DATENBANKANWENDUNG

Wintersemester 2013/2014

Holger Schwarz
Universität Stuttgart, IPVS
holger.schwarz@ipvs.uni-stuttgart.de

Beginn:	**23.10.2013**
Mittwochs:	**11.45 – 15.15 Uhr, Raum 46-268 (Pause 13.00 – 13.30)**
Donnerstags:	**10.00 – 11.30 Uhr, Raum 46-268**
	11.45 – 13.15 Uhr, Raum 46-260

http://wwwlgis.informatik.uni-kl.de/cms/courses/datenbankanwendung/

7. Synchronisation - Algorithmen[1]

- **Entwurf des Schedulers**
- **Klassifikation von Synchronisationsalgorithmen**
- **Sperrprotokolle**
  - Sperrregeln
  - Zweiphasige Sperrprotokolle (2PL, C2PL, S2PL und SS2PL)
  - Deadlocks und ihr Behandlung
- **Nicht-sperrende Protokolle**
  - Zeitstempel-Verfahren
  - Serialization Graph Testing (SGT)
- **Optimistische Synchronisation (OCC)**
  - Eigenschaften
  - BOCC, FOCC

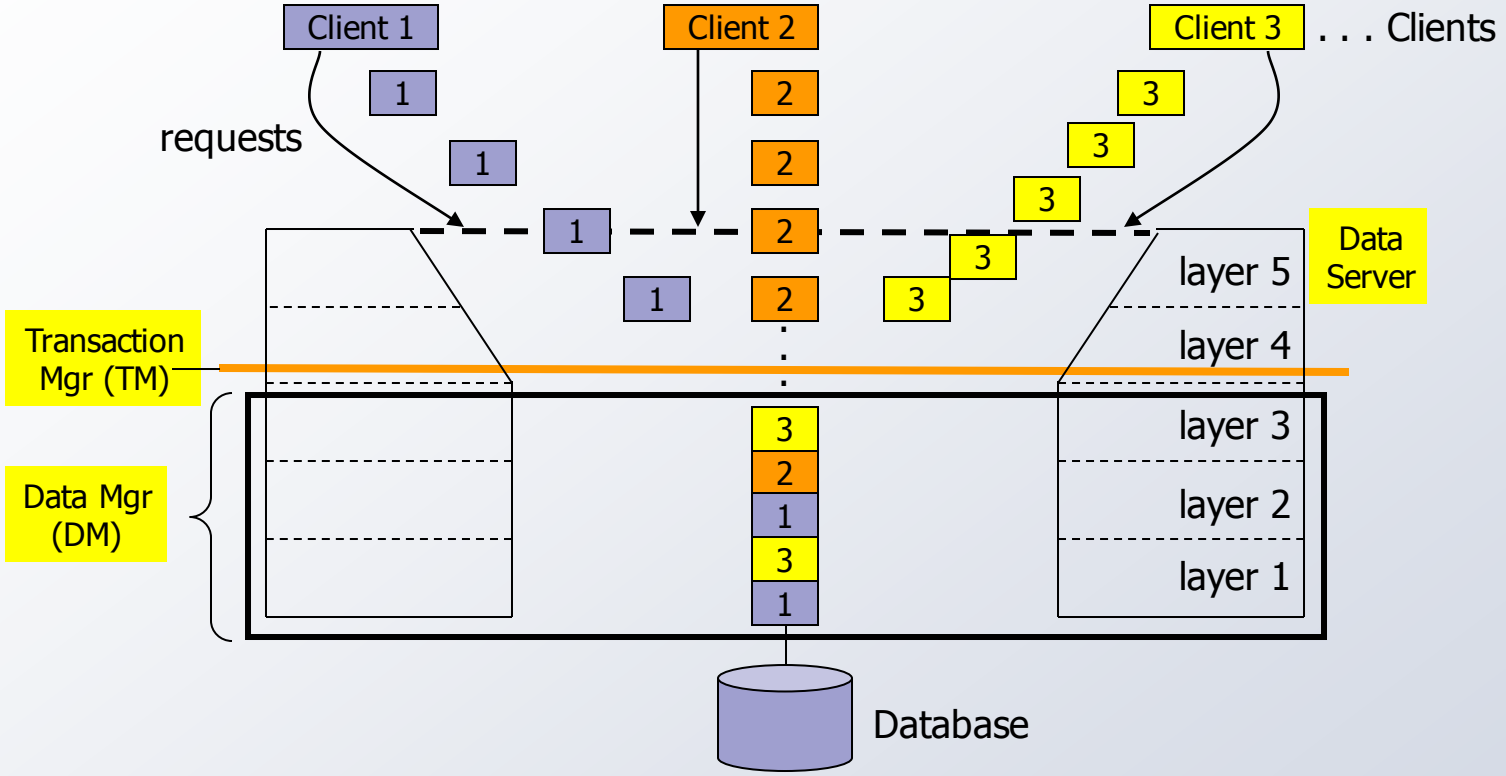
1. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, Addison-Wesley Publ. Comp., 1987 (http://research.microsoft.com/pubs/ccontrol/)

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 generieren kann. S heißt CSR-sicher, wenn Gen(s) $\subseteq$ CSR

