

6. Serialisierbarkeit¹

„Nothing is as practical as a good theory“
– Albert Einstein

- **Anomalien im Mehrbenutzerbetrieb**
- **Synchronisation von Transaktionen**
 - Ablaufpläne, Modellannahmen
 - Korrektheitskriterium: Serialisierbarkeit (bekannt aus Informationssysteme)

Die parallele Ausführung einer Menge von TAs ist **serialisierbar**, wenn es eine serielle Ausführung derselben TA-Menge gibt, die den **gleichen DB-Zustand** und die **gleichen Ausgabewerte** wie die ursprüngliche Ausführung erzielt.

- **Theorie der Serialisierbarkeit**

Sie wird zunächst entwickelt unter Annahme eines fehlerfreien Systems

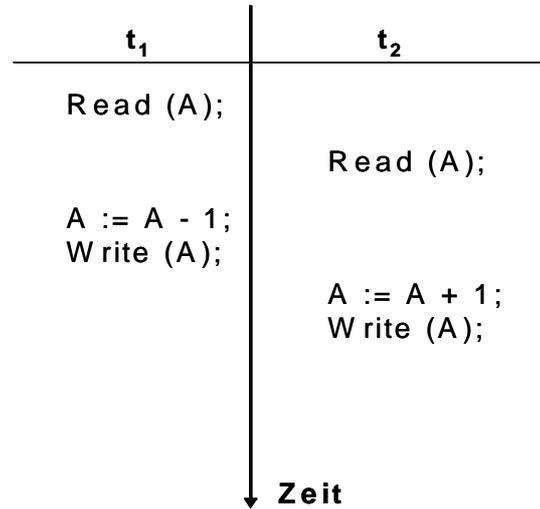
 - Historien und Schedules
 - Korrektheit
 - Klasse CSR
 - Konfliktäquivalenz und Konfliktserialisierbarkeit
 - Serialisierbarkeitstheorem
 - Kommutativitätsregeln
 - Verallgemeinerung des Konfliktbegriffs
 - Klassen OCSR und COCSR
 - Die ganze Wahrheit
- **Anhang**
 - Klassen FSR und VSR (mit Beispielen erklärt)

1. Weikum, G., Vossen, G.: Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann Publishers, 2002

Motivation – Erinnerung

- Anomalien im unkontrollierten Mehrbenutzerbetrieb
- Verlorengegangene Änderung (Lost Update)

ist in jedem Fall auszuschließen



zugehörige Historie: $r_1(A)$ $r_2(A)$ $w_1(A)$ $w_2(A)$ c_1 c_2

- Inkonsistente Analyse (Inconsistent Read, Non-repeatable Read):

Lesetransaktion (Gehaltssumme berechnen)	Änderungstransaktion	DB-Inhalt (Pnr, Gehalt)
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 2345; summe := summe + gehalt;	UPDATE Pers SET Gehalt = Gehalt + 1000 WHERE Pnr = 2345;	2345 39.000 3456 48.000
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 3456; summe := summe + gehalt;	UPDATE Pers SET Gehalt = Gehalt + 2000 WHERE Pnr = 3456;	2345 40.000 3456 50.000

