DIS‘2010

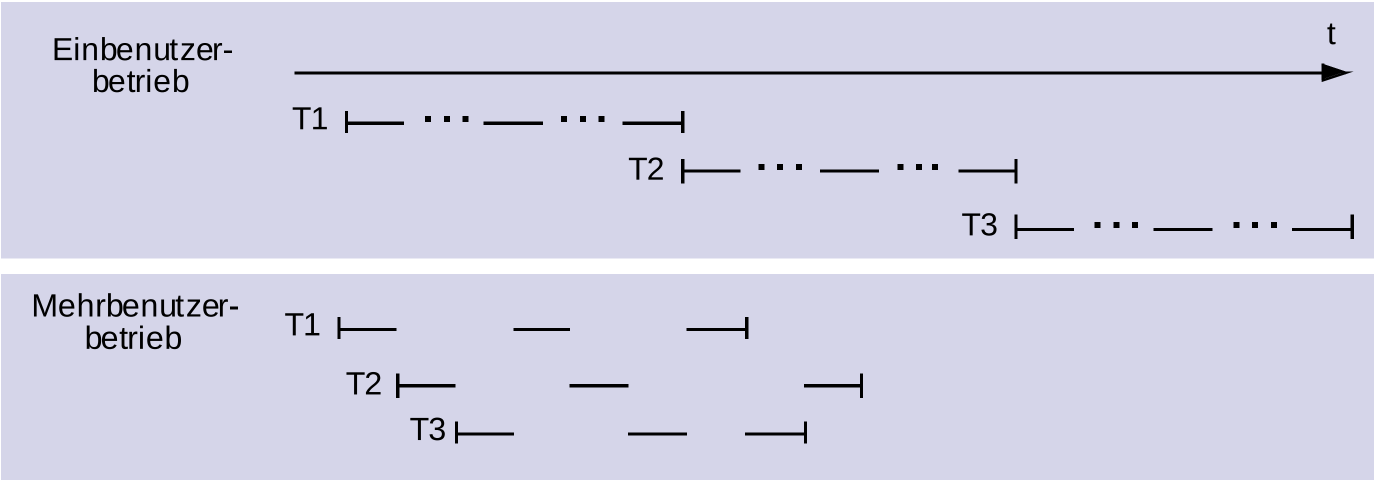
-

4. Synchronisation - Korrektheit

Norbert Ritter
Datenbanken und Informationssysteme
vsis-www.informatik.uni-hamburg.de

Motivation - Erinnerung (1)

- Einbenutzer-/Mehrbenutzerbetrieb



  - CPU-Nutzung während TA-Unterbrechungen
    - E/A
    - Denkzeiten bei Mehrschritt-TA
    - Kommunikationsvorgänge in verteilten Systemen
  - bei langen TAs zu große Wartezeiten für andere TA (Scheduling-Fairness)

Motivation - Erinnerung (2)

- Anomalien im unkontrollierten Mehrbenutzerbetrieb
  1. Abhängigkeit von nicht-freigegebenen Änderungen (Dirty-Read)
     - Geänderte, aber noch nicht freigegebene Daten werden als „schmutzig“ bezeichnet (dirty data), da die TA ihre Änderungen bis Commit (einseitig) zurücknehmen kann
     - Schmutzige Daten dürfen von anderen TAs nicht in „kritischen” Operationen benutzt werden

T1	T2
Read (A);	
A := A + 100;	
Write (A)	
	Read (A);
	Read (B);
	B := B + A;
	Write (B);
	Commit;
Abort;	

Zeit

Motivation - Erinnerung (3)

- Anomalien im unkontrollierten Mehrbenutzerbetrieb
  2. Verlorengegangene Änderung (Lost Update)
     - ist in jedem Fall auszuschließen

T1	**T2**
Read (A);	
	Read (A);
A := A - 1; Write (A);	
	A := A + 1; Write (A);

Zeit

Motivation - Erinnerung (4)