Scheduler – Allgemeiner Entwurf



- **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

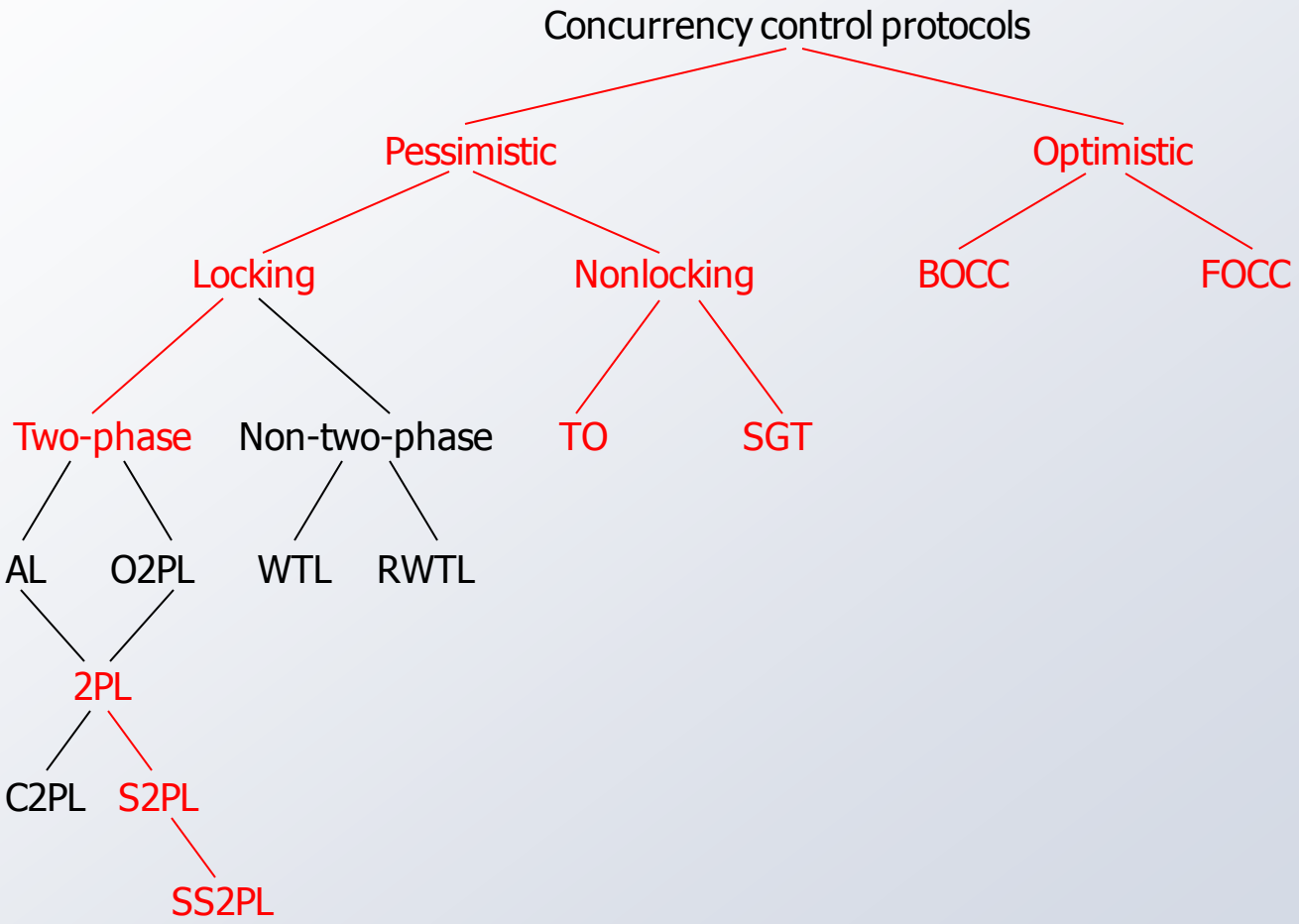
Generischer Scheduler


```
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 };
```

- **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

Klassifikation von Protokollen

- **Klassifikationsschema**



Klassifikation von Protokollen (2)

- **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

- **Allgemeine Idee**
  - 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**
  - 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
- **Kompatibilität und neuer Modus**

angeforderte Sperre	aktueller Modus des Objekts x: NL	R	X
rl(x)	+	+	-
wl(x)	+	-	-

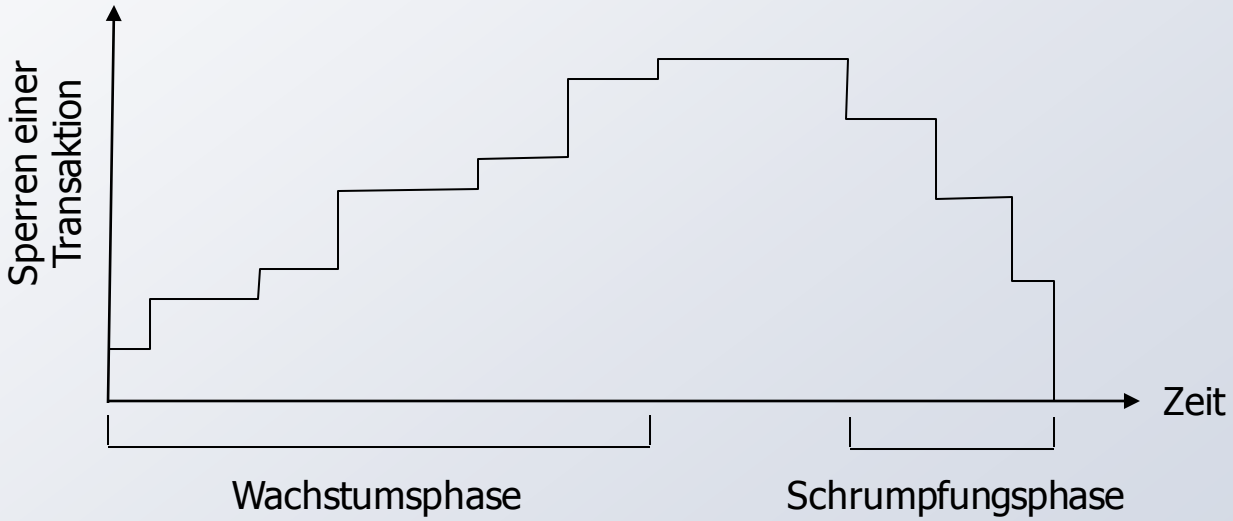
angeforderte Sperre	neuer Modus des Objekts x: NL	R	X
rl(x)	R	R	-
wl(x)	X	-	-

Sperrregeln

- **Allgemeine Sperrregeln (locking well-formedness rules)**
  - LR1: Jeder Datenoperation $r_i(x)$ [$w_i(x)$] muss ein $rl_i(x)$ [$wl_i(x)$] 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

