Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Mehrbenutzersynchronisation – Aspekte

Konzept der Serialisierbarkeit

- Final-State-Serialisierbarkeit (FSR)
- View-Serialisierbarkeit (VSR)
- Konflikt-Serialisierbarkeit (CSR)

Synchronisation

- Basierend auf Sperren
- Basierend auf Zeitstempeln
- Behandlung von Deadlocks

Wiederholung: Das lost-update Problem

t_1	Time	t_2
	/* $x = 100$ */	
$r(x)$	1	
	2	$r(x)$
/*update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$*/	

Wiederholung: Das lost-update Problem / Notation

- Die Essenz dieses Problems kann durch folgende Sequenz von Lese- und Schreiboperationen ausgedrückt werden:

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

Wiederholung: Das inconsistent-read Problem

Beispiel aus z.B. Anwendung in Bank. Aktueller Stand $x = y = 50$, also $x + y = 100$. Transaktion t_1 berechnet die Summe von x und y, während t_2 einen Wert von 10 von x nach y transferiert.

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* $sum := 0$ */	4	
$r(x)$	5	
$r(y)$	6	
/* $sum := sum + x$ */	7	
/* $sum := sum + y$ */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Wiederholung: Das inconsistent-read Problem (2)

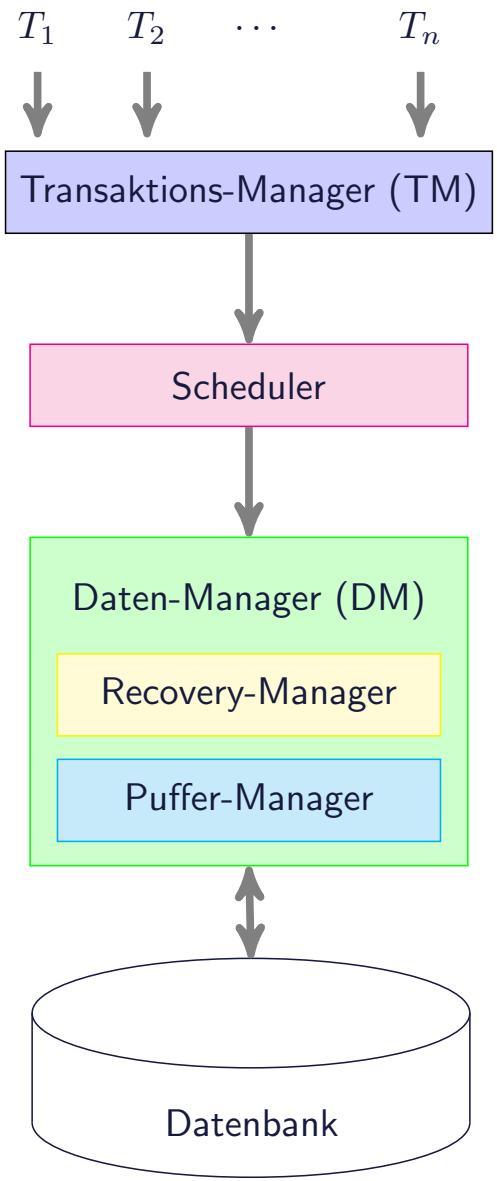
- Offensichtlich ist auch hier wieder das Problem, dass Lese- und Schreiboperationen der einzelnen Transaktionen gemischt ablaufen

$$r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)$$

Wiederholung: Das dirty-read Problem

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
failure & rollback	6	
	7	$w(x)$

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)

Aktionen des Schedulers

Scheduler empfängt Schritte der Form r, w, a und c als Eingabe. Diese müssen in einen serialisierbaren Schedule uberführt werden. Dazu kann Scheduler folgende Aktionen durchführen:

1. **Output:** Schritt (r, w, a oder c) wird an Output-Schedule (am Anfang leer) angehängt.
2. **Reject:** Schritt wird nicht ausgegeben (weil z.B. dies die Serialisierbarkeit des Outputs zerstören würde). Also muss die dazugehörige TA abgesprochen werden.
3. **Block:** Schritt wird nicht ausgegeben und auch nicht rejected, sondern als "momentan nicht ausführbar" angesehen und deshalb auf einen späteren Zeitpunkt verschoben.

Es gibt optimistische und pessimistische Scheduler. Sperrbasierte Scheduler (z.B. nach 2PL) gehören zur Klasse der pessimistischen Scheduler.

Ziel der heutigen Vorlesung

- Formale Betrachtung von verschiedenen Klassen von Schedules
- Welche Probleme mit welcher Klasse behoben werden und welche nicht

Wir betrachten erstmal nicht:

- Wie ein Protokoll aussieht, das z.B. durch Sperren nur Schedules einer bestimmten Klasse erzeugt.
- Dazu später mehr.