- Anomalien im unkontrollierten Mehrbenutzerbetrieb
  3. Inkonsistente Analyse (Non-repeatable Read)

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

Motivation - Erinnerung (5)

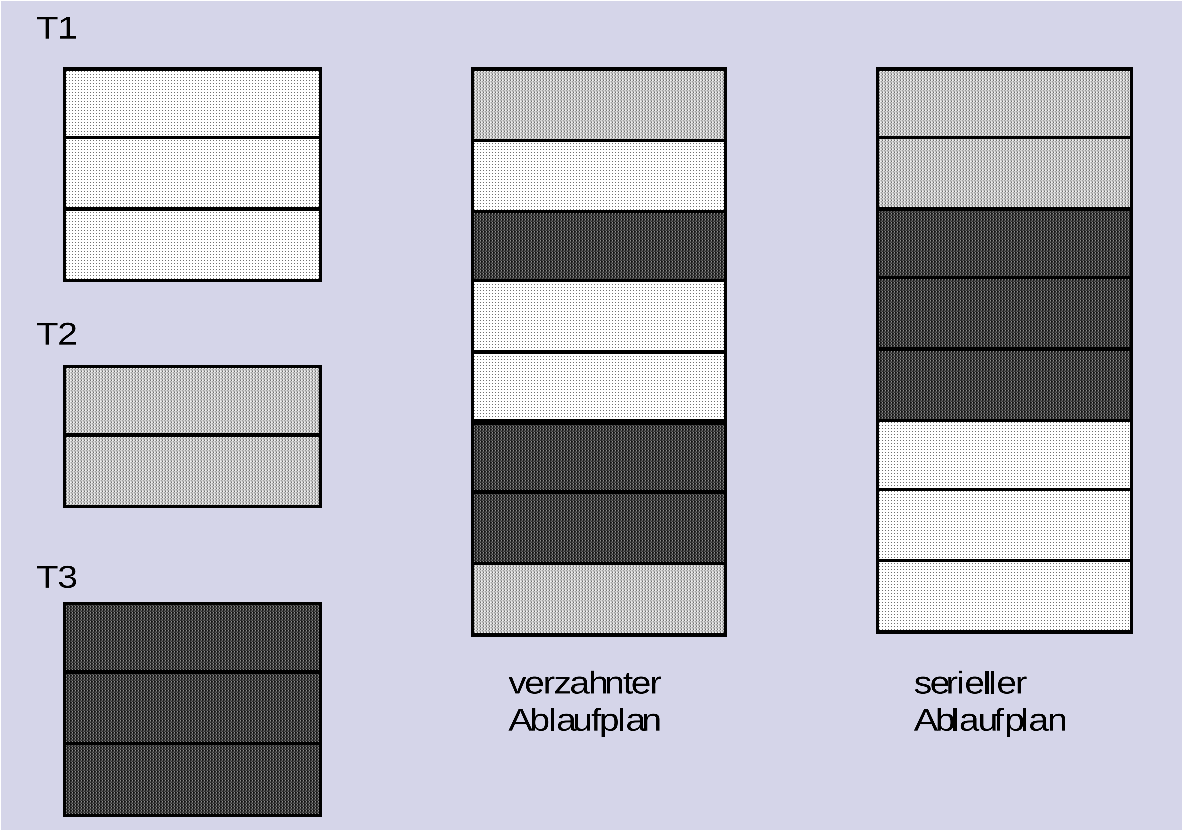
- Anomalien im unkontrollierten Mehrbenutzerbetrieb
  4. Phantom-Problem

Lesetransaktion (Gehaltssumme überprüfen)	**Änderungstransaktion** (Einfügen eines neuen Angestellten)
SELECT SUM (Gehalt) INTO :summe FROM Pers WHERE Anr = 17;	
	INSERT INTO Pers (Pnr, Anr, Gehalt) VALUES (4567, 17, 55.000);
	UPDATE Abt SET Gehaltssumme = Gehaltssumme + 55.000 WHERE Anr = 17;
SELECT Gehaltssumme INTO :gsumme FROM Abt WHERE Anr = 17;	
IF gsumme <> summe THEN <Fehlerbehandlung>;	

Zeit

Motivation - Erinnerung (6)

- Korrektheit – Vorüberlegungen (Forts.)
  - mehrere TAs (Forts.)