2PL

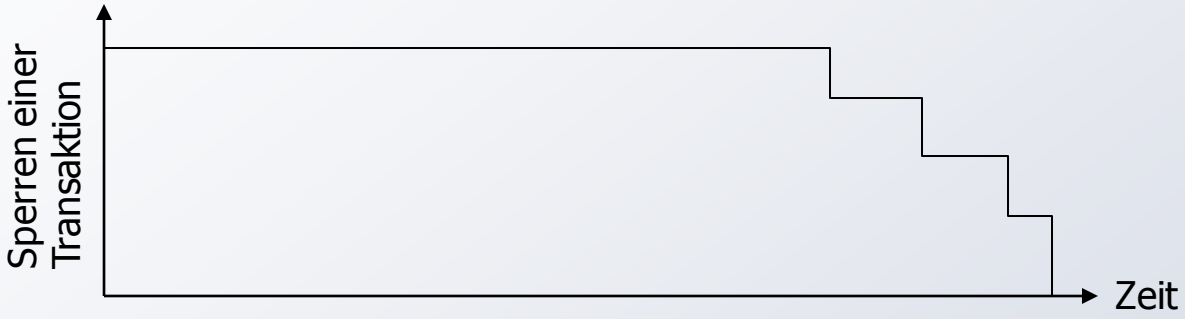
- **Definition: 2PL**
  - Ein Sperrprotokoll ist zweiphasig (2PL), wenn für jeden (Ausgabe-)Schedule s und jede TA $t_i \in trans(s)$ kein ql_i-Schritt dem ersten ou_i-Schritt folgt ($o, q \in \{r, w\}$)
- **Ausgabe eines 2PL-Schedulers**



Konservatives 2PL (C2PL)

- **Definition: Konservatives 2PL**

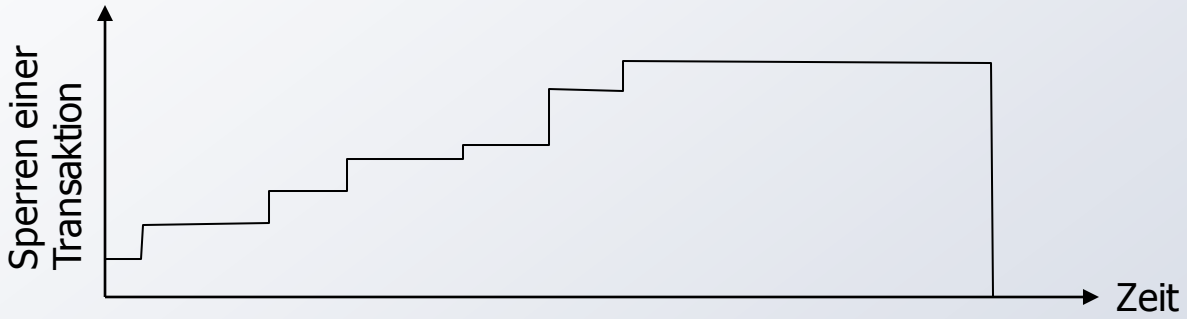
  Unter statischem oder konservativem 2PL (C2PL) fordert jede TA alle Sperren an, bevor sie den ersten Read- oder Write-Schritt ausführt (Preclaiming)



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

- **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



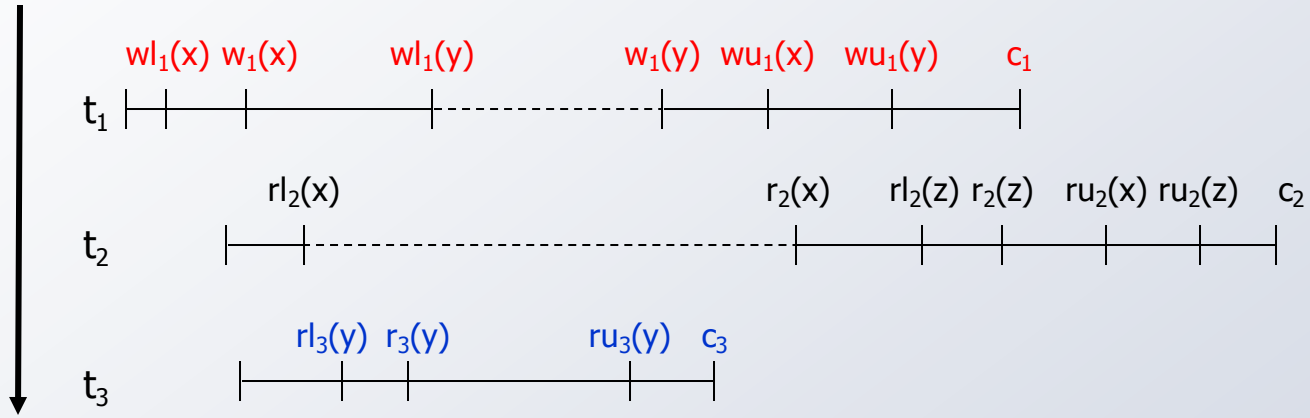
- **Definition Starkes 2PL**
  Unter starkem 2PL (strong 2PL, SS2PL) werden alle Sperren einer TA (wl, rl) bis zu ihrer Terminierung gehalten

Sperrprotokolle – 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**



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

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

Sperrprotokolle – 2PL (2)

- **Theorem**
  Ein 2PL-Scheduler ist CSR-sicher, d.h., Gen(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(2PL)
  - Forderung von 2PL nicht erfüllt:
    s_2 kann nicht in Gen(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 $r_3(y) < ru_3(y)$ (Wohlgeformtheitsregeln)
  - $r_2(x) < r_3(y)$ (aus dem Schedule)
  - c_2 vor c_3 würde $wu_1(x) < wl_1(y)$ implizieren, was der 2PL-Eigenschaft (Zweiphasigkeit) widerspricht
  - s_2 wird durch 2PL-Scheduler transformiert zu

Warten

Gewährung

$wl_1(x)\ \ w_1(x)\ \ rl_2(x)\ \ rl_3(y)\ \ r_3(y)\ \ ru_3(y)\ \ c_3\ \ wl_1(y)\ \ w_1(y)\ \ wu_1(x)\ \ rl_2(x)\ \ r_2(x)$