Transaktion

- Eine Transaktion t ist eine Partialordnung von Schritten der Form

$$p_i \in \{r(x), w(x)\}$$

  mit $x \in D$ ein Datenobjekt.
- Lese- und Schreiboperationen sowie mehrfache Schreiboperationen auf demselben Datenobjekt sind geordnet.
- Eine vollständige TA hat als letzte Operation entweder einen Abbruch a oder ein Commit c

$$t = p_1 \ldots p_n a$$

  oder

$$t = p_1 \ldots p_n c$$

- Man kann für Partialordnungen vollständige Ordnungen angeben, in dem man nicht geordnete Elements einfach in eine (beliebige) Reihenfolge bringt.
- Achtung: in der Schreibweise $t = p_1\ p_2\ p_3\ \ldots$ ist die Ordnung aller Operationen gegeben (von links nach rechts).

Historien und Schedules

- Es sei $T = \{t_1, \ldots, t_n\}$ eine Menge von Transaktionen, wobei jedes $t_i \in T$ die Form $t_i = \{op_i, <_i\}$ besitzt, op_i die Menge der Operationen von t_i und $<_i$ ihre Ordnung $(1 \leq i \leq n)$ bezeichnen.
- Eine **Historie** für T ist ein Paar $s = (op(s), <_s)$, so dass:
  (a) $op(s) \subseteq \bigcup_{i=1}^{n} op_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$ und $\bigcup_{i=1}^{n} op_i \subseteq op(s)$
  (b) $(\forall i, 1 \leq i \leq n) c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
  (c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$
  (d) $(\forall i, 1 \leq i \leq n)(\forall p \in op_i)\ p <_s a_i$ oder $p <_s c_i$
  (e) Jedes Paar von Operatoren p,$q \in op(s)$ von verschiedenen Transaktionen, die auf dasselbe Datenobjekt zugreifen und von denen wenigstens eine davon eine Schreiboperation ist, sind so geordnet, dass $p <_s q$ oder $q <_s p$ gilt.
- Ein **Schedule** ist ein Präfix einer Historie

Historien und Schedules (2)

Erläuterungen zu den zuvor genannten Punkten (a) bis (e):

- (a) Historie enthält alle Operationen aller Transaktionen
- (b) Historie benötigt eine Terminierungsoperation für jede TA
- (c) Historie bewahrt alle Ordnungen innerhalb der TA
- (d) Terminierungsoperation ist letzte Operation in jeder TA
- (e) Konfliktoperationen sind geordnet.

Historien und Schedules (3)

Bemerkung

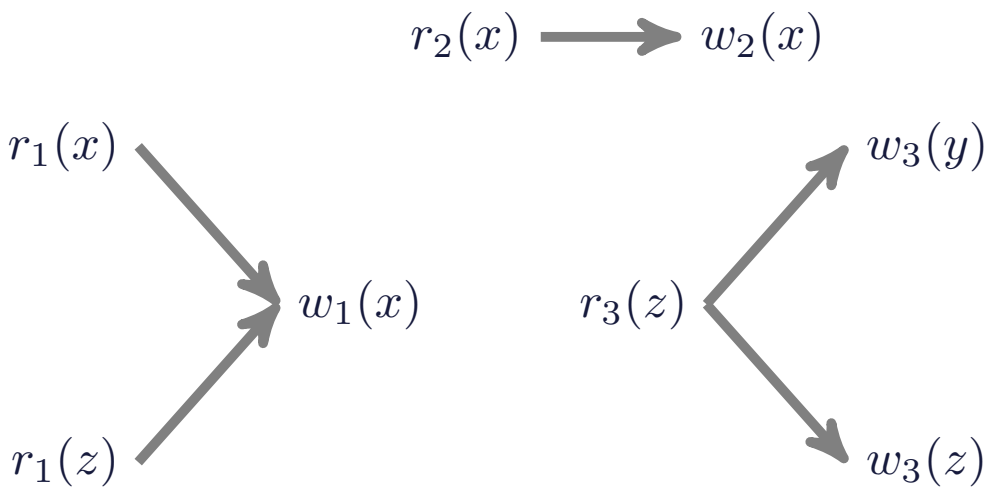
- Wegen (a) und (b) wird eine Historie auch als vollständiger Schedule bezeichnet
- Ein Präfix einer Historie kann die Historie selbst sein
- Historien lassen sich als Spezialfälle von Schedules betrachten; es genügt deshalb meist, einen gegebenen Schedule zu betrachten

Definition: Serielle Historie

- Eine Historie s ist **seriell**, wenn für jeweils zwei TA t_i und t_j $(i \neq j)$ alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.