↓ Zeit

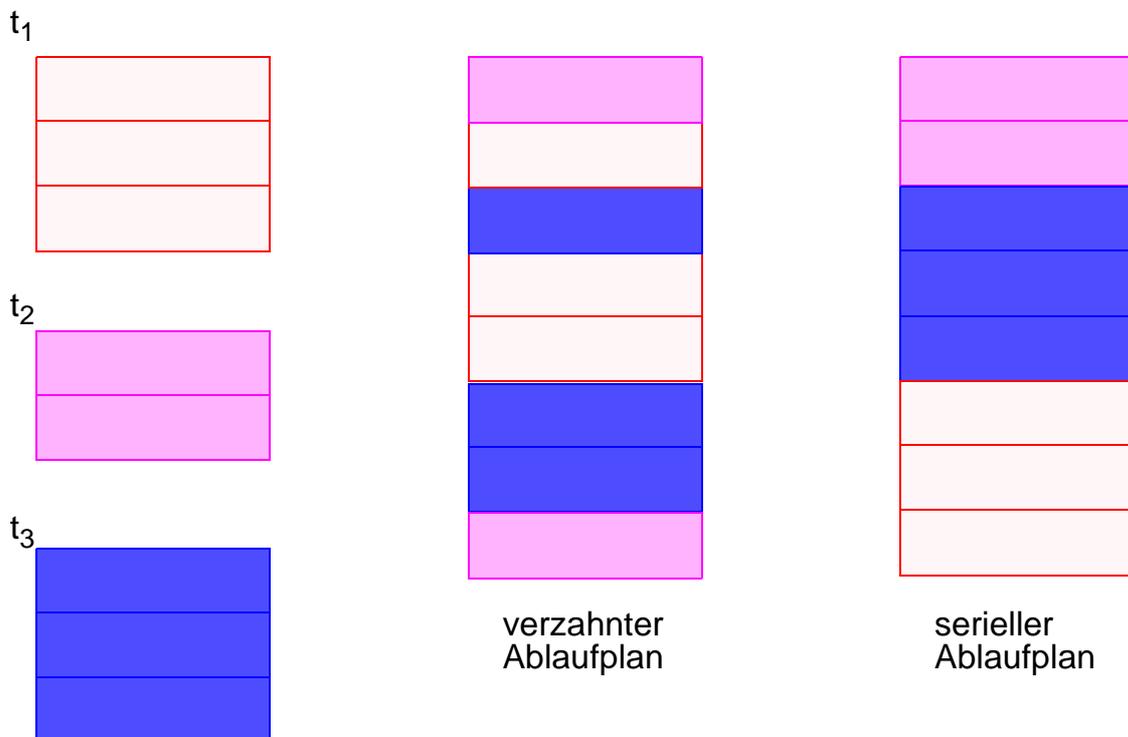
zugehörige Historie: $r_1(A)$ $w_2(A)$ $w_2(B)$ $r_1(B)$...

Synchronisation von Transaktionen

- **Transaktion:** Ein Programm t mit DML-Anweisungen, das folgende Eigenschaft erfüllt:

Wenn t **allein** auf einer konsistenten DB ausgeführt wird, dann terminiert t (irgendwann) und hinterlässt die DB in einem konsistenten Zustand. (Während der TA-Verarbeitung gibt es keine Konsistenzgarantien!)

- **Ablaufpläne für 3 Transaktionen**



➔ Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.

- **Ziel der Synchronisation:**

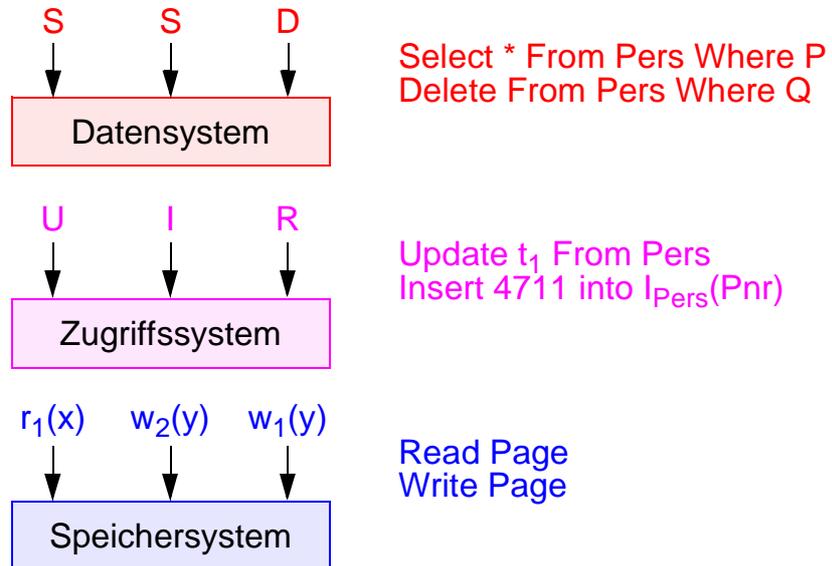
logischer Einbenutzerbetrieb,
d.h. Vermeidung aller Mehrbenutzeranomalien

➔ **Fundamentale Fragestellung:**

Wann ist die parallele Ausführung von n Transaktionen auf gemeinsamen Daten korrekt?

