Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Klasse CSR (Konfliktserialisierbarkeit)

Ziel

- VSR taugt nicht für den praktischen Einsatz
  - VSR ist nicht monoton
  - Testen der VSR-Mitgliedschaft ist NP-vollständig!
- Deshalb weitere Einschränkungen
- Konzept, das einfach zu testen ist und sich für den Einsatz in Schedulern eignet

Definition: Konflikt und Konfliktrelation

- Sei s ein Schedule; $t, t' \in trans(s)$ und $t \neq t'$:
- Zwei Datenoperationen $p \in t$ und $q \in t'$ sind in Konflikt in s, wenn sie auf dasselbe Datenobjekt zugreifen und wenigstens eine von ihnen eine Schreiboperation (Write) ist.
- $conf(s) := \{(p,q) | p,q$ sind in Konflikt in s und $p <_s q\}$ heißt Konfliktrelation von s

Klasse CSR - Konfliktäquivalenz

Definition: Konfliktäquivalenz

- Schedules s und s' sind konfliktäquivalent, ausgedrückt durch $s \approx_c s'$, wenn
  - $op(s) = op(s')$
  - $conf(s) = conf(s')$

Beispiel

- $s = r_1(x)\ r_1(y)\ w_2(x)\ w_1(y)\ r_2(z)\ w_1(x)\ w_2(y)$
- $s' = r_1(y)\ r_1(x)\ w_1(y)\ w_2(x)\ w_1(x)\ r_2(z)\ w_2(y)$

Gilt hier $conf(s) = conf(s')$?

Klasse CSR - Konfliktserialisierbarkeit

Definition: Konfliktserialisierbarkeit

- Eine Historie s ist konfliktserialisierbar, wenn eine serielle Historie s' mit $s \approx_c s'$ existiert.
- CSR bezeichnet die Klasse aller konfliktserialisierbaren Historien.

Beispiele

- $s_1 = r_1(x)\ \ r_2(x)\ \ r_1(z)\ \ w_1(x)\ \ w_2(y)\ \ r_3(z)\ \ w_3(y)\ \ c_1\ \ c_2\ \ w_3(z)\ \ c_3$
- $s_1 \in$ CSR
- $s_2 = r_2(x)\ \ w_2(x)\ \ r_1(x)\ \ r_1(y)\ \ r_2(y)\ \ w_2(y)\ \ c_1\ \ c_2$
- $s_2 \notin$ CSR

Klasse CSR: Konfliktschritte-Graph

Definition: Konfliktschritte-Graph $D(s)$

- Konfliktäquivalenz lässt sich durch einen Graph $D(s) := (V,E)$ mit $V = ops(s)$ und $E = conf(s)$ veranschaulichen. $D(s)$ heißt Konfliktschritte-Graph (conflicting-step graph) und es gilt

$$s \approx_c s' \Leftrightarrow D(s) = D(s')$$

Beispiel

- $s = r_1(x)\ r_2(x)\ w_1(x)\ r_3(x)\ w_3(x)\ w_2(y)\ c_3\ c_2\ w_1(y)\ c_1$
- $D(s) =$

Klasse CSR: Konfliktgraph

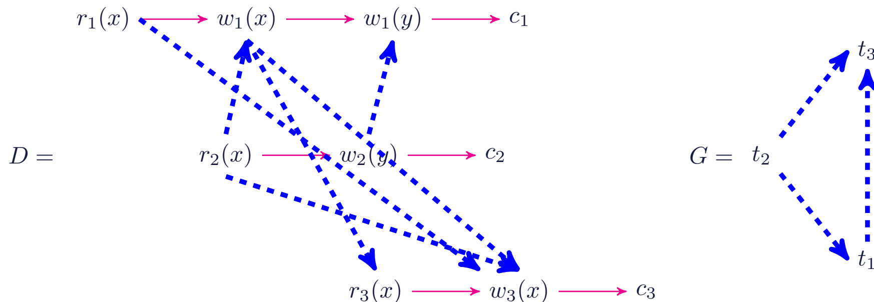
Definition: Konfliktgraph (Serialisierungsgraph)

- Sei s ein Schedule. Der Konfliktgraph $G(s) = (V,E)$ ist ein gerichteter Graph mit
  - $V = commit(s)$
  - $(t,t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t)(\exists q \in t')(p,q) \in conf(s)$

Anmerkung

- Der Konfliktgraph abstrahiert von individuellen Konflikten zwischen Datenoperationen, wie sie im Konfliktschritte-Graph (nach $conf(s)$) beschrieben sind und repräsentiert Konflikte zwischen (abgeschlossenen) Transaktionen (durch eine Kante)

Beispiel



- Kante $w_1(x) \to r_3(x)$ aus D führt zur Kante $t_1 \to t_3$ des SG
- weitere Kanten analog
- (Dünne Pfeile beschreiben Ordnung innerhalb einer TA)

Ist diese Historie konfliktserialisierbar?