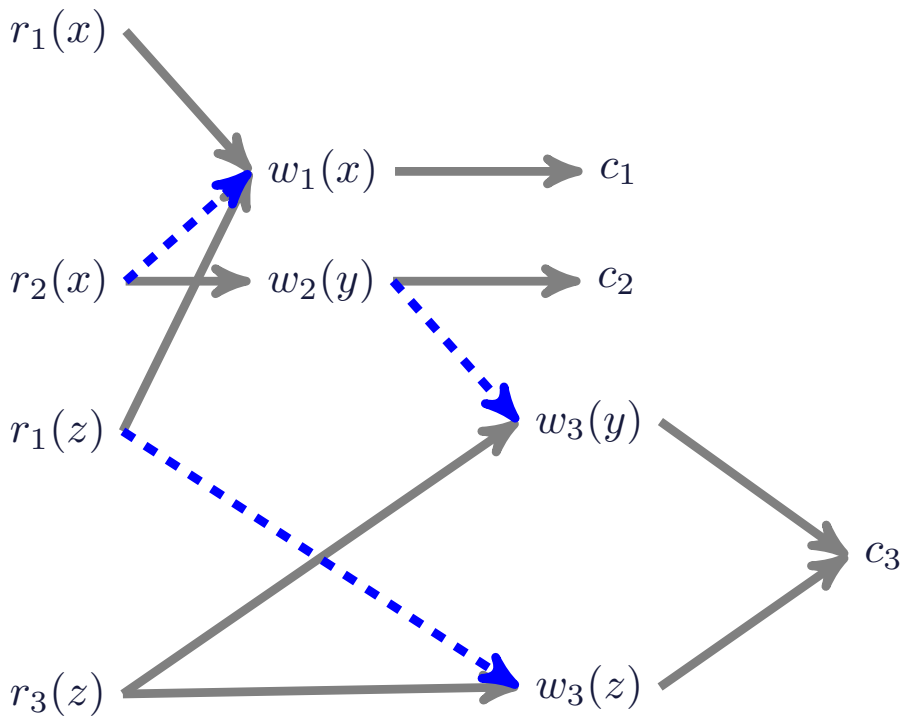
Beispiel

- Wir sehen hier drei Beispiel-Transaktionen, t_1, t_2 und t_3.
- Hier, jeweils, dargestellt als ein gerichteter azyklischer Graph, dessen Kanten die (partielle) Ordnung der Schritte beschreibt.
- Partiell, weil z.B. in Transaktion t_1 nicht festgelegt ist ob $r_1(x)$ vor oder nach $r_1(z)$ ausgeführt werden soll.
- Aber beide Operationen müssen vor $w_1(x)$ ausgeführt werden!



Beispiel (Fortsetzung)

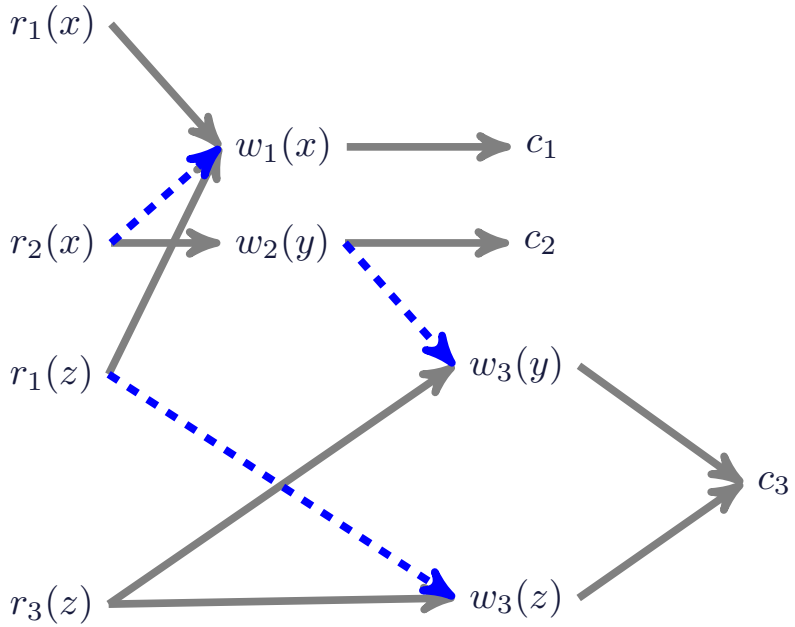
- Nehmen wir noch an, dass diese drei TA auch committen
- Folgende partielle Ordnung ist möglich:



- Die normalen Kanten stammen aus den ursprünglichen TA
- Die blauen gestrichelten Kanten mussten aufgrund von Regel (e) eingefügt werden, da Konfliktoperationen geordnet sein müssen!