Synchronisation – Modellannahmen

- **Möglichkeiten der Modellbildung für die Synchronisation**



- **Read/Write-Modell (Seitenmodell)**

- DB ist Menge von unteilbaren, uninterpretierten Datenobjekten (z. B. Seiten):
 $D = \{x, y, z, \dots\}$
- DB-Anweisungen lassen sich nachbilden durch atomare Lese- und Schreiboperationen auf Objekten:
 - $r_i(x)$, $w_i(x)$ zum Lesen bzw. Schreiben des Datenobjekts x
 - c_i , a_i zur Durchführung eines **commit** bzw. **abort**
- Jeder Wert, der von einer TA t geschrieben wird, ist potentiell abhängig von allen Datenobjekten, die t vorher gelesen hat!

- **Definition: Transaktion**

Eine Transaktion t ist eine Partialordnung von Schritten der Form $p_i \in \{r(x), w(x)\}$ mit $x \in D$. 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

$$t = p_1 \dots p_n a \quad \text{oder} \quad t = p_1 \dots p_n c$$

Historien und Schedules

- **Definition: Historien und Schedules**

- Es sei $T = \{t_1, \dots, t_n\}$ eine (endliche) Menge von TAs, 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 Operationen $p, q \in op(s)$ von verschiedenen TAs, die auf dasselbe Datenelement 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

- **Erläuterungen zur Definition:**

- Eine Historie (für partiell geordnete TAs)
 - a) enthält alle Operationen aller TAs
 - b) benötigt eine bestimmte Terminierungsoperation für jede TA
 - c) bewahrt alle Ordnungen innerhalb der TA
 - d) hat die Terminierungsoperationen als letzte Operationen in jeder TA
 - e) ordnet Konfliktoperationen
- Wegen (a) und (b) wird eine Historie auch als vollständiger Schedule bezeichnet

Historien und Schedules (2)

- **Bemerkung²**

- 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 TAs t_i und t_j ($i \neq j$) alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.

- **Definitionen: TA-Mengen eines Schedules**

- $\text{trans}(s) := \{t_i \mid s \text{ enthält Schritte von } t_i\}$
- $\text{commit}(s) := \{t_i \in \text{trans}(s) \mid c_i \in s\}$
- $\text{abort}(s) := \{t_i \in \text{trans}(s) \mid a_i \in s\}$
- $\text{active}(s) := \text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s))$

2. Der Begriff Historie bezeichnet eine retrospektive Sichtweise, also einen abgeschlossenen Vorgang. Ein Scheduling-Algorithmus (Scheduler) produziert Schedules, wodurch noch nicht abgeschlossene Vorgänge bezeichnet werden. Manche Autoren machen jedoch keinen Unterschied zwischen Historie und Schedule.

Historien und Schedules (3)

- **Beispiel**

- $s_1 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) r_1(y) w_1(y) w_2(z) w_3(z) c_1 c_2 a_3$

$\text{trans}(s_1) = \{t_1, t_2, t_3\}$

$\text{commit}(s_1) = \{t_1, t_2\}$

$\text{abort}(s_1) = \{t_3\}$

$\text{active}(s_1) = \emptyset$

- $s_2 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) w_1(y) w_2(z) w_3(z) c_1$

$\text{trans}(s_2) = \{t_1, t_2, t_3\}$

$\text{commit}(s_2) = \{t_1\}$

$\text{abort}(s_2) = \emptyset$

$\text{active}(s_2) = \{t_2, t_3\}$

- **Für jede Historie s gilt:**

- $\text{trans}(s) = \text{commit}(s) \cup \text{abort}(s)$

- $\text{active}(s) = \emptyset$

- **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,

$s' = \prod_T(s)$ mit $\text{op}(s') = \text{op}(s) - \cup_{t \notin T} \text{op}(t)$,