$wu_1(y)\ \ c_1\ \ ru_2(x)\ \ c_2$

  - also zu

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

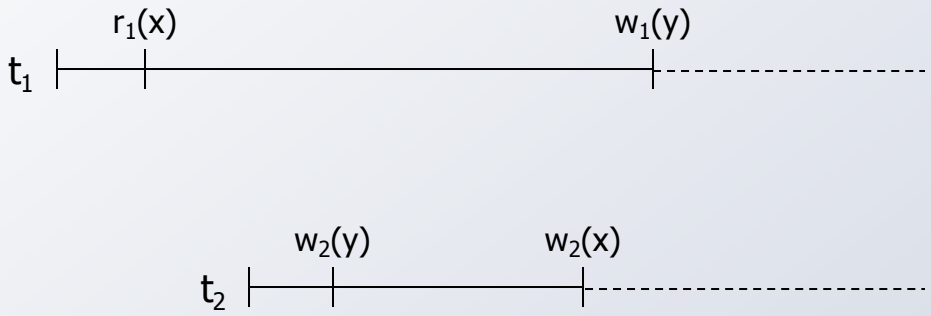
Sperrprotokolle – 2PL (3)

- **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, wie folgt
- **Theorem: Gen(2PL) $\subset$ OCSR**
- **Beispiel (siehe 7 - 13)**
  - $s_1 = w_1(x)\ r_2(x)\ r_3(y)\ r_2(z)\ w_1(y)\ c_3\ c_1\ c_2$
  - da es in s_1 kein Paar von strikt sequentiellen TAs gibt, ist OCSR-Bedingung automatisch erfüllt
  - s_1 fällt damit in die Klasse OCSR, aber nicht in Gen(2PL)
- **Theorem: Gen(SS2PL) $\subset$ Gen(S2PL) $\subset$ Gen(2PL)**
- **Theorem: Gen(SS2PL) $\subset$ COCSR**
  - Erinnerung:
  - Eine Historie bewahrt die Commit-Reihenfolge gdw. die Reihenfolge der Commits der einer Serialisierungsreihenfolge entspricht
  - wird im Kontext verteilter Systeme ausgenutzt

Deadlocks

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

G:



WfG:

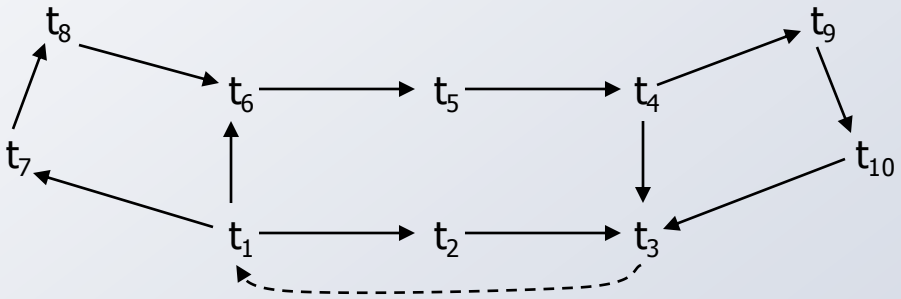
- **Deadlock-Erkennung**
  - Aufbau eines dynamischen Wait-for-Graph (WfG) mit aktiven TAs als Knoten und Wartebeziehungen als Kanten: Eine Kante von t_i nach t_j ergibt sich, wenn t_i auf eine von t_j gehaltene Sperre wartet.
- **Testen des WfG zur Zyklenerkennung**
  - kontinuierlich (bei jedem Blockieren)
  - periodisch (z.B. einmal pro Sekunde)

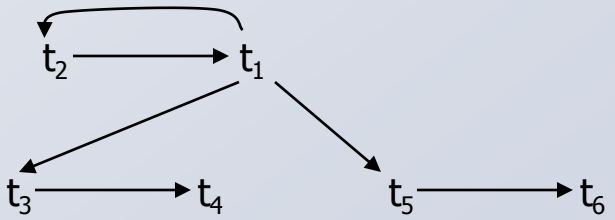
Deadlocks (2)

- **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

- **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

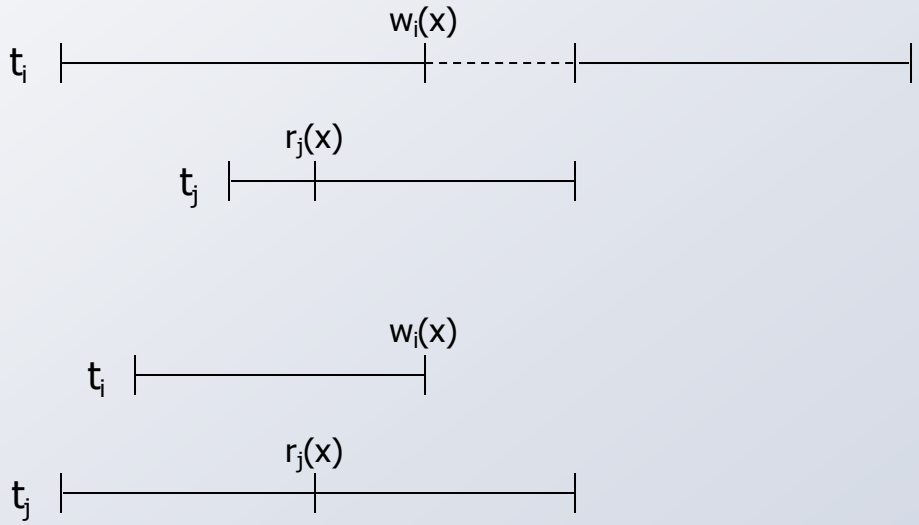
Deadlocks (3)

Prinzip der Deadlock-Verhütung

- Blockierungen (lock waits) werden eingeschränkt, so dass stets ein azyklischer WfG gewährleistet werden kann