Beispiel (Fortsetzung)

Schauen wir uns nochmal die partielle Ordnung von zuvor an:



- Eine mögliche **totale Ordnung** darauf ist gegeben durch (topologisches Sortieren):

$$r_1(x)\ \ r_2(x)\ \ r_1(z)\ \ w_1(x)\ \ w_2(y)\ \ r_3(z)\ \ w_3(y)\ \ c_1\ \ c_2\ \ w_3(z)\ \ c_3$$

Historien und Schedules (4)

Definition: TA-Mengen eines Schedules

- Alle TA in s:

$$trans(s) := \{t_i | s \text{ enthält Schritte von } t_i\}$$

- Alle TA die in s committen:

$$commit(s) := \{t_i \in trans(s) | c_i \in s\}$$

- Alle TA die in s aborten:

$$abort(s) := \{t_i \in trans(s) | a_i \in s\}$$

- Alle aktiven TA in s:

$$active(s) := trans(s) - (commit(s) \cup abort(s))$$

Historien und Schedules (5)

Für jede Historie s gilt

- $trans(s) = commit(s) \cup abort(s)$
- $active(s) = \emptyset$

Beispiel

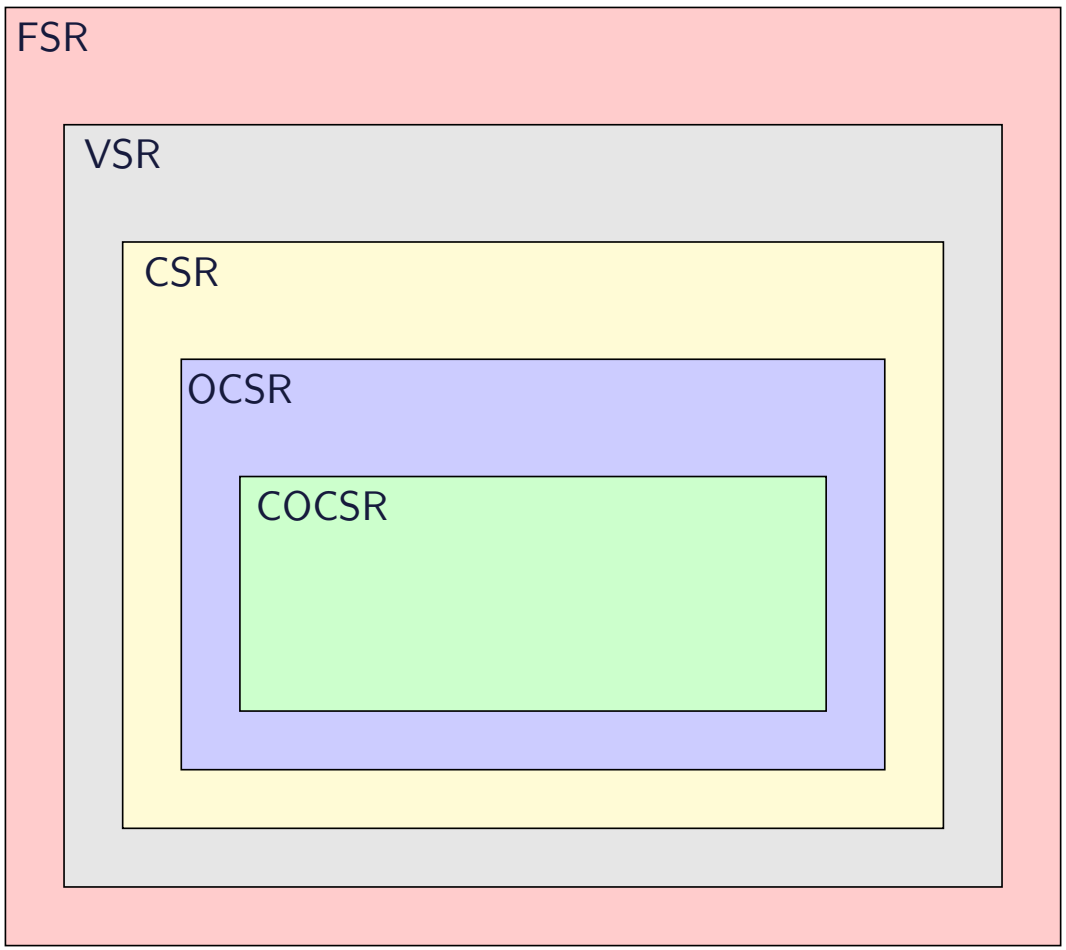
- $s_1 =$
  $r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ c_2\ a_3$
  - $trans(s_1) = \{t_1, t_2, t_3\}$
  - $commit(s_1) = \{t_1, t_2\}$
  - $abort(s_1) = \{t_3\}$
  - $active(s_1) = \emptyset$
- $s_2 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1$
  - $trans(s_1) = \{t_1, t_2, t_3\}$
  - $commit(s_1) = \{t_1\}$
  - $abort(s_1) = \emptyset$
  - $active(s_1) = \{t_2, t_3\}$