in E für jedes $T \subseteq \text{trans}(s)$

- Mit anderen Worten, E ist unter beliebigen Projektionen abgeschlossen

- **Monotonizität**

Monotonizität einer Historienklasse E ist eine wünschenswerte Eigenschaft,

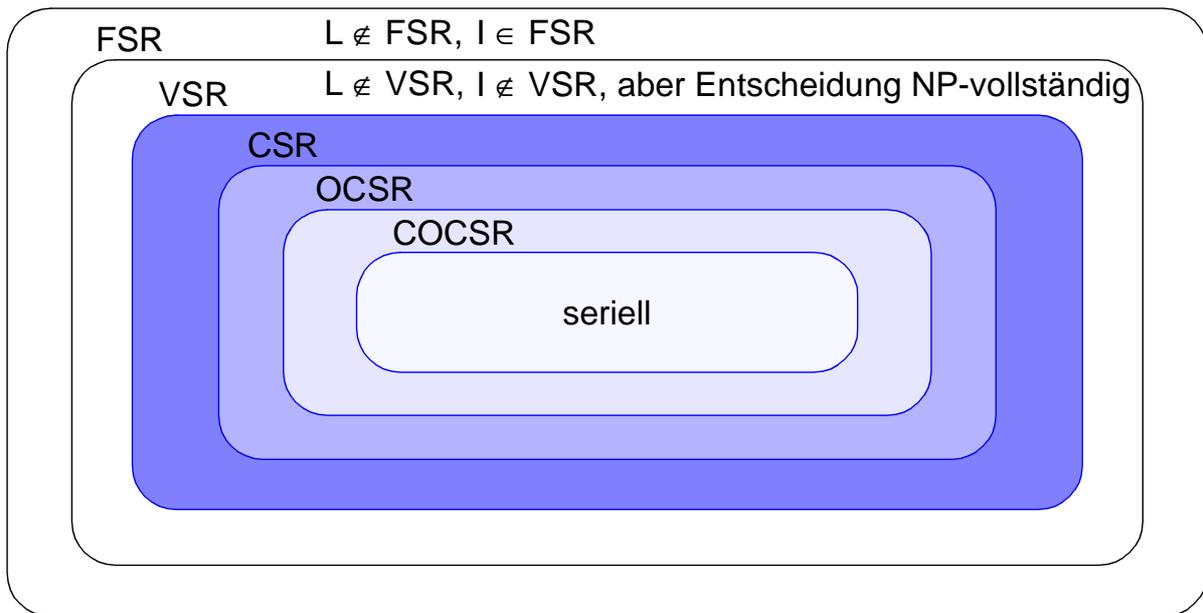
da sie E unter beliebigen Projektionen bewahrt!

Serialisierbarkeitsklassen

- Ziel dieses Kapitels

- detailliertere und
- formale Betrachtung des Serialisierbarkeitsbegriffs

- Klassen (vereinfachter Ausblick)



- Akzeptable Klasse von Schedules muss

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

$$D = r_1(x) \ w_1(x) \ r_2(x) \ a_1 \ w_2(x) \ c_2$$

- Deshalb: Konzentration auf Konflikt-Serialisierbarkeit (CSR)

➔ **CSR ist wichtigste Art der Serialisierbarkeit für die praktische Nutzung**

Korrektheit

- Ein Korrektheitskriterium kann formal betrachtet werden als Abbildung

- $\sigma: S \rightarrow \{0, 1\}$ mit S Menge aller Schedules.
- $\text{correct}(S) := \{s \in S \mid \sigma(s) = 1\}$

- Ein konkretes Korrektheitskriterium sollte mindestens die folgenden Anforderungen erfüllen

1. $\text{correct}(S) \neq \emptyset$
2. „ $s \in \text{correct}(S)$ “ ist **effizient** entscheidbar
3. $\text{correct}(S)$ ist „ausreichend groß“,
 - so dass der Scheduler viele Möglichkeiten hat, korrekte Schedules herbeizuführen
 - **Je größer die Menge der erlaubten Schedules, desto höher die Nebenläufigkeit, desto höher die Effizienz!**

- Fundamentale Idee der Serialisierbarkeit

- Einzelne TA ist korrekt, da sie die Datenbank konsistent erhält
- Konsequenz: serielle Historien sind korrekt!
- Serielle Historien sollen jedoch „**nur**“ als **Korrektheitsmaß** via geeignet gewählten Äquivalenzrelationen nutzbar gemacht werden