Motivation - Erinnerung (7)

- Formales Korrektheitskriterium: *Serialisierbarkeit*

  Die parallele Ausführung einer Menge von TA ist serialisierbar,

  wenn es eine serielle Ausführung derselben TA-Menge gibt,

  die *den gleichen DB-Zustand*

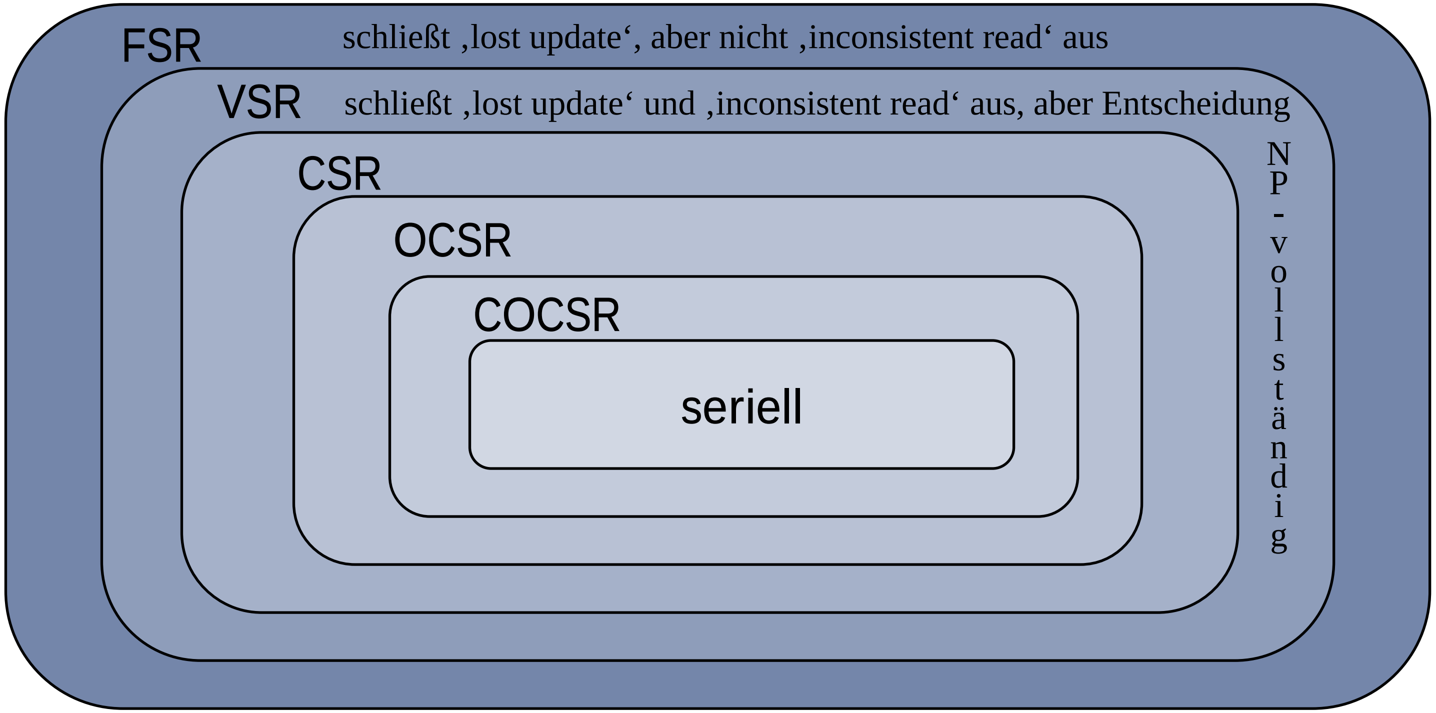
  <u>und</u> *die gleichen Ausgabewerte*

  wie die ursprüngliche Ausführung erzielt.