Serialisierbarkeitsklassen

Ziel

- formale Betrachtung des Serialisierbarkeitsbegriffs

Klassen



Historien und Schedules - Monotonie

Definition: Monotone Klassen von Historien

Eine Klasse E von Historien heißt monoton, wenn Folgendes gilt:

- Wenn s in E ist, dann ist die Projektion s' von s auf T $s' = \Pi_T(s)$ mit $op(s') = op(s) - \cup_{t \notin T} op(t)$, in E für jedes $T \subseteq trans(s)$
- Mit anderen Worten: E ist unter beliebigen Projektionen abgeschlossen

Monotonie

- Monotonie einer Historienklasse E ist eine wünschenswerte Eigenschaft, da sie E unter beliebigen Projektionen bewahrt!

Serialisierbarkeitsklassen

Akzeptable Klasse von Schedules ...

- muss mindestens **Lost Update** und **Inconsistent Read** ausschließen
- muss Zugehörigkeit eines Schedules effizient entscheiden können
- muss bei Annahme von Fehlern (Aborts) Abhängigkeit von nicht freigegebenen Änderungen (**Dirty Read**) vermeiden:

$$r_1(x)\, w_1(x)\, r_2(x)\, a_1\, w_2(x)\, c_2$$

Klasse FSR (Final-State-Serialisierbarkeit)

Definition: Final-State-Serialisierbarkeit

- Eine Historie s ist final-state-serialisierbar, wenn eine serielle Historie s' existiert, so dass $s \approx_f s'$.

  FSR bezeichnet die Klasse aller final-state serialisierbaren Historien.

Final-State-Serialisierbarkeit

- Final-State-Äquivalenz: $s \approx_f s'$, wenn sie ausgehend vom selben Ausgangszustand denselben Endzustand der DB erzeugen.
- Konsistenter DB-Zustand wird nur am Ende der Historie gewährleistet. FSR macht deshalb nur Sinn für Historien (vollständige Schedules).

Hinweis: $\approx_f$ und später auch $\approx_v$ sind hier nicht formal definiert, da recht komplex; siehe Buch von Weikum und Vossen, Kapitel 3.

Klasse FSR - Beispiel

Ist Historie s_{FSR}, die einen Zyklus enthält, final-state-serialisierbar?

$$s_{FSR} = w_1(x)\ \ r_2(x)\ \ w_2(y)\ \ c_2\ \ r_1(y)\ \ w_1(y)\ \ c_1\ \ w_3(x)\ \ w_3(y)\ \ c_3$$

- Ja, denn es gilt $s_{FSR} \approx_f s'$, mit $s' = t_1 t_2 t_3$
- Wieso?

Annahmen für Beispiele mit konkreten Werten

- Initiale DB $= \{x = 0, y = 0\}$
- r liest aktuellen Wert a
- r vor w: w schreibt $a + 1$
- blindes schreiben: w schreibt irgendeinen Wert

Beispiel von zuvor mit konkreten Werten:

$$s_{FSR} = \quad w_1(x = 5)\ \ r_2(x = 5)\ \ w_2(y = 7)\ \ c_2\ \ r_1(y = 7)\ \ w_1(y = 8)\ \ c_1$$
$$w_3(x = 1)\ \ w_3(y = 1)\ \ c_3$$

$$s' = \quad w_1(x = 5)\ \ r_1(y = 0)\ \ w_1(y = 1)\ \ c_1\ \ r_2(x = 5)\ \ w_2(y = 7)\ \ c_2$$
$$w_3(x = 1)\ \ w_3(y = 1)\ \ c_3$$

Also: $s_{FSR} \approx_f s' = t_1\ t_2\ t_3$

Klasse FSR - Plausibilitätstest

Plausibilitätstest: FSR ist nicht ausreichend!