- Vorgehensweise

1. Definition einer Äquivalenzrelation „ \approx “ auf S (Menge aller Schedules), so dass
$$[S]_{\approx} = \{[s]_{\approx} \mid s \in S\} \quad (\text{Menge der Äquivalenzklassen})$$
2. Betrachten solcher Äquivalenzklassen und Definition der Serialisierbarkeit ihrer Vertreter über die Äquivalenz zu seriellen Historien

➔ **Aborts werden hier nicht betrachtet (nur commit(s) und active(s))!**

Klasse CSR

- **Ziel**

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

- **Definition: Konflikte und Konfliktrelationen**

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

- **Bemerkung**

Konflikte bestehen nur zwischen Datenoperationen, unabhängig vom Terminierungsstatus der TA; **Operationen von abgebrochenen TAs** können dennoch **ignoriert** werden

- **Beispiel**

- $s = w_1(x) \ r_2(x) \ w_2(y) \ r_1(y) \ w_1(y) \ w_3(x) \ w_3(y) \ c_1 \ a_2$
- $\text{conf}(s) = \{(w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}$

Klasse CSR (2)

- **Definition: Konfliktäquivalenz**

Schedules s und s' sind konfliktäquivalent, ausgedrückt durch $s \approx_c s'$, wenn

- $op(s) = op(s')$
- $conf(s) = conf(s')$

- **Beispiel ($s \approx_c s'$)**

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

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

➔ $conf(s) = \{(r_1(x), w_2(x)), (r_1(y), w_2(y)), (w_2(x), w_1(x)), (w_1(y), w_2(y))\} = conf(s')$

- **Konfliktschritte-Graph $D(s)$**

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

- **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**

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

$r_2(x) \dashrightarrow w_1(x) \quad r_1(z) \dashrightarrow w_3(z)$

$w_2(y) \dashrightarrow w_3(y)$

$s_1 \in \text{CSR}$

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

$w_2(x) \dashrightarrow r_1(x)$

$w_2(y) \dashleftarrow r_1(y)$

$s_2 \notin \text{CSR}$

Klasse CSR (3)

- **Definition: Konfliktgraph (Serialisierungsgraph)**

Sei s ein Schedule. Der Konfliktgraph $G(s) = (V, E)$ ist ein gerichteter Graph mit

- $V = \text{commit}(s)$
- $(t, t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t) (\exists q \in t') (p, q) \in \text{conf}(s)$

- **Anmerkung**

Konfliktgraph abstrahiert von individuellen Konflikten zwischen Paaren von TAs ($\text{conf}(s)$) und repräsentiert mehrfache Konflikte zwischen denselben (abgeschlossenen) TAs durch eine einzige Kante

- **Beispiel**

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

- **Serialisierbarkeitstheorem**

Sei s eine Historie; dann gilt: $s \in \text{CSR}$ gdw $G(s)$ azyklisch

- **Aufgabe**

- Finden einer seriellen Historie, die konsistent mit allen Kanten in $G(s)$ ist

Klasse CSR (4)

- **Beispiel**

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

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

- $G(s') =$

- **Korollar**

Mitgliedschaft in CSR lässt sich in polynomialer Zeit in der Menge der am betreffenden Schedule teilnehmenden TAs testen

Klasse CSR (5)

- **Blindes Schreiben**

- Ein blindes Schreiben eines Datenelements x liegt vor, wenn eine TA ein $\text{Write}(x)$ ohne ein vorhergehendes $\text{Read}(x)$ durchführt

- Wenn wir blindes Schreiben für TAs verbieten, verschärft sich die Definition einer TA um die Bedingung:

Wenn $w_i(x) \in T_i$, dann gilt $r_i(x) \in T_i$ und $r_i(x) < w_i(x)$

- Dann gilt:

Eine Historie ist view-serialisierbar gdw sie konfliktserialisierbar ist!