Klasse CSR - Serialisierbarkeitstheorem

Serialisierbarkeitstheorem

- Sei s eine Historie; dann gilt: $s \in$ CSR genau dann wenn $G(s)$ azyklisch ist.

Korollar

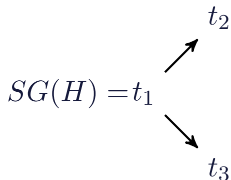
- Mitgliedschaft in CSR lässt sich in polynomialer Zeit in der Menge der am betreffenden Schedule teilnehmenden Transaktionen testen.

Serialisierbarkeitstheorem: Beispiel

Historie

$$H = w_1(x)\ w_1(y)\ c_1\ r_2(x)\ r_3(y)\ w_2(x)\ c_2\ w_3(y)\ c_3$$

Serialisierbarkeitsgraph:



Topologische Ordnung(en):

$$H_s^1 = t_1|t_2|t_3$$

$$H_s^2 = t_1|t_3|t_2$$

$$H \equiv H_s^1 \equiv H_s^2$$

Klasse CSR - Zusammenhang zu View-Serialisierbarkeit

Der Vollständigkeit wegen erwähnt:

Blindes Schreiben

- Ein blindes Schreiben eines Datenobjekts x liegt vor, wenn eine TA ein $w(x)$ ohne ein vorhergehendes $r(x)$ durchführt
- Wenn wir blindes Schreiben für Transaktionen verbieten, verschärft sich die Definition einer TA um die Bedingung: Wenn $w_i(x) \in t_i$, dann gilt auch $r_i(x) \in t_i$ und $r_i(x) < w_i(x)$
- Dann gilt: Eine Historie ist view-serialisierbar gdw. sie konflikt-serialisierbar ist!

Klasse CSR - Konflikte und Kommutativität

Konflikte und Kommutativität

- bisher wurde Konfliktserialisierbarkeit über den Konfliktgraph G definiert
- Ziel jetzt:
  - s soll mit Hilfe von Kommutativitätsregeln schrittweise so transformiert werden, dass eine serielle Historie entsteht
  - s ist dann äquivalent zu einer seriellen Historie

Definition: Kommutativitätsbasierte Äquivalenz

- Zwei Schedules s und s' mit $op(s) = op(s')$ sind kommutativitätsbasiert äquivalent, ausgedrückt durch $s \sim^* s'$, wenn s nach s' transformiert werden kann durch eine endliche Anwendung der nachfolgenden Regeln C1, C2, C3 und C4.

Klasse CSR - Kommutativitätsregeln (C1, C2, C3, C4)

Kommutativitätsregeln

- $\sim$ bedeutet, dass die geordneten Paare von Aktionen gegenseitig ersetzt werden können

  C1: $r_i(x)\ r_j(y) \sim r_j(y)\ r_i(x)$, wenn $i \neq j$

  C2: $r_i(x)\ w_j(y) \sim w_j(y)\ r_i(x)$, wenn $i \neq j, x \neq y$

  C3: $w_i(x)\ w_j(y) \sim w_j(y)\ w_i(x)$, wenn $i \neq j, x \neq y$

- Ordnungsregel bei partiell geordneten Schedules:

  C4: $o_i(x), p_j(y)$ ungeordnet $\Rightarrow o_i(x) p_j(y)$, wenn $x \neq y \vee (o = r \wedge p = r)$ besagt, dass zwei ungeordnete Operationen beliebig geordnet werden können, wenn sie nicht in Konflikt stehen

$s = w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ w_2(y)\ w_3(y)\ w_3(z)$

Regel C2 $\rightarrow$ $w_1(x)\ w_1(y)\ r_2(x)\ w_1(z)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

Regel C2 $\rightarrow$ $w_1(x)\ w_1(y)\ w_1(z)\ r_2(x)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

$= t_1\ t_2\ t_3$

Klasse CSR - Kommutativitätsbasierte Reduzierbarkeit

Definition: Kommutativitätsbasierte Reduzierbarkeit

- Historie s ist kommutativitätsbasiert reduzierbar, wenn es eine serielle Historie s' gibt mit $s \sim^* s'$

Theorem

- s und s' seien Schedules mit $op(s) = op(s')$, dann gilt

$$s \approx_c s' \text{ gdw } s \sim^* s'$$

Korollar

- Eine Historie s ist kommutativitätsbasiert reduzierbar genau dann wenn (gdw) $s \in CSR$.