  - Hintergrund:
    - Serielle Ablaufpläne sind korrekt
    - Jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar

Motivation (1)

- Ziel dieses Kapitels
  - detailliertere und
  - formale Betrachtung des Serialisierbarkeitsbegriffs
- Klassen (vereinfachter Ausblick)



Motivation (1)

- Anforderungen an akzeptable Klasse von so genannten Schedules (siehe unten)
  - Mindestens *lost update* und *inconsistent read* werden vermieden
  - Zugehörigkeit eines Schedules kann effizient entschieden werden
  - Bei Annahme von Fehlern (Aborts) wird Abhängigkeit von nicht-freigegebenen Änderungen *(dirty read)* vermieden
- Daher Konzentration auf Konfliktserialisierbarkeit (CSR)
  - CSR ist wichtigste Art der Serialisierbarkeit für die praktische Nutzung

Seiten-Modell (1)

- Modellbildung
  - Seiten-Modell (Grundlage dieses Kapitels)
    - abstraktes Modell, nicht notwendigerweise auf den tatsächlichen Seitenbegriff beschränkt
    - jedoch ist seiten-orientierte Synchronisation und Recovery (im Speichersystem eines DBS) der Hauptanwendungsbereich des Seitenmodells
- Grundlegende Begriffe
  - Menge von *unteilbaren, uninterpretierten Datenobjekten* (Seiten)
    - D = {x, y, z, ...}
    - mit *atomaren Lese- und Schreiboperationen*

Seiten-Modell (2)

- Grundlegende Begriffe (Forts.)
  - Eine *Transaktion t* wird zunächst als eine endliche Folge von Schritten/Aktionen der Form r(x) oder w(x) betrachtet:
    - $t = p_1 \ldots p_{n,}$ mit $n < \infty$, $p_i \in \{r(x), w(x)\}$ für $1 \leq i \leq n$, $x \in D$;
    - r steht für Lesen, w für Schreiben
  - Verschiedene Transaktionen haben keine Schritte gemeinsam; Schritte können eindeutig identifiziert werden:
    - p_{ij} bezeichnet den j-ten Schritt von Transaktion i
      (Transaktions-Index kann weggelassen werden, falls Kontext klar)

Seiten-Modell (3)

- Interpretation einer Transaktion
  - $p_j = r(x)$
    - der j-te Schritt der TA ist eine Leseoperation, mit der der aktuelle Wert von x einer lokalen Variablen v_j zugewiesen wird
    - $v_j := x$
  - $p_j = w(x)$
    - der j-te Schritt der TA ist eine Schreiboperation, mit der ein im zugehörigen Programm berechneter Wert x zugewiesen wird
    - jeder Wert, der von einer TA geschrieben wird, ist potentiell abhängig von allen Datenobjekten, die t vorher gelesen hat
    - $x := f_j\ (v_{j1}, \ldots, v_{jk})$
    - x ist der Rückgabewert einer beliebigen, unbekannten Funktion f_j mit $\{j_1, \ldots j_k\} = \{j_r \mid p_{j_r}$ ist Leseoperation $\wedge\ j_r < j\}$

Seiten-Modell (4)

- Bisher Annahme einer totalen Ordnung über den Schritten einer TA
  - nicht nötig, solange ACID eingehalten wird
  - nicht sinnvoll, z.B. im Falle einer parallelisierten TA-Ausführung auf einem Mehrprozessorsystem
- Definition *Partialordnung*
  Sei A beliebige Menge. $R \subseteq A \times A$ ist eine Partialordnung auf A, wenn für beliebige Elemente $a, b, c \in A$ gilt:
  - $(a, a) \in R$ (Reflexivität)
  - $(a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$ (Antisymmetrie)
  - $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ (Transitivität)

  Beachte: jedes R kann als gerichteter Graph dargestellt werden.

