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://www.wlgis.informatik.uni-kl.de/cms/courses/datenbankanwendung/

Datenbank-anwendung

6. Serialisierbarkeit[1]

"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.

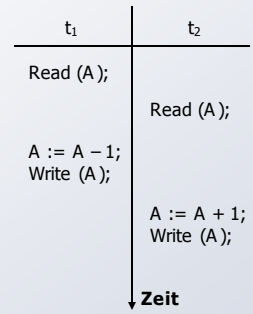
- **Theorie der Serialisierbarkeit**
  - Historien und Schedules
  - Korrektheit
  - Klasse CSR
    - Konfliktäquivalenz und Konfliktserialisierbarkeit
    - Serialisierbarkeitstheorem
    - Kommutativitätsregeln
    - Verallgemeinerung des Konfliktbegriffs
  - Klassen OCSR und COCSR
  - Die ganze Wahrheit
- **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

Motivation – Erinnerung (2)

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

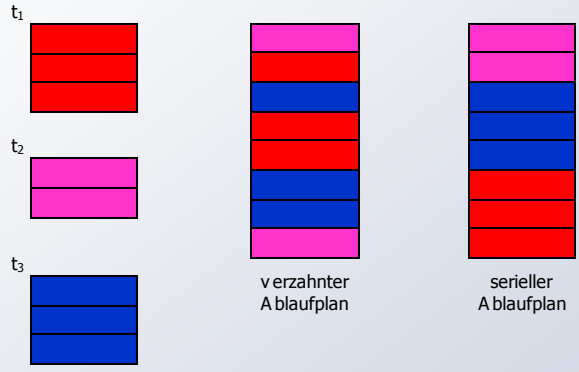
Lesetransaktion (Gehaltssumme berechnen)	Änderungstransaktion	DB-Inhalt (Pnr, Gehalt)
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 2345; summe := summe + gehalt;		2345 39.000 3456 48.000
	UPDATE Pers SET Gehalt = Gehalt + 1000 WHERE Pnr = 2345;	2345 40.000
	UPDATE Pers SET Gehalt = Gehalt + 2000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

Synchronisation von Transaktionen (2)

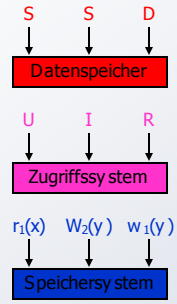
- **Ziel der Synchronisation:**

  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**



Select * From Pers Where P
Delete From Pers Where Q

Select t_i From Pers
Insert 4711 into I_{Pers}(Pnr)

Read Page
Write Page

Synchronisation – Modellannahmen (2)

- **Read/Write-Modell (Seitenmodell)**
  - DB ist Menge von unteilbaren, uninterpretierten Datenobjekten (z. B. Seiten): D = {x, y, z, ...}
  - DB-Anweisungen der Transaktion i 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 potenziell abhängig von allen Datenobjekten, die t vorher gelesen hat!

Synchronisation – Modellannahmen (3)

- **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

    $t = p_1 \ldots p_n\ a$ oder $t = p_1 \ldots p_n\ c$

Historien und Schedules

- **Definition: *Historien und Schedules***
  - Es sei $T = \{t_1, \ldots, t_n\}$ 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 $(1 \le i \le n)$ 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) $(\forall i, 1 \le i \le n)\ c_i \in op(s) \Leftrightarrow a_i \notin op(s)$

    c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$

    d) $(\forall i, 1 \le i \le n)\ (\forall\ p \in op_i)\ p <_s a_i$ 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

Historien und Schedules

- **Definition: *Historien und Schedules***
  - Es sei $T = \{t_1, \ldots, t_n\}$ 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 $(1 \le i \le n)$ beschreibt
  - 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) $(\forall i, 1 \le i \le n)\ c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
    c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$
    d) $(\forall i, 1 \le i \le n)\ (\forall p \in op_i)\ p <_s a_i$ 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

enthält alle Operationen aller TAs

benötigt eine Terminierungs-operation für jede TA

bewahrt alle Ordnungen innerhalb der TA

Terminierungsop. ist letzte Operation in jeder TA

ordnet Konflikt-operationen

Historien und Schedules (2)

- **Bemerkung[2]**
  - 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 TAs t_i und t_j $(i \ne j)$ alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.
- **Definitionen: TA-Mengen eines Schedules**
  - trans(s) := $\{t_i \mid s \text{ enthält Schritte von } t_i\}$
  - commit(s) := $\{t_i \in trans(s) \mid c_i \in s\}$
  - abort(s) := $\{t_i \in trans(s) \mid a_i \in s\}$
  - active(s) := $trans(s) - (commit(s) \cup abort(s))$
- **Für jede Historie s gilt:**
  - $trans(s) = commit(s) \cup abort(s)$
  - $active(s) = \varnothing$

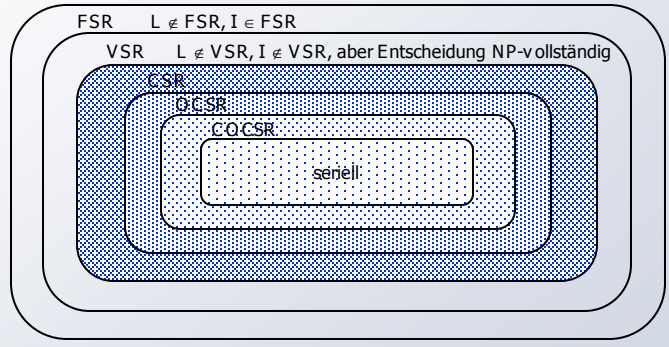
2. Der Begriff Historie bezeichnet eine retrospektive Sichtweise, also einen abgeschlossenen Vorgang. Ein Scheduling Algorithmus (Scheduler) produziert Schedules, wodurch noch nicht abgeschlossene Vorgänge bezeichnet werden. Manche Autoren machen jedoch keinen Unterschied zwischen Historie und Schedule.