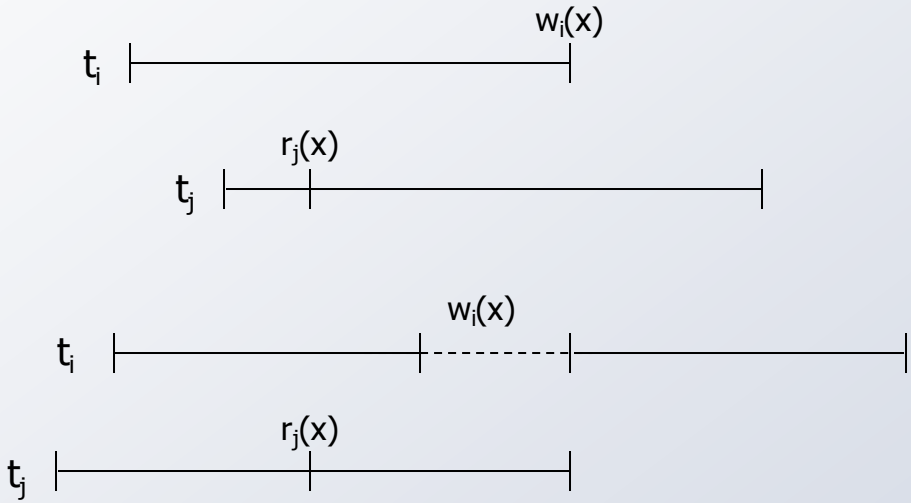
Strategien zur Deadlock-Verhütung

- **Wait-Die**:
  Sobald Sperranforderung von t_i zu Konflikt mit t_j führt
  (kurz: *sobald t_i mit t_j in Konflikt gerät*):
  Wenn **t_i vor t_j** gestartet ist (älter ist), dann **wait(t_i)**, sonst **restart(t_i)**
  (TA wird nur von jüngeren blockiert)



Deadlocks (4)

- **Wound-Wait**: Sobald t_i mit t_j in Konflikt gerät:
  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 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**
  - TA bekommt bei BOT einen systemweit eindeutigen Zeitstempel; er legt die Serialisierbarkeitsreihenfolge fest
  - TA hinterlässt den Wert ihres Zeitstempels bei 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: $p_i(x) <_s q_j(x)$ gdw $ts(t_i) < ts(t_j)$
- **Theorem: Gen(TO) $\subset$ CSR**
- **Prinzipielle Arbeitsweise**
  - Vergabe von aufsteigend eindeutigen TA-IDs (Zeitstempel ts der TA)
  - „Stempeln" des Objektes O bei Zugriffen von t_i : $TS(O) := ts(t_i)$
  - **Konfliktprüfung (sehr einfach!):**

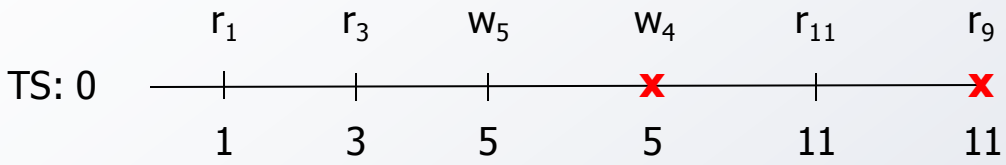
```
if   ts(ti) < TS(O)     then     ABORT
                        else     verarbeite;
```

⇨ Wenn eine Transaktion „zu spät" kommt, wird sie zurückgesetzt und muss wiederholt werden

	O_n	O_m	O_k
T7:	$TS(O_n) = 5$	$TS(O_m) = 3$	$TS(O_k) = 10$

Zeitstempelverfahren (2)

- **Zugriffsfolge auf Objekt O** (nur ein allgemeiner Zeitstempel TS):

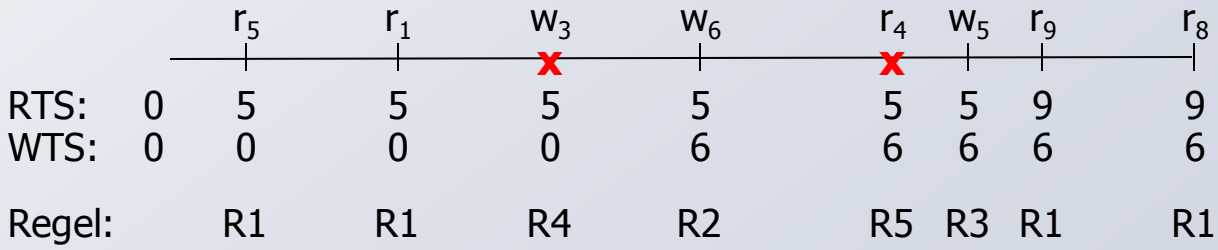


⇨ kein Konflikt bei r_9!

- **Verfeinerung: 2 Zeitstempel pro Objekt**
  - Erhöhung von WTS beim Schreiben und RTS beim Lesen
  - Regeln für t_i und O (Abk. $ts(t_i) = i$)

R1:	r_i	$\wedge$	$(i \geq WTS(O))$	$\Rightarrow$ if RTS(O) < i then RTS(O) := i; Lesen
R2:	w_i	$\wedge$	$(i \geq RTS(O)) \wedge (i \geq WTS(O))$	$\Rightarrow$ WTS(O) := i; Ändern
R3:	w_i	$\wedge$	$(i \geq RTS(O)) \wedge (i < WTS(O))$	$\Rightarrow$ kein Konflikt *(blind update)* – kein Schreiben – weiter[2]
R4:	w_i	$\wedge$	$(i < RTS(O))$	$\Rightarrow$ Zurücksetzen
R5:	r_i	$\wedge$	$(i < WTS(O))$	$\Rightarrow$ Zurücksetzen

- **Zugriffsfolge auf Objekt O:**

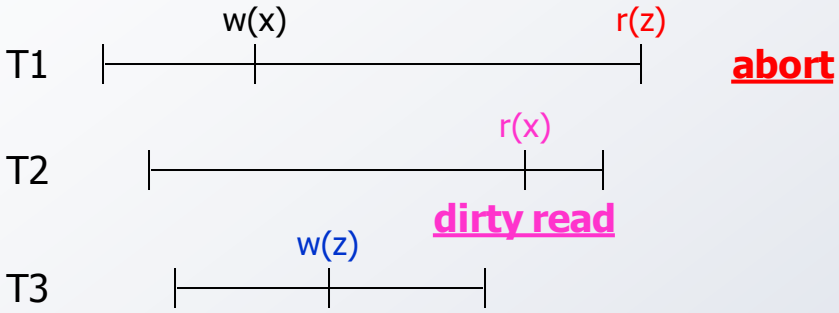


2. Diese Regel wird in der Literatur mit "Thomas' Write Rule" bezeichnet

Zeitstempelverfahren (3)