Seiten-Modell (5)

- Definition ***Transaktion***
  - Eine *Transaktion t* ist eine Partialordnung von Schritten der Form r(x) oder w(x) mit $x \in D$ und Lese- und Schreiboperationen sowie mehrfache Schreiboperationen auf demselben Datenobjekt sind geordnet
  - Formal: $t = (op, <)$
    - op ist endliche Menge von Schritten r(x) oder w(x), $x \in D$
    - $< \subseteq op \times op$ ist Partialordnung über op mit
    - falls $\{p, q\} \subseteq op$ und p und q greifen auf dasselbe Datenobjekt zu und mindestens eine der beiden ist eine Schreiboperationen gilt:

      $p < q \vee q < p$.

Seiten-Modell (6)

- Ordnungsanforderung in der Definition sichert eindeutige Interpretation
  - Würde man beispielsweise eine Lese- und ein Schreiboperation auf demselben Datenobjekt ungeordnet belassen
    - wäre der gelesene Wert nicht eindeutig
    - es könnte der Wert vor dem Schreiben oder der danach sein
- Weitere Annahmen
  - in jeder TA wird jedes Datenobjekt höchstens einmal gelesen oder geschrieben
  - kein Datenobjekt wird (nochmal) gelesen, nach dem es geschrieben wurde

    (schließt nicht ‚blindes' Schreiben aus!)

Historien und Schedules (1)

- Ziel
  - Entwickeln eines Korrektheitsbegriffes für parallele TA-Ausführungen
  - Scheduler, als Kern der Synchronisationskomponente, braucht Korrektheitskriterien, die effizient angewendet werden können

- (zusätzliche) Terminierungsoperationen
  - c_i: erfolgreiches Ende einer TA t_i, Commit
  - a_i: nicht-erfolgreiches Ende einer TA t_i, Abbruch, Abort

Historien und Schedules (2)

- Definition ***Historien und Schedules***
  - Es sei $T = \{t_1, \ldots, t_n\}$ eine (endliche) Menge von TA, 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 TA, die auf dasselbe Datenelement zugreifen und von denen wenigstens eine davon eine Schreib-Operation ist, sind so geordnet, dass $p <_s q$ oder $q <_s p$ gilt
  - Ein *Schedule* ist ein Präfix einer Historie

Historien und Schedules (3)

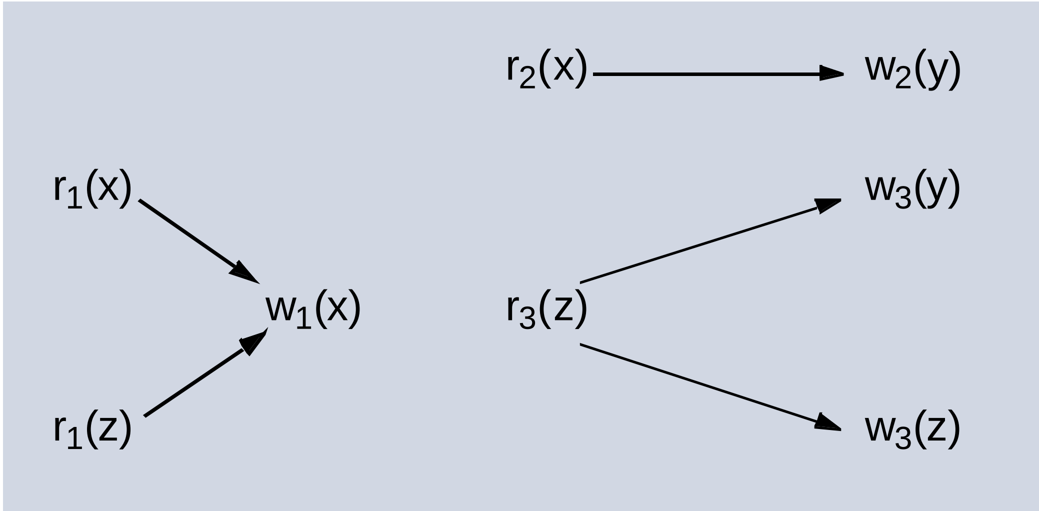
- Erläuterungen zur Definition:
  - eine Historie (für partiell geordnete TA)
    a) enthält alle Operationen aller TA
    b) benötigt eine bestimmte Terminierungsoperation für jede TA
    c) bewahrt alle Ordnungen innerhalb der TA
    d) hat die Terminierungsoperationen als letzte Operationen in jeder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