Historien und Schedules (3)

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



Serialisierbarkeitsklassen (2)

- **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**

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 Schedulern eignet
- **Definition: Konflikte und Konfliktrelationen**
  - Sei s ein Schedule; t, t' ∈ trans(s), t ≠ 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
  - conf(s) := {(p, q) | p, q sind in Konflikt in s 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**
  - $s = w_1(x)\ \ r_2(x)\ \ w_2(y)\ \ r_1(y)\ \ w_1(y)\ \ w_3(x)\ \ w_3(y)\ \ c_1\ \ a_2$
  - conf(s) =

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) = conf (s') ?

Klasse CSR (3)

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

Klasse CSR (4)

- **Definition: 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: 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**

  Konfliktgraph abstrahiert von individuellen Konflikten zwischen Paaren von TAs (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) =

Klasse CSR (5)

- **Serialisierbarkeitstheorem**

  Sei s eine Historie; dann gilt: $s \in CSR$ gdw G(s) azyklisch
- **Beispiele**
  - $s = r_1(y)\ \ r_3(w)\ \ r_2(y)\ \ w_1(y)\ \ w_1(x)\ \ w_2(x)\ \ w_2(z)\ \ w_3(x)\ \ c_1\ \ c_3\ \ c_2$

    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 (6)

- **Blindes Schreiben**
  - Ein blindes Schreiben eines Datenelements x liegt vor, wenn eine TA ein Write(x) ohne ein vorhergehendes 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!

Klasse CSR (7)

- **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 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 (8)

- **Kommutativitätsregeln**
  - ~ 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**

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

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

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

$= t_1\ t_2\ t_3$

Klasse CSR (9)

- **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 $s \approx_c s'$ gdw $s \sim^* s'$
- **Korollar**
  Eine Historie s ist kommutativitätsbasiert reduzierbar gdw $s \in CSR$

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

⇨ 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 $t_i, t_j \in trans(s)$ 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, $t_j \in$ commit(s), $i \neq j$:
  Wenn $(p, q) \in$ 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 COCSR $\subset$ CSR

- **Beweisskizze**
  - $s = r_1(x) \quad w_2(x) \quad c_2 \quad c_1$
  - $s \in$ 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), $t_i <_s t_j \Rightarrow c_{ti} <_s c_{tj}$

- **Theorem:** COCSR $\subset$ OCSR

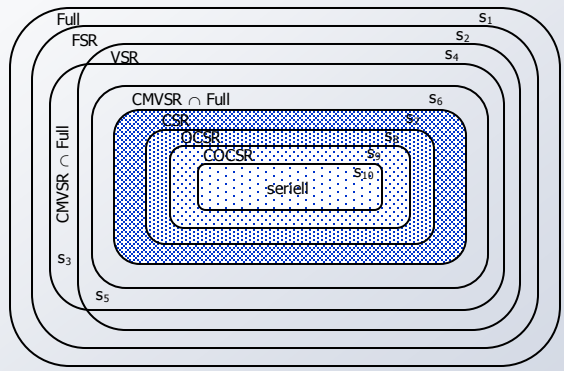
Die ganze Wahrheit

- **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**



s7 = w1(x) w2(x) w2(y) c2 w1(z) c1
s8 = w3(y) c3 w1(x) r2(x) c2 w1(y) c1
s9 = w3(y) c3 w1(x) r2(x) w1(y) c1 c2

s10 = w1(x) w1(y) c1 w2(x) w2(y) c2

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

Zusammenfassung (2)

- **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-Serialisierbarkeit**[3]
  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 „$\approx$" 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:

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

  - $L \notin FSR$,
    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

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

  - $I \in FSR$,
    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
  - „View-äquivalent" bedeutet, dass alle Leseoperationen Werte liefern wie in einem seriellen Schedule

  - 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?

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

- **Beispiel mit konkreten Werten**
  - Annahmen wie bisher: initiale DB = {x = 0, y = 0} usw.

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

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

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

Datenbank-anwendung

Anomalien

Modellannahmen

CSR

OCSR

COCSR

FSR/VSR

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:
  $L = r_1(x = 0)\ r_2(x = 0)\ w_1(x = 1)\ w_2(x = 1)\ c_1\ c_2$
  - $L \notin VSR$, 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:
  $I = r_2(x = 0)\ w_2(x = 1)\ r_1(x = 1)\ r_1(y = 0)\ r_2(y = 0)\ w_2(y = 1)\ c_1\ c_2$
  - $I \notin VSR$, 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 $s \in VSR$ gilt, ist NP-vollständig

CSR ⊂ VSR

- **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 $L \not\approx_c t_2\ t_1$
- **Inconsistent Read**
  - $I = r_2(x)\ \ w_2(x)\ \ r_1(x)\ \ r_1(y)\ \ r_2(y)\ \ w_2(y)\ \ c_1\ \ c_2$
  - $conf(I) = \{(w_2(x), r_1(x)), (r_1(y), w_2(y))\}$
  - $I \not\approx_c t_1\ t_2$ und $I \not\approx_c t_2\ t_1$
- **Theorem:** CSR ⊂ VSR
- **Korollar:** CSR ⊂ VSR ⊂ 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 $s \in VSR$
- **Theorem**
  - CSR ist monoton
  - $s \in CSR \Leftrightarrow PT(s) \in VSR$ für alle $T \in trans(s)$ (d.h., CSR ist die größte monotone Teilmenge von VSR)