- **Konflikte und Kommutativität**

- bisher wurde Konfliktserialisierbarkeit über den Konfliktgraph G definiert

- **Ziel**

- 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 $\text{op}(s) = \text{op}(s')$ 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 (6)

• 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

• Beispiel

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

• Theorem

s und s' seien Schedules mit $op(s) = op(s')$;
dann gilt $s \approx_c s'$ gdw $s \sim^* s'$

Klasse CSR (7)

- **Definition: Kommutativitätsbasierte Reduzierbarkeit**

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

- **Korollar**

Eine Historie s ist kommutativitätsbasiert reduzierbar gdw $s \in \text{CSR}$

- **Verallgemeinerung des Konfliktbegriffs**

- Scheduler muss nicht die Art der Operationen kennen,
sondern nur wissen, welche Schritte in Konflikt stehen

- Beispiel

$s = p_1 q_1 p_2 o_1 p_3 q_2 o_2 o_3 p_4 o_4 q_3$

mit den Konfliktschritten (q_1, p_2) , (p_2, o_1) , (q_1, o_2) und (o_4, q_3)

- nutzbar für semantische Synchronisation

- Spezifikation einer Kommutativitäts- bzw. Konflikttabelle für „neue“
(mglw. anwendungsspezifische) Operationen und

- Ableitung der Konfliktserialisierbarkeit von dieser Tabelle

- Beispiele für Operationen

- increment/decrement

- enqueue/dequeue

- ...

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**

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

- $G(s_{312}) =$

↳ Kontrast zwischen Serialisierungs- und tatsächlicher Ausführungsreihenfolge möglicherweise unerwünscht!

↳ Situation lässt sich durch Ordnungserhaltung vermeiden

Klasse OCSR (2)

- **Definition: 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 $t_i, t_j \in \text{trans}(s)$ 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: $OCSR \subset CSR$

- **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 Inklusionsbeziehung echt ist:
 $s_{312} \in CSR - OCSR$

- **Weitere Einschränkung von CSR**

- nützlich für verteilte und möglicherweise heterogene Anwendungen
- Beobachtung: Für Konflikt-Serialisierbarkeit ist es hinreichend, wenn in Konflikt stehende TAs ihr Commit in Konfliktreihenfolge ausführen

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 $t_i, t_j \in \text{commit}(s)$, $i \neq j$:

Wenn $(p, q) \in \text{conf}(s)$ für $p \in t_i$, $q \in t_j$, 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 $\text{COCSR} \subset \text{CSR}$

- **Beweisskizze**

- $s = r_1(x) \ w_2(x) \ c_2 \ c_1$

- $s \in \text{CSR} - \text{COCSR}$ (die Inklusion ist also echt)

- **Theorem**

Sei s eine Historie: $s \in \text{COCSR}$ gdw

$s \in \text{CSR}$ und es existiert eine serielle Historie s' ,

so dass $s' \approx_c s$ und für alle $t_i, t_j \in \text{trans}(s)$, $t_i <_{s'} t_j \Rightarrow c_{t_i} <_s c_{t_j}$