Historien und Schedules (4)

- Bemerkung
  - 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.

Historien und Schedules (5)

- Beispiel
  - 3 TA als DAG (directed acyclic graph)

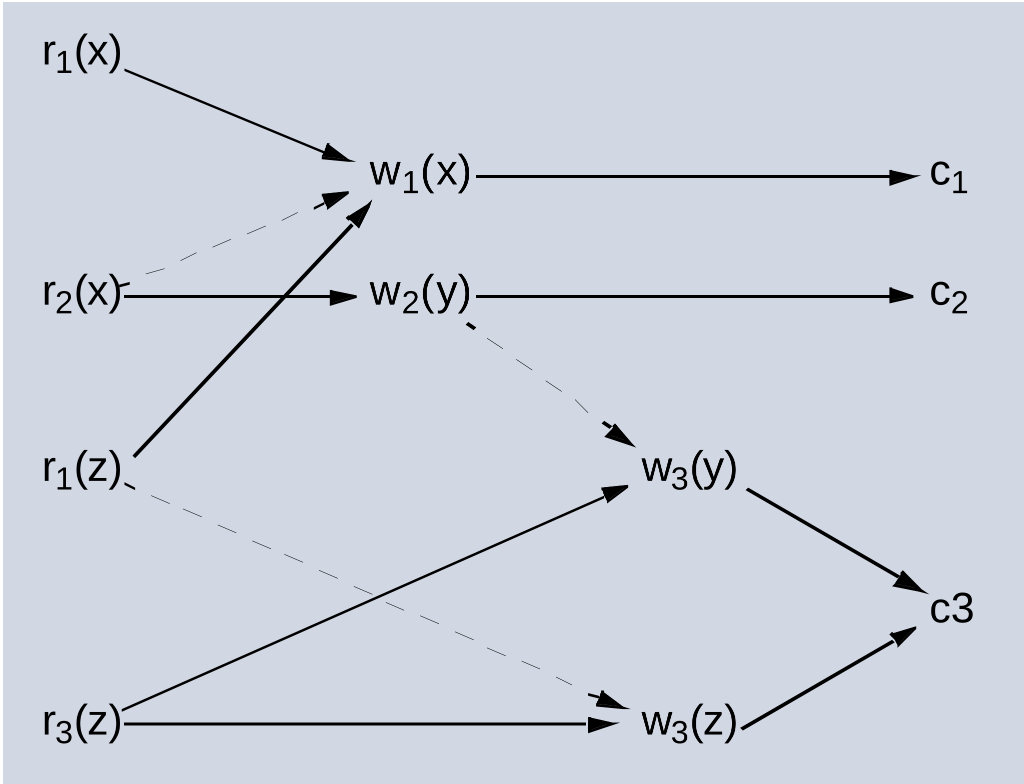


  - Beispiel einer vollständigen geordneten Historie dieser 3 TA

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

Historien und Schedules (6)

- Beispiel (Forts.)
  - Beispiel einer partiell geordneten Historie dieser 3 TA



  - Teilordnungen lassen sich stets erweitern zu einer Vielfalt an Vollordnungen (als Spezialfälle)

Historien und Schedules (7)