- **Wo liegt das Problem?**

  $ts(t_i) = i$



- **Vorkehrungen für den ABORT-Fall**
  - Sofortige Zulassung aller Schreiboperationen erzeugt inkonsistente DB
  - Einfrieren der Zeitstempel bis COMMIT der ändernden TA
  - Basic Timestamp Ordering (BTO)[3] schlägt Lösung mit Sperrverfahren und Verwaltung komplexer Abhängigkeiten vor
- **Eigenschaften von TO**
  - Serialisierbarkeitsreihenfolge einer Transaktion wird bei BOT festgelegt
  - Deadlocks sind ausgeschlossen
  - aber: (viel) höhere Rücksetzraten als pessimistische Verfahren
  - ungelöste Probleme, z. B. wiederholter ABORT einer Transaktion
- **Hauptsächlicher Einsatz**
  - Synchronisation in Verteilten DBS
  - lokale Prüfung der Serialisierbarkeit direkt am Objekt O_i (geringer Kommunikations-aufwand)

3. Bernstein, P.A., Goodman, N.: Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems, in: Proc. 6th Int. Conf. on VLDB, 1980, 285-300

Nicht-sperrende Protokolle – Serialization Graph Testing

- **Erinnerung:**
  CSR wird erreicht, wenn der Konfliktgraph G stets azyklisch gehalten wird

- **Theorem: Gen(SGT) = CSR**

- **SGT-Protokoll**
  1. Erzeuge einen neuen Knoten im Graph für TA t_i, 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

- **Beispiel:** $r_1(x)$ $r_2(z)$ $w_3(y)$ $r_4(x)$ $w_2(x)$ ...

  G:

Nicht-sperrende Protokolle – Serialization Graph Testing

- **Löschen von Knoten**
  - 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 (Mengen der von einer TA gelesenen und aktualisierten Objekte) von bereits abgeschlossenen TA erforderlich
  - Deshalb ist SGT **ungeeignet für praktische Implementierungen**

  - Wann kann ein Knoten aus G gelöscht werden?

- **Beispiel:**

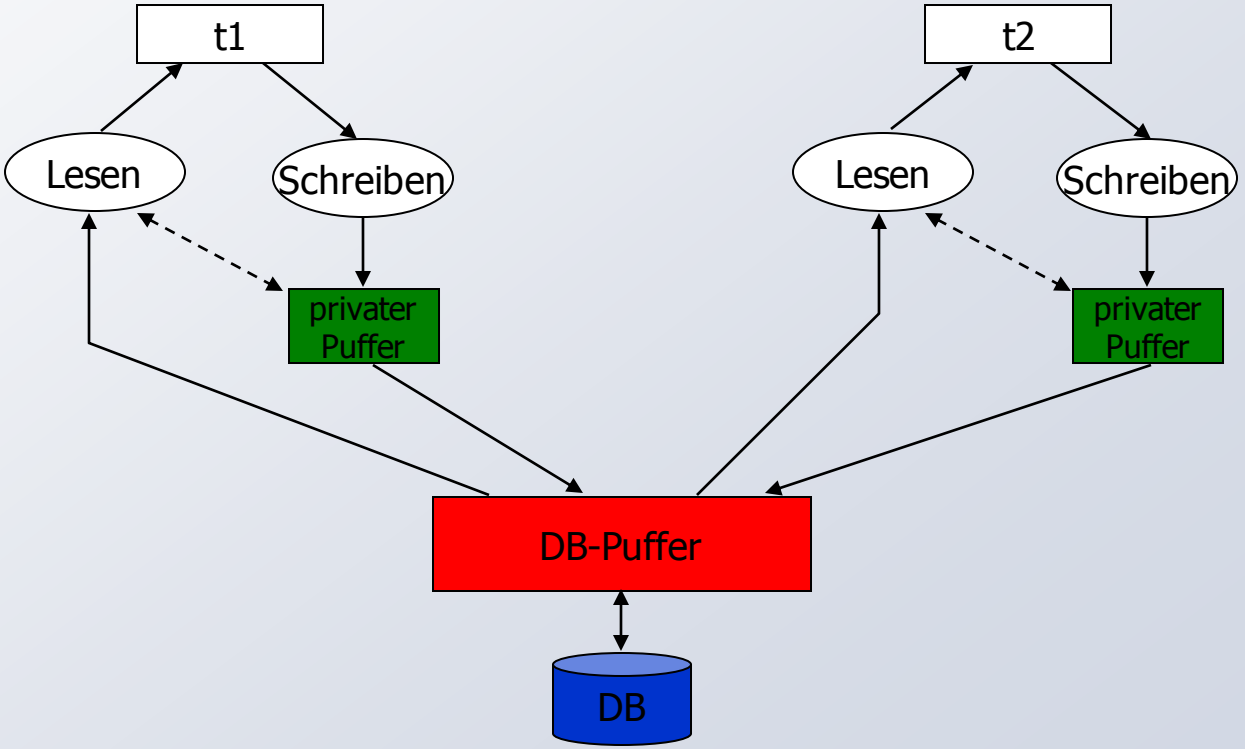
  $s = w_1(x) \quad w_2(x) \quad w_2(y) \quad c_2 \quad w_1(y) \quad c_1 \quad \notin \quad CSR$

  G:

Optimistische Synchronisation (OCC)

- **Dreiphasige Verarbeitung:**

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



Optimistische Synchronisation (2)

- **Validierungsphase**
  - Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen passiert ist
  - Konfliktauflösung durch Zurücksetzen von Transaktionen
- **Schreibphase**
  - nur bei positiver Validierung
  - Lese-Transaktion ist ohne Zusatzaufwand beendet
  - Schreib-Transaktion schreibt hinreichende Log-Information und propagiert ihre Änderungen
- **Grundannahme: geringe Konfliktwahrscheinlichkeit**
- **Allgemeine Eigenschaften von OCC:**
  - \+ einfache TA-Rücksetzung
  - \+ keine Deadlocks
  - \+ potentiell höhere Parallelität als bei Sperrverfahren
  - \- mehr Rücksetzungen als bei Sperrverfahren
  - \- Gefahr des „Verhungerns" von TAs

Optimistische Synchronisation (3)

- **Durchführung der Validierungen:**
  Pro Transaktion werden geführt
  - Read-Set (RS) und
  - Write-Set (WS)

- **Forderung:**
  Eine TA kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten TAs gesehen hat