- **Theorem: COCSR \subset OCSR**

Die ganze Wahrheit

- Definition: Commit-Serialisierbarkeit**

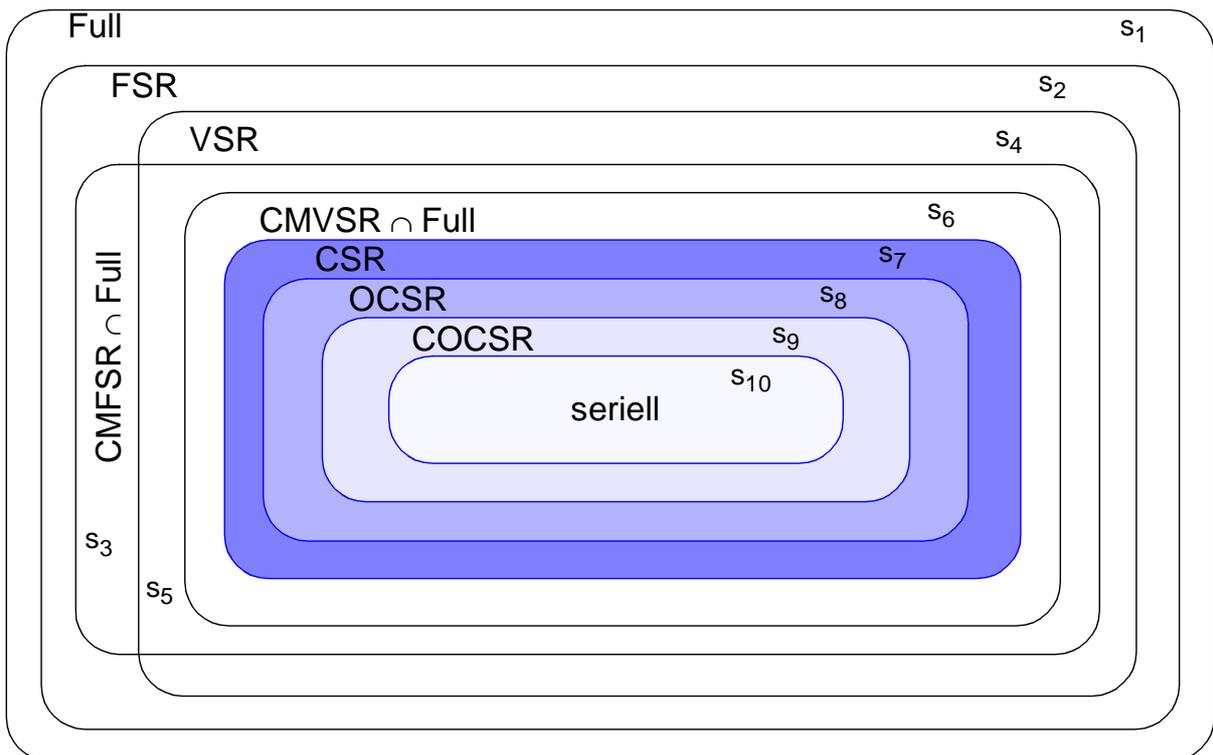
Ein Schedule s heißt commit-serialisierbar, wenn $CP(s)$ serialisierbar ist für jeden Präfix s von s .

(CP: Präfix-Commit-Abgeschlossenheit)

- Klassen commit-serialisierbarer Schedules**

- CMFSR : commit final state serializable histories
- CMVSR: commit view serializable histories
- CMCSR: commit conflict serializable histories

- Alle Klassen im Überblick**



$s_7 = w_1(x) w_2(x) w_2(y) c_2 w_1(z) c_1$

$s_8 = w_3(y) c_3 w_1(x) r_2(x) c_2 w_1(y) c_1$

$s_9 = w_3(y) c_3 w_1(x) r_2(x) w_1(y) c_1 c_2$

$s_{10} = w_1(x) w_1(y) c_1 w_2(x) w_2(y) c_2$

Zusammenfassung

- Beim **ungeschützten und konkurrierenden Zugriff** von **Lesern und Schreibern** auf **gemeinsame Daten** können **Anomalien** auftreten
- **Korrektheitskriterium der Synchronisation: Serialisierbarkeit**
(gleicher DB-Zustand, gleiche Ausgabewerte wie bei seriellem Ablaufplan)
- **Theorie der Serialisierbarkeit**
 - FSR erfüllt nicht einmal Minimalbedingungen
 - VSR ist nicht monoton und Testen der VSR-Mitgliedschaft ist NP-vollständig!
 - Im Gegensatz zur Final-State-Serialisierbarkeit und View-Serialisierbarkeit 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 DB-Daten**, wenn diese Operationen **nicht reihenfolgeunabhängig** sind!
 - **Serialisierbarkeitstheorem:**
Sei s eine Historie; dann gilt: $s \in CSR$ gdw $G(s)$ azyklisch
 - Verschärfung des Serialisierbarkeitsbegriffs durch OCSR und COCSR
- **Achtung: Bisher wurde der Fehlerfall ausgeschlossen**
 - Praktische Anwendungen erfordern deshalb weitere Einschränkungen
 - Schedules müssen „recoverable“ (RC) sein und die Eigenschaft „avoiding cascading aborts“ (ACA) besitzen
- **Serialisierbare Abläufe**
 - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
 - Anzahl der möglichen Historien (Schedules) bestimmt erreichbaren Grad an Parallelität

Klasse FSR

- **Definition: Final-State-Serialisierbarkeit³**

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

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

- **Beispiele mit konkreten Werten**

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

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

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

➔ $S_{FSR} \approx_f s' = t_1 \ t_2 \ t_3$

3. Beachte: „ \approx_f “ und „ \approx_v “ sind hier nicht definiert! Die Definition der Final-State- und View-Äquivalenz erfordert eine komplexe Einführung der Herbrand-Semantik und wird deshalb hier weggelassen