Lost Update

$$L = r_1(x)\ \ r_2(x)\ \ w_1(x)\ \ w_2(x)\ \ c_1\ \ c_2$$

- Mit konkreten Beispielwerten:

- In dem Schedule oben haben wir:

$$L = r_1(x=0)\ \ r_2(x=0)\ \ w_1(x=1)\ \ w_2(x=1)\ \ c_1\ \ c_2$$

- Für die seriellen Schedules haben wir aber:

$$t_1t_2 \equiv r_1(x=0)\ \ w_1(x=1)\ \ c_1\ \ r_2(x=1)\ \ w_2(x=2)\ \ c_2$$

$$t_2t_1 \equiv r_2(x=0)\ \ w_2(x=1)\ \ c_2\ \ r_1(x=1)\ \ w_1(x=2)\ \ c_1$$

- Also $L \notin$ FSR, da t_1t_2 oder t_2t_1 andere Endzustände erzeugen würden.

- OK! Aber was ist mit Inconsistent Read?

Klasse FSR - Plausibilitätstest (2)

Inconsistent Read

- $I = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$
- Mit konkreten Beispielwerten:
- Für Schedule I haben wir:

$$I = r_2(x=0)\ w_2(x=1)\ r_1(x=1)\ r_1(y=0)$$
$$r_2(y=0)\ w_2(y=1)\ c_1\ c_2$$

- Für die seriellen Schedules haben wir:

$$t_2t_1 \equiv r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2$$
$$r_1(x=1)\ r_1(y=1)\ c_1$$
$$t_1t_2 \equiv r_1(x=0)\ r_1(y=0)\ c_1$$
$$r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2$$

- Also $I \in$ FSR , da t_1t_2 oder t_2t_1 denselben Endzustand erzeugen, obwohl t_1 inkonsistente Werte liest. **FSR verhindert also nicht inkonsistentes Lesen.**

Klasse FSR - Abschließende Bermerkungen

- Wie wir gerade gesehen haben genügt die Klasse FSR nicht unseren Anforderungen bzw. der Vermeidung von Inconsistend Read.
- Bzgl. Entscheidbarkeit: Für zwei gegebene Schedules s und s' kann in polynomialer Zeit (in der Länge der beiden Schedules) entschieden werden ob $s \approx_f s'$ gilt.
- Aber es gibt für Schedule s mit n Transaktionen $n!$ verschiedene serielle Schedules. Testen ist nicht einfach (Literatur!).
- Ist aber auch nicht so wichtig, da ja auch bereits die Anforderungen oben nicht erfüllt worden sind und FSR in der Praxis daher nicht relevant.

Klasse VSR (View-Serialisierbarkeit)

Definition: View-Serialisierbarkeit

- Ein Schedule s ist view-serialisierbar, wenn ein serieller Schedule s' existiert, so dass $s \approx_v s'$. VSR bezeichnet die Klasse aller view-serialisierbaren Historien.

View-Serialisierbarkeit

- s erfüllt VSR, wenn eine view-äquivalente serielle Historie erzeugt werden kann und
- die gesamte Historie einen konsistenten DB-Zustand hinterlässt
- View-äquivalent bedeutet, dass alle Leseoperationen Werte liefern wie in einem seriellen Schedule.

- Verhindert inkonsistentes Lesen; da gewährleistet wird, dass die Sicht (View) jeder TA konsistent ist.

Klasse VSR - Beispiel

Ist Historie s_{VSR}, die einen Zyklus enthält, view serialisierbar?

$$s_{VSR} = r_1(x)w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c1\ w_3(x)\ w_3(y)\ c_3$$

Beispiel mit konkreten Werten:

- Annahme wie bisher: Initiale DB $= \{x = 0, y = 0\}$ usw.

$$s_{VSR} = r_1(x = 0)w_1(x = 1)\ w_2(x = 5)\ w_2(y = 7)\ c_2$$
$$w_1(y = 3)\ c1\ w_3(x = 1)\ w_3(y = 1)\ c_3$$

$$s' = r_1(x = 0)\ w_1(x = 1)\ w_1(y = 3)\ c_1\ w_2(x = 5)\ w_2(y = 7)\ c_2$$
$$w_3(x = 1)\ w_3(y = 1)c_3$$

- Also $s_{SVR} \approx_v s' = t_1t_2t_3$

Klasse VSR - Plausibilitätstest

Plausibilitätstest: Ist VSR ausreichend?

Lost Update

$$L = r_1(x)\ \ r_2(x)\ \ w_1(x)\ \ w_2(x)\ \ c_1\ \ c_2$$

Mit konkreten Beispielwerten:

$$L = r_1(x=0)\ \ r_2(x=0)\ \ w_1(x=1)\ \ w_2(x=1)\ \ c_1\ \ c_2$$

- $L \notin$ VSR, da keine view-äquivalente serielle Historie erzeugt werden kann

$$t_1t_2 \equiv r_1(x=0)\ \ w_1(x=1)\ \ c_1\ \ r_2(x=1)\ \ w_2(x=2)\ \ c_2$$
$$t_2t_1 \equiv r_2(x=0)\ \ w_2(x=1)\ \ c_2\ \ r_1(x=1)\ \ w_1(x=2)\ \ c_1$$

Klasse VSR - Plausibilitätstest (2)

Plausibilitätstest: Ist VSR ausreichend?