- *Präfix einer partiellen Ordnung*
  - wird erreicht durch das Weglassen von Teilen vom Ende der *„Erreichbarkeitskette"*
  - wenn $s = (op(s), <_s)$, dann hat ein Präfix von s die Form $s' = (op_{s'}, <_{s'})$, so dass gilt:
    - $op_{s'} \subseteq op(s)$
    - $<_{s'} \subseteq <_s$
    - $(\forall\, p \in op_{s'})\ (\forall\, q \in op(s))\ \ q <_s p \ \Rightarrow\ q \in op_{s'}$
    - $(\forall\, p, q \in op_{s'})\ p <_s q \ \Rightarrow\ p <_{s'} q$

Historien und Schedules (8)

- *Shuffle-Produkt (Misch-Produkt)*
  - sei $T = \{t_1, \ldots, t_n\}$ eine Menge von vollständig geordneten TA
  - shuffle(T) bezeichnet das Shuffle-Produkt, d.h., die Menge aller Operationsfolgen, in denen jede Folge $t_i \in T$ als Teilfolge auftritt und die keine anderen Operationen enthält
- Vollständig geordnete Historien und Schedules
  - eine Historie s für T wird von einer Folge $s' \in$ shuffle(T) abgeleitet, wobei c_i oder a_i für jedes $t_i \in T$ hinzugefügt wird (Regel b) und d) von Definition auf Folie 17).
  - ein Schedule ist, wie bisher, ein Präfix einer Historie.
  - eine Historie s ist seriell, wenn $s = t_{i_1}, \ldots, t_{i_n}$ wobei $i_1, \ldots, i_n$ eine Permutation von 1, …, n ist

Historien und Schedules (9)

- Beispiel (Fortführung Folie 20)
  - es können vollständig geordnete TA wie folgt gebildet werden:

    $t_1 = r_1(x)\ r_1(z)\ w_1(x)$
    $t_2 = r_2(x)\ w_2(y)$
    $t_3 = r_3(z)\ w_3(y)\ w_3(z)$

  - Die Historie

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

    ist vollständig geordnet und hat (unter anderen)

    $r_1(x)\ r_2(x)\ r_1(z)\ w_1(x)\ w_2(y)\ r_3(z)\ w_3(y)$,

    $r_1(x)\ r_2(x)\ r_1(z)\ w_1(x)\ w_2(y)$, und

    $r_1(x)\ r_2(x)\ r_1(z)$

    als Präfixe

Historien und Schedules (10)

- (Neues) Beispiel
  - $T = \{t_1, t_2, t_3\}$ mit

    $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
    $t_2 = r_2(z)\ w_2(x)\ w_2(z)$
    $t_3 = r_3(x)\ r_3(y)\ w_3(z)$

    $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)$ $\in$ shuffle(T);

    $s_2 = s_1\ c_1\ c_2\ a_3$ ist eine Historie, in der das Shuffle-Produkt von T ergänzt wurde um die Terminierungsschritte;

    $s_3 = r_1(x)\ r_2(z)\ r_3(x)$ ist ein Schedule;

    $s_4 = s_1\ c_1$ ist ein anderer Schedule;

    $s_5 = t_1\ c_1\ t_3\ a_3\ t_2\ c_2$ ist eine serielle Historie.

Historien und Schedules (11)

- Anmerkung
  - die hier erhaltenen Ergebnisse gelten für vollständige wie auch für partielle Ordnungen
  - es ist meist einfacher, sie für vollständige Ordnungen herzuleiten
- Definitionen *TA-Mengen eines Schedules*
  - trans(s) := $\{t_i \mid s$ enthält Schritte von $t_i\}$
  - commit(s) := $\{t_i \in \text{trans}(s) \mid c_i \in s\}$
  - abort(s) := $\{t_i \in \text{trans}(s) \mid a_i \in s\}$
  - active(s) := trans(s) - (commit(s) $\cup$ abort(s))