  ⇒ Validierungsreihenfolge bestimmt Serialisierbarkeitsreihenfolge

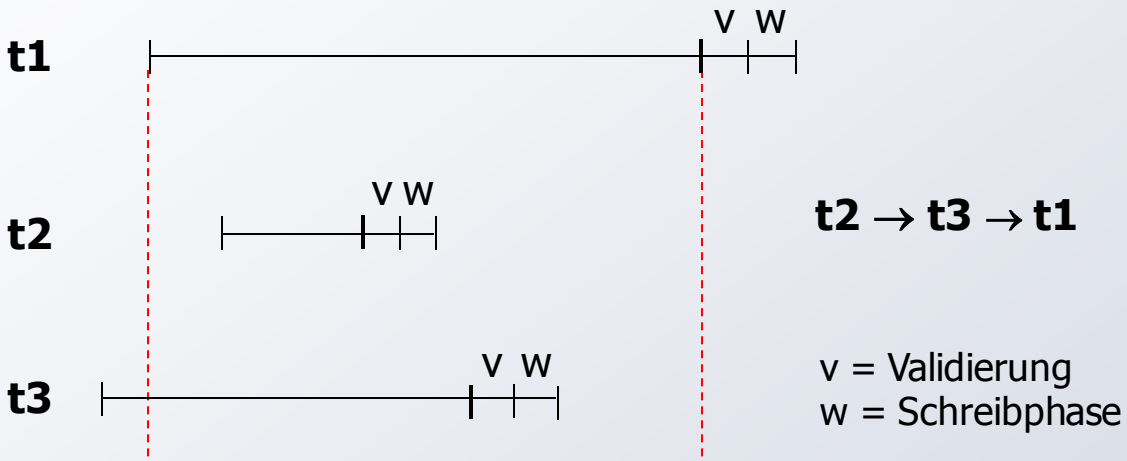
- **Validierungsstrategien:**
  - **Backward Oriented (BOCC):**
    Validierung gegenüber bereits beendeten (Änderungs-) TAs
  - **Forward Oriented (FOCC):**
    Validierung gegenüber laufenden TA

B O C C

- Erstes publizierte Verfahren zur optimistischen Synchronisation[4]
- **Validierung von Transaktion t:**

  BOCC-Test gegenüber allen Änderungs-TA t_j, die seit BOT von t erfolgreich validiert haben:

  IF $RS(t) \cap WS(t_j) \neq \emptyset$ THEN ABORT t
  ELSE SCHREIBPHASE

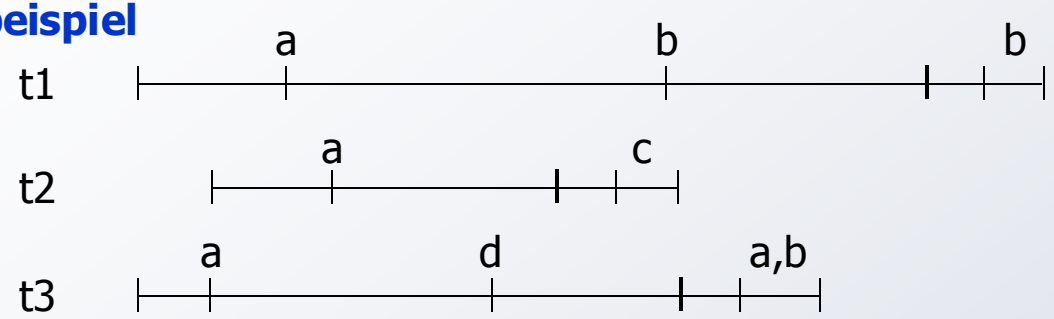


- **Theorem: Gen(BOCC) ⊆ CSR**
- **Nachteile/Probleme:**
  - unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
  - Aufbewahren der Write-Sets beendeter TA erforderlich
  - hohe Anzahl von Vergleichen bei Validierung
  - Rücksetzung erst bei EOT ⇨ viel unnötige Arbeit
  - Nur die validierende TA kann zurückgesetzt werden ⇨ Gefahr von 'starvation'
  - hohes Rücksetzrisiko für lange TA und bei *Hot Spots*

4. Kung, H.T., Robinson, J.T.: On optimistic method for concurrency control, in: ACM Trans. on Database Systems 6:2, 1981, 213-226

B O C C (2)

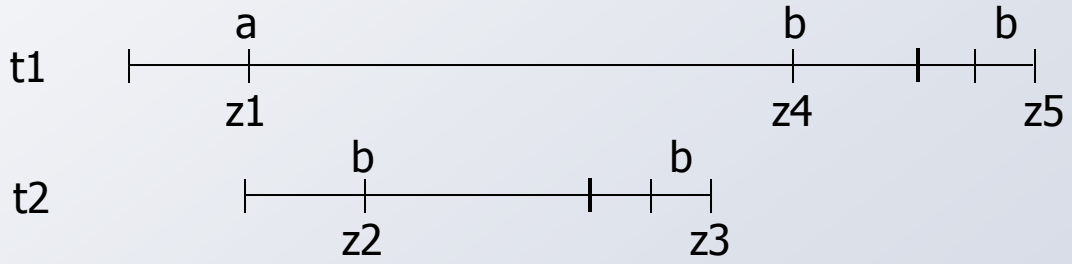
- **Ablaufbeispiel**



⇨ Validierung von t1:

1. RS(t1) ∩ WS(t2) =
2. RS(t1) ∩ WS(t3) =

- **Optimierung von BOCC**



Zeitstempel in WS (Schreibzeitpunkt) und in RS (erste Referenz)

⇨ Validierung von t1:

t1 : RS
t2 : WS
RS(t1) ∩ WS(t2) =

Zusätzliche Prüfung
Write(t2) = Ref(t1) =

BOCC+

Variation des Verfahrens

- Konflikterkennung über Zeitstempel (Änderungszähler) statt Mengenvergleich
- erfolgreich validierte TA erhalten eindeutige, monoton wachsende TA-Nummer
- geänderte Objekte erhalten TA-Nummer der ändernden TA als Zeitstempel TS
- beim Lesen eines Objektes wird Zeitstempel ts der gesehenen Version im Read-Set vermerkt
- **Validierung** überprüft, ob gesehene Objektversionen zum Validierungszeitpunkt noch aktuell sind:

```
VALID := true;
for all r in RS (t) do
  if ts (r,t) < TS (r) then VALID := false;
                /* ts (r,t) ist der Zeitstempel des Objektes r zum Lesezeitpunkt von t
                /* TS (r) der zum Validierungszeitpunkt
end;
if VALID then do
  TNC := TNC + 1;          /* ergibt TA-ID r für t
  for all w in WS (t) do
                TS (w) := TNC;
                /* Einsparen unnötiger Arbeit
                setze alle laufenden TA mit w in RS zurück;
  end;
  Schreibphase für t;
end;
else (setze t zurück);
```

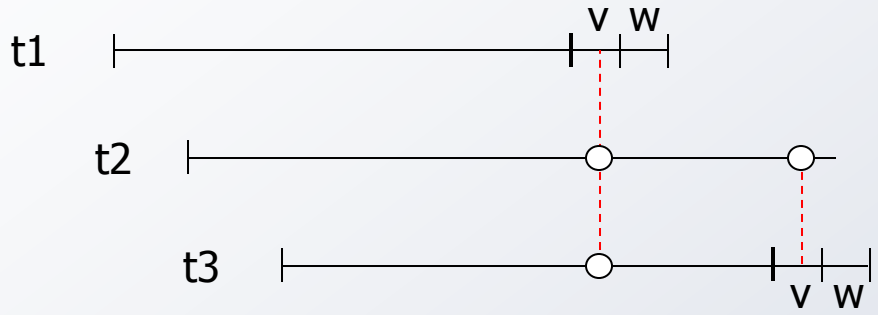
- Zeitstempel TS für geänderte Objekte können zur Durchführung der Validierungen in Objekttabelle geführt werden

BOCC+ (2)

- **Vorteile**
  - Zum Scheitern verurteilte TA können sofort zurückgesetzt werden
  - keine unnötigen Rücksetzungen
  - sehr schnelle Validierung
- **Probleme:**
  - wie bei BOCC ist 'starvation' einzelner TA möglich
  - potentiell hohe Rücksetzrate
- **Lösungsmöglichkeiten:**
  - Reduzierung der Konfliktwahrscheinlichkeit, z. B. durch
    - geringere Konsistenzebene
      (Lese-TA werden bei Validierung nicht mehr berücksichtigt)
    - Mehrversionen-Verfahren
  - Kombination mit Sperrverfahren

F O C C[5]

- Nur Änderungs-TA validieren gegenüber laufenden TA t_i
- **Validierungstest:** $WS(t) \cap RS(t_j) = \varnothing$



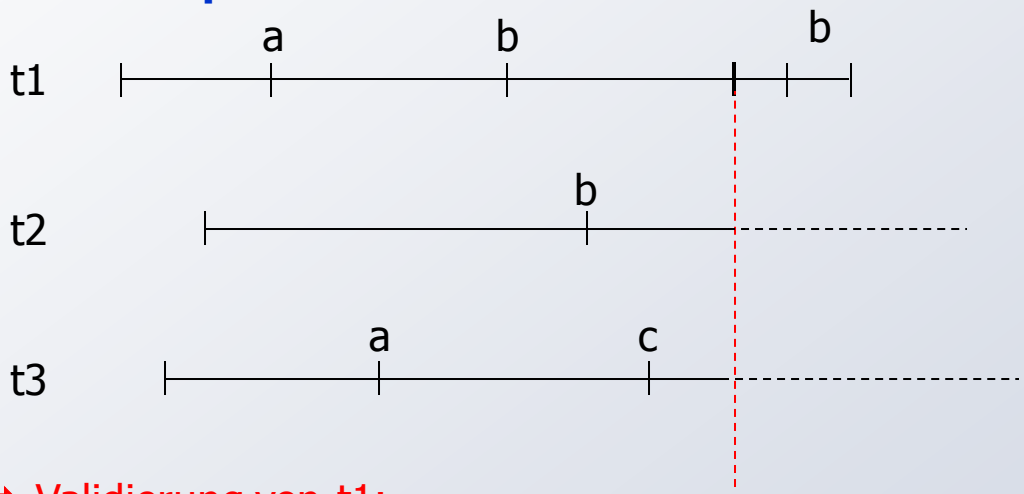
- **Theorem: Gen(FOCC) $\subseteq$ CSR**
  FOCC garantiert sogar COCSR
- **Vorteile:**
  - Wahlmöglichkeit des Opfers (Kill, Abort, Prioritäten, ...)
  - keine unnötigen Rücksetzungen, frühzeitige Rücksetzung möglich
    ⇨ Einsparen unnötiger Arbeit
  - keine Aufbewahrung von Write-Sets, geringerer Validierungsaufwand als bei BOCC

5. Härder, T.: Observations on optimistic concurrency control schemes, Information Systems 9:2, 1984, 111-120

F O C C (2)

- **Probleme:**
  - Während Validierungs- und Schreibphase müssen die Objekte von WS (t) „gesperrt" sein, damit sich die zu prüfenden RS (t_i) nicht ändern (keine Deadlocks damit möglich)
  - immer noch hohe Rücksetzrate möglich
  - Es kann immer einer TA ein Durchkommen definitiv zugesichert werden

- **Ablaufbeispiel**



⇨ Validierung von t1:

1. WS(t1) ∩ RS(t2) =

2. WS(t1) ∩ RS(t3) =

Zusammenfassung

Scheduler
Klassifikation
Sperrprotokolle
Deadlocks
Zeitstempel
SGT
OCC

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch" Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
- **Realisierung der Synchronisation durch Sperrverfahren**
  - Sperren stellen während des laufenden Betriebs sicher, dass die resultierende Historie serialisierbar bleibt
  - Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt
  - Es gibt mehrere Varianten

  ⇨ **Sperrverfahren sind pessimistisch und universell einsetzbar**
- **Deadlock-Problem ist bei blockierenden Verfahren inhärent!**
- **Zeitstempel-Verfahren und Optimistische Protokolle**
  - nur konzeptionell einfach
  - Reine OCC- und Zeitstempel-Verfahren erzeugen zu viele Rücksetzungen
  - Einsatz als hybride Verfahren möglich:
    Sie müssen mit Sperrverfahren kombiniert werden