Inconsistent Read

$$I = r_2(x)\ \ w_2(x)\ \ r_1(x)\ \ r_1(y)\ \ r_2(y)\ \ w_2(y)\ \ c_1\ \ c_2$$

Mit konkreten Beispielwerten:

$$I = r_2(x=0)\ \ w_2(x=1)\ \ r_1(x=1)\ \ r_1(y=0)$$
$$r_2(y=0)\ \ w_2(y=1)\ \ c_1\ \ c_2$$

- $I \notin$ VSR , da **keine** view-äquivalente serielle Historie erzeugt werden kann

$$t_2t_1 \equiv r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2$$
$$r_1(x=1)\ r_1(y=1)\ c_1$$
$$t_1t_2 \equiv r_1(x=0)\ r_1(y=0)\ c_1r_2(x=0)\ w_2(x=1)$$
$$r_2(y=0)\ w_2(y=1)\ c_2$$

Klasse VSR - Eignung

Eignung bzw. Konsistenz

- VSR erfüllt Anforderungen für das Lost-Update-Problem und das Inconsistent-Read-Problem.

Aber:

Theorem zu Komplexität des Entscheidungsproblem

- Das Entscheidungsproblem, ob für einen gegebenen Schedule $s \in$ VSR gilt, ist NP-vollständig.

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

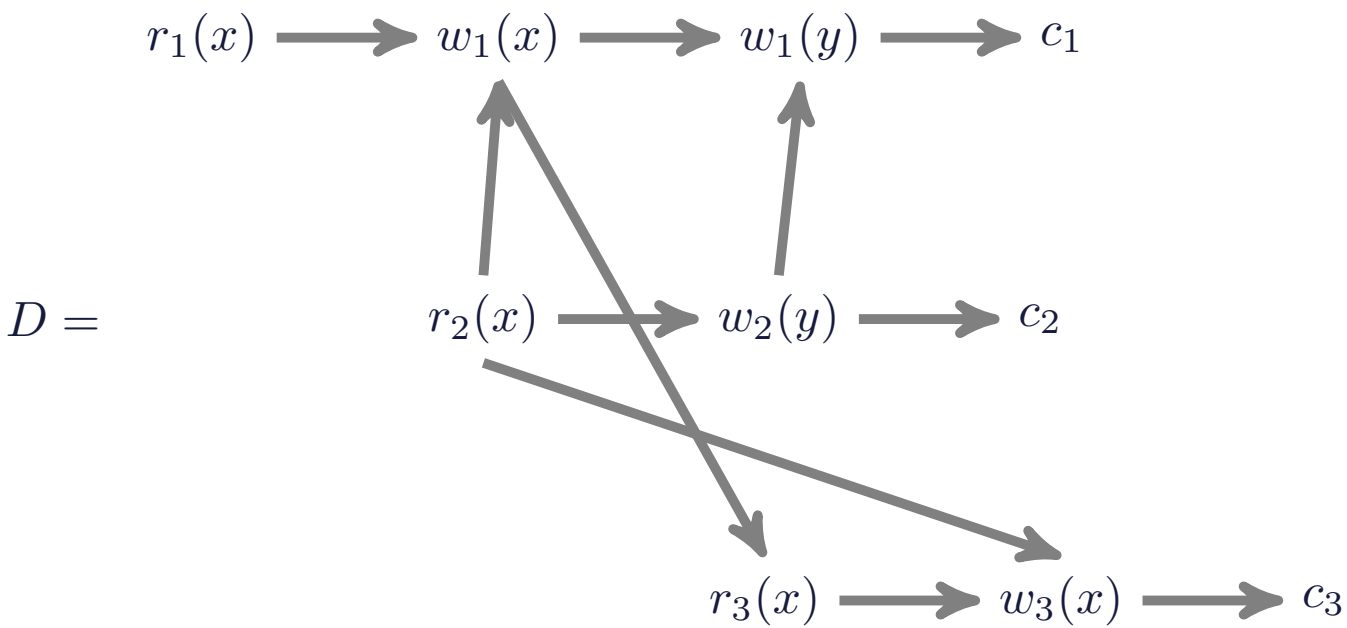
Definition: Konfliktgraph (Serialisierungsgraph)