Klasse FSR (2)

- **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:

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

- $L \notin \text{FSR}$,
da $t_1 \ t_2$ oder $t_2 \ t_1$ andere Endzustände erzeugen würden

$$t_1 \ t_2 \equiv r_1(x = 0) \ w_1(x = 1) \ c_1 \ r_2(x = 1) \ w_2(x = 2) \ c_2$$

$$t_2 \ t_1 \equiv r_2(x = 0) \ w_2(x = 1) \ c_2 \ r_1(x = 1) \ w_1(x = 2) \ c_1$$

- **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

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

- $I \in \text{FSR}$,
da $t_1 \ t_2$ oder $t_2 \ t_1$ denselben Endzustand erzeugen, obwohl t_1 inkonsistente Werte liest. Final-State-Serialisierbarkeit verhindert also nicht

inkonsistentes Lesen

$$t_2 \ t_1 \equiv r_2(x = 0) \ w_2(x = 1) \ r_2(y = 0) \ w_2(y = 1) \ c_2 \ r_1(x = 1) \ r_1(y = 1) \ c_1$$

$$t_1 \ t_2 \equiv r_1(x = 0) \ r_1(y = 0) \ c_1 \ r_2(x = 0) \ w_2(x = 1) \ r_2(y = 0) \ w_2(y = 1) \ c_1$$

Klasse VSR

- **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
- Neues Konzept der View-Serialisierbarkeit verhindert **inkonsistentes Lesen**; sie gewährleistet, dass die Sicht jeder TA konsistent ist
- Ist Historie s_{VSR} , die einen Zyklus enthält, view-serialisierbar?

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

„View-äquivalent“ bedeutet, dass alle Leseoperationen Werte liefern wie in einem seriellen Schedule

- **Beispiel mit konkreten Werten**

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

$$s_{VSR} = r_1(x = 0) \quad w_1(x = 1) \quad w_2(x = 5) \quad w_2(y = 7) \quad c_2 \quad w_1(y = 3) \quad c_1 \\ w_3(x = 1) \quad w_3(y = 1) \quad c_3$$

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

→ $s_{VSR} \approx_v s' = t_1 \quad t_2 \quad t_3$

Klasse VSR (2)

- **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:

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

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

$$t_1 \ t_2 \equiv r_1(x=0) \ w_1(x=1) \ c_1 \ r_2(x=1) \ w_2(x=2) \ c_2$$

$$t_2 \ t_1 \equiv r_2(x=0) \ w_2(x=1) \ c_2 \ r_1(x=1) \ w_1(x=2) \ c_1$$

- **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:

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

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

$$t_2 \ t_1 \equiv r_2(x=0) \ w_2(x=1) \ r_2(y=0) \ w_2(y=1) \ c_2 \ r_1(x=1) \ r_1(y=1) \ c_1$$

$$t_1 \ t_2 \equiv r_1(x=0) \ r_1(y=0) \ c_1 \ r_2(x=0) \ w_2(x=1) \ r_2(y=0) \ w_2(y=1) \ c_1$$

- VSR bestätigt unsere Erwartung: **konsistente Sicht jeder TA.**

➔ Neben der Recovery ist für VSR aber auch Komplexität zu berücksichtigen!

- **Theorem**

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

CSR \subset VSR

- **Lost Update**

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

- **Inconsistent Read**

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

- **Theorem: CSR \subset VSR**

- **Korollar: CSR \subset VSR \subset FSR**

- **Beispiel**

- $s_{\text{VSR}} = r_1(x) \ w_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \ r_3(x) \ r_3(y) \ c_3$
- $s \not\equiv_c t_1 t_2 t_3$ und $s \notin \text{CSR}$, aber
 $s \approx_v t_1 t_2 t_3$ und damit $s \in \text{VSR}$

- **Theorem**

- CSR ist monoton
- $s \in \text{CSR} \Leftrightarrow \Pi_T(s) \in \text{VSR}$ für alle $T \in \text{trans}(s)$
(d.h., CSR ist die größte monotone Teilmenge von VSR)