Historien und Schedules (12)

- Beispiel (Fortführung Folie 25)
  - $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ c_2\ a_3$

    trans(s_1) = $\{t_1, t_2, t_3\}$
    commit(s_1) = $\{t_1, t_2\}$
    abort(s_1) = $\{t_3\}$
    active(s_1) = Ø

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

    trans(s_2) = $\{t_1, t_2, t_3\}$
    commit(s_2) = $\{t_1\}$
    abort(s_2) = Ø
    active(s_2) = $\{t_2, t_3\}$

Historien und Schedules (13)

- Für jede Historie s gilt:
  - trans(s) = commit(s) $\cup$ abort(s)
  - active(s) = Ø

Historien und Schedules (14)

- Definition *Monotone Klassen von Historien*
  - Eine Klasse E von Historien heißt monoton, wenn folgendes gilt:
    - Wenn s in E ist, dann ist $\Pi_T(s)$, die Projektion von s auf T genannt, 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
  - VSR ist nicht monoton

Korrektheit (1)

- Ein *Korrektheitskriterium* kann formal betrachtet werden als Abbildung
  - $\sigma : S \rightarrow \{0, 1\}$ mit S Menge aller Schedules.
  - $correct(S) := \{s \in S \mid \sigma(s)=1\}$
- Ein *konkretes Korrektheitskriterium* sollte mindestens die folgenden Anforderungen erfüllen
  1. $correct(S) \neq \emptyset$
  2. „$s \in correct(S)$“ ist effizient entscheidbar
  3. 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

Korrektheit (2)

- 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
     - $[S]_{\approx} = \{[s]_{\approx} \mid s \in S\}$ Menge der Äquivalenzklassen
  2. Betrachten solcher Klassen mit seriellen Schedules als repräsentative Vertreter

Klasse CSR (1)

- Konflikt-Serialisierbarkeit
  - wichtigste Art der Serialisierbarkeit für die praktische Nutzung
- Ziel
  - weitere Einschränkungen im Vergleich zu VSR (nicht monoton und Test auf Mitgliedschaft 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' $\in$ 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
  - conf(s) := {(p, q) | p, q sind in Konflikt in s und p $<_s$ q} heißt Konfliktrelation von s

Klasse CSR (2)

- Bemerkung
  - Konflikte bestehen nur zwischen Datenoperationen, unabhängig vom Terminierungsstatus der TA; Operationen von abgebrochenen TA können dennoch ignoriert werden
- Beispiel
  - $s = w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(y)\ w_3(x)\ w_3(y)\ c_1\ a_2$
  - $conf(s) = \{(w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}$
- Definition *Konfliktäquivalenz*
  - Schedules s und s' sind konfliktäquivalent, ausgedrückt durch $s \approx_c s'$, wenn
    - $op(s) = op(s')$
    - $conf(s) = conf(s')$

Motivation

Seiten-Modell

Historien

Korrektheit

CSR

Klasse CSR (3)

- Beispiel ($s \approx_c s'$)
  - $s = r_1(x)\ r_1(y)\ w_2(x)\ w_1(y)\ r_2(z)\ w_1(x)\ w_2(y)$
  - $s' = r_1(y)\ r_1(x)\ w_1(y)\ w_2(x)\ w_1(x)\ r_2(z)\ w_2(y)$
- Konfliktschritte-Graph $D_2(s)$
  - Konfliktäquivalenz lässt sich durch einen Graph $D_2(s) := (V, E)$ mit $V = op(s)$ und $E = conf(s)$ veranschaulichen
  - $D_2(s)$ heißt Konfliktschritte-Graph (conflicting-step graph) und
  - es gilt: $s \approx_c s' \Leftrightarrow D_2(s) = D_2(s')$
- 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

Klasse CSR (4)

- 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 (5)

- 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 $I \not\approx_c t_2\ t_1$
- $CSR \subset VSR \subset FSR$

Klasse CSR (6)