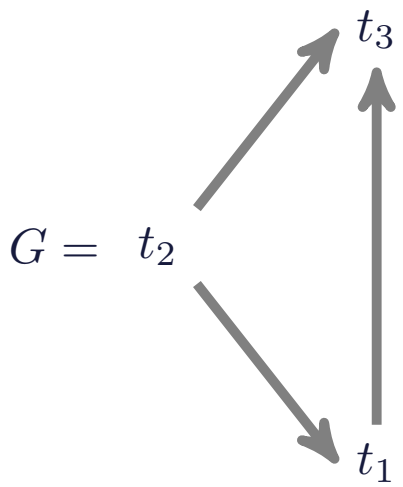
- Sei s ein Schedule. Der Konfliktgraph $G(s) = (V,E)$ ist ein gerichteter Graph mit
  - $V = commit(s)$
  - $(t,t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t)(\exists q \in t')(p,q) \in conf(s)$

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 $w_1(x) \rightarrow r_3(x)$ aus D führt zur Kante $t_1 \rightarrow t_3$ des SG
- weitere Kanten analog

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.

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 sch 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

$$
\begin{array}{lcccccccc}
 & s = & w_1(x) & r_2(x) & w_1(y) & w_1(z) & r_3(z) & w_2(y) & w_3(y) & w_3(z) \\
\text{Regel C2} \rightarrow & & w_1(x) & w_1(y) & r_2(x) & w_1(z) & w_2(y) & r_3(z) & w_3(y) & w_3(z) \\
\text{Regel C2} \rightarrow & & w_1(x) & w_1(y) & w_1(z) & r_2(x) & w_2(y) & r_3(z) & w_3(y) & w_3(z) \\
 & = t_1\ t_2\ t_3 & & & & & & & &
\end{array}
$$

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

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

- 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

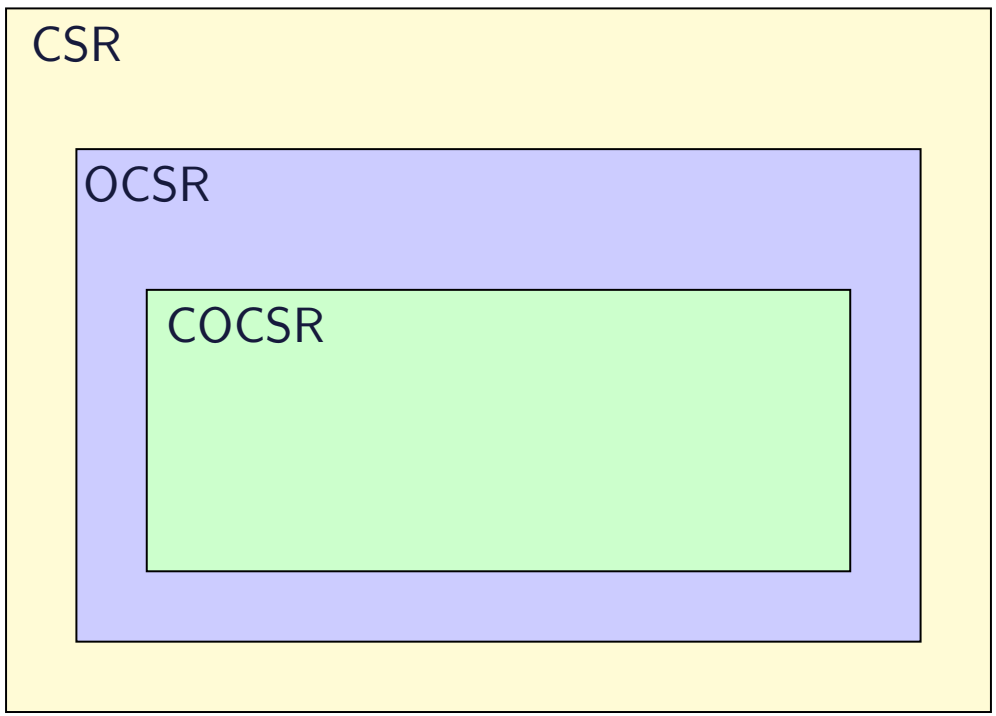
COCSR $\subset$ OCSR

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

- $L = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$
- $conf(L) = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x), w_2(x))\}$
- $L \not\approx_c t_1t_2$ und $L \not\approx_c t_2t_2$

Inconsistent Read

- $I = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)w_2(y)c_1c_2$
- $conf(I) = \{(w_2(x), r_1(x)),(r_1(y), w_2(y))\}$
- $I \not\approx_c t_1t_2$ und $I \not\approx_c t_2t_2$

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 $s \in VSR$