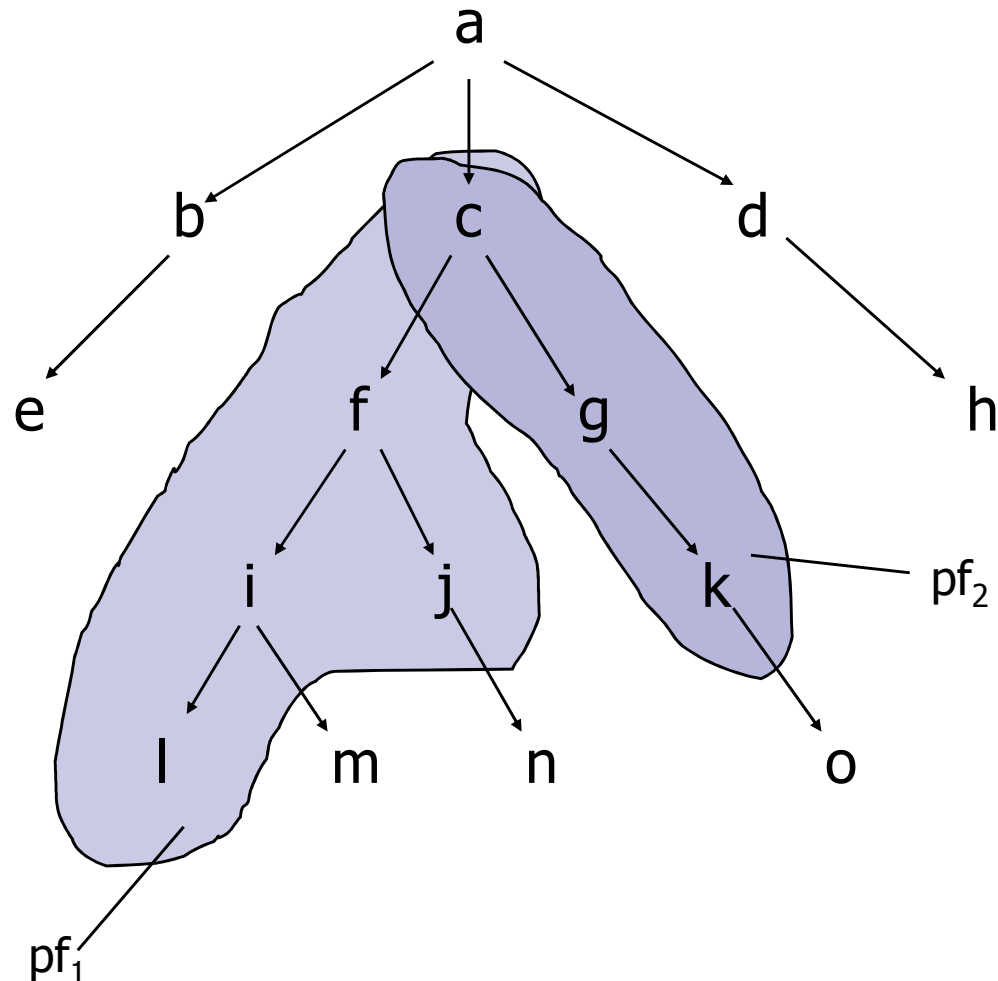
Klassifikation

Locking

Non-Locking

Optimistic

Hybrid



Nicht-2PL-Sperrprotokolle (7)

■ RWTL (Forts.)

- Protokoll: WTL und (zusätzlich)
 - RWTL1: die Transaktion erfüllt die 2-Phasen-Eigenschaft in jedem ihrer *pitfalls*
- Theorem: $\text{Gen}(\text{RWTL}) \subseteq \text{CSR}$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Hybride Protokolle (1)

■ Idee der hybriden Protokolle

- Zerlegung des CC-Problems in folgende Konflikttypen
 1. rw- (und wr-) Synchronisation
 2. ww-Synchronisation
- Aufteilung in verschiedene Datenpartitionen
- Die Kombination muss garantieren, dass die *Vereinigung* der lokalen Konfliktgraphen ($G_{rw}(s)$ und $G_{ww}(s)$) azyklisch ist

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Hybride Protokolle (2)

■ Beispiel

- SS2PL für rw/wr-Synchronisation und TO für ww-Synchronisation mit TWR (Thomas' Write Rule)

- TWR

- für $w_j(x)$: wenn $ts(t_j) > \max-w(x)$, dann führe $w_j(x)$ aus, sonst tue nichts (d.h., ignoriere $w_j(x)$)

- $s_1 = w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1$

- $s_2 = w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ r_1(y) \ w_1(y) \ c_1$

- Beide werden von SS2PL/TWR akzeptiert mit $ts(t_1) < ts(t_2)$, aber $s_2 \notin \text{CSR}$

- Problem mit s_2 : Synchronisation zwischen zwei lokalen Serialisierungsreihenfolgen erforderlich

- Lösung: Weise Zeitstempel zu, so dass die Serialisierungsreihenfolgen von SS2PL und TWR korrespondieren: $ts(i) < ts(j) \Leftrightarrow c_i < c_j$

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid

Hybride Protokolle (3)

■ Hybride Protokolle für Datenpartitionen

- Partitioniere D in disjunkte Teilmengen D_1, \dots, D_n
 - mit $n \geq 2$, $D_i \cap D_j = \emptyset$ für $i \neq j$ und
 - $\bigcup_{i=1}^n D_i = D$
- Verschiedene CC-Protokolle für die n Partitionen
- Vereinigung der Konfliktgraphen muss stets zyklensfrei sein

Scheduler

Klassifikation

Locking

Non-Locking

Optimistic

Hybrid