Anmerkung: Schedules erzeugt nach dem 2PL Protokoll sind in CSR

Klasse OCSR

Einschränkungen der Konflikt-Serialisierbarkeit

- Historien/Schedules aus VSR und FSR lassen sich praktisch nicht nutzen.
- Weitere Einschränkungen von CSR dagegen sind in manchen praktischen Anwendungen sinnvoll.

Beispiel

- $s_{312} = w_1(x) \quad r_2(x) \;\; c_2 \;\; w_3(y) \;\; c_3 \quad w_1(y) \;\; c_1$
- $G(s_{312}) =$

- Kontrast zwischen Serialisierungs- und tatsächlicher Ausführungsreihenfolge möglicherweise unerwünscht!
- Situation lässt sich durch Ordnungserhaltung vermeiden

Klasse OCSR (2)

Ordnungserhaltende Konfliktserialisierbarkeit

- Eine Historie s heißt ordnungserhaltend konfliktserialisierbar (order-preserving serializable) wenn
  - sie konfliktserialisierbar ist, d.h. es existiert ein s', so dass $op(s) = op(s')$ und $s \approx_c s'$ gilt und
  - wenn zusätzlich folgendes für alle $t_i, t_j \in trans(s)$ gilt: Wenn t_i vollständig vor t_j in s auftritt, dann gilt dasselbe auch für s'

Theorem

- OCSR bezeichne die Klasse aller ordnungserhaltenden konfliktserialisierbaren Historien; Es gilt

$$\text{OCSR} \subset \text{CSR}$$

Klasse OCSR (3)

Beweisskizze

- Aus der Definition folgt: OCSR $\subseteq$ CSR
- $s_{312} = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1$
- s_{312} zeigt, dass die Inklusionsbedingung echt ist:
  $s_{312} \in$ CSR $-$ OCSR

Klasse COCSR

Definition: Einhaltung der Commit-Reihenfolge

- Eine Historie s hält die Commit-Reihenfolge ein (commit order-preserving conflict serializable), wenn folgendes gilt:

  Für alle $t_i, t_j \in commit(s)$ mit $i \neq j$:

  Wenn $(p,q) \in conf(s)$ für $p \in t_i$ und $q \in t_j$, dann $c_i < c_j$ in s

Die Reihenfolge der Konfliktoperationen bestimmt die Reihenfolge der zugehörigen Commit-Operationen

Theorem

- COCSR bezeichne die Klasse aller Historien, die "commit order-preserving conflict serializable" sind; es gilt

$$\text{COCSR} \subset \text{CSR}$$

Klasse COCSR (2)

Beweisskizze

- $s = r_1(x)\ \ w_2(x)\ \ c_2\ \ c_1$
- $s \in$ CSR $-$ COCSR (die Inklusion ist also echt)

Theorem

- Sei s eine Historie:

  $s \in$ COCSR gdw

  $s \in$ CSR und es existiert eine serielle Historie s', so dass $s' \approx_c s$ und für alle $t_i, t_j \in trans(s) : t_i <_{s'} t_j \Rightarrow c_{t_i} <_s c_{t_j}$

Theorem

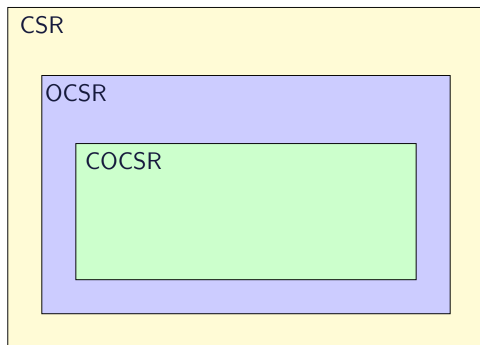
COCSR $\subset$ OCSR

Beispiel und Beziehungen zwischen CSR, OCSR und COCSR

$$s_1 = w_1(x)\ \ r_2(x)\ \ c_2\ \ w_3(y)\ \ c_3\ \ w_1(y)\ \ c_1 \in \text{ CSR } - \text{ OCSR}$$

$$s_2 = w_3(y)\ \ c_3\ \ w_1(x)\ \ r_2(x)\ \ c_2\ \ w_1(y)\ \ c_1 \in \text{ OCSR } - \text{ COCSR}$$

$$s_3 = w_3(y)\ \ c_3\ \ w_1(x)\ \ r_2(x)\ \ w_1(y)\ \ c_1\ \ c_2 \in \text{ COCSR}$$



CSR Plausibilitätscheck

Lost Update

- $L = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$
- $conf(L) = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x), w_2(x))\}$
- $L \not\approx_c t_1 t_2$ und $L \not\approx_c t_2 t_2$

Inconsistent Read

- $I = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y) w_2(y) c_1 c_2$
- $conf(I) = \{(w_2(x), r_1(x)), (r_1(y), w_2(y))\}$
- $I \not\approx_c t_1 t_2$ und $I \not\approx_c t_2 t_2$

Monotonie von CSR und CSR $\subset$ VSR

Theorem

- CSR ist monoton
- Und CSR ist die größte monotone Teilmenge von VSR.

Der Vollständigkeit wegen erwähnt:

Theorem

CSR $\subset$ VSR

Korollar

CSR $\subset$ VSR $\subset$ FSR

Beispiel

- $s_{VSR} = r_1(x)\ w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$
- $s \not\approx_c t_1 t_2 t_3$ und $s \notin CSR$, aber
- $s \approx_v t_1 t_2 t_3$ und damit $s \in VSR$

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

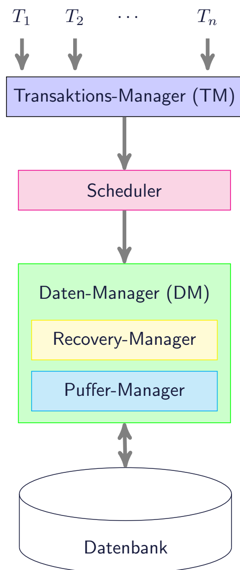
Sperrprotokolle

- 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)

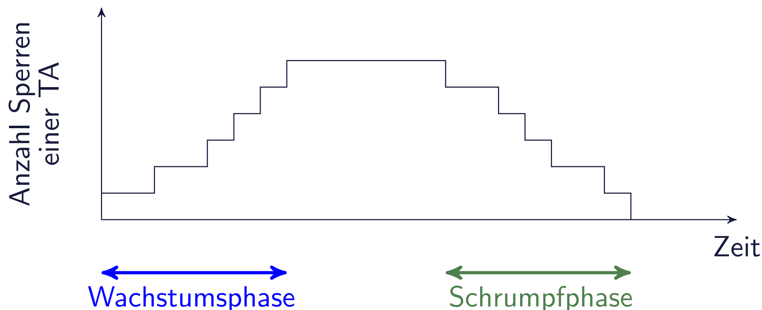
- LR1: Jeder Datenoperation $r_i(x)$ $[w_i(x)]$ muss ein $rl_i(x)$ $[wl_i(x)]$ (Sperre) vorausgehen und ein $ru_i(x)$ $[wu_i(x)]$ (Freigabe) folgen.
- LR2: Es gibt höchstens ein $rl_i(x)$ und ein $wl_i(x)$ für jedes x und t_i
- LR3: Es ist kein $ru_i(.)$ oder $wu_i(.)$ redundant
- LR4: Wenn x durch t_i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

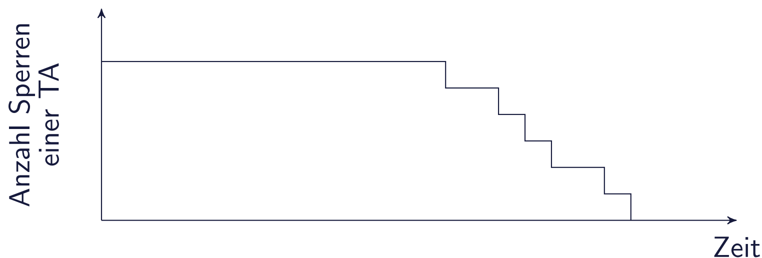
Ein Sperrprotokoll ist **zweiphasig** (2PL), wenn für jeden (Ausgabe-) Schedule s und jede TA $t_i \in trans(s)$ kein ql_i Schritt dem ersten ou_i Schritt folgt ($o, q \in \{r,w\}$).



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.

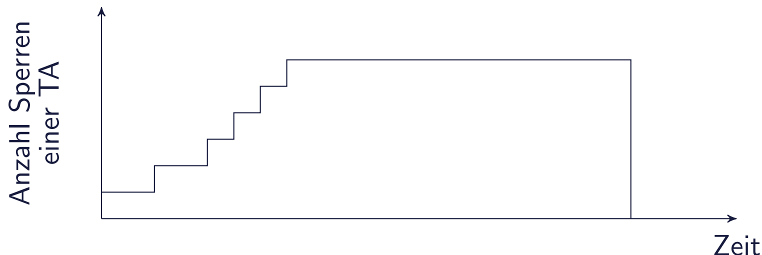


Striktes 2PL (S2PL) + Starkes 2PL (SS2PL)

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:

$wl_1(x)$ $w_1(x)$ $wl_1(y)$ … $w_1(y)$ $wu_1(x)$ $wu_1(y)$ c_1

$rl_2(x)$ … $r_2(x)$ $rl_2(z)$ $r_2(z)$ $ru_2(x)$ $ru_2(z)$ c_2

$rl_3(y)$ $r_3(y)$ $ru_3(y)$ c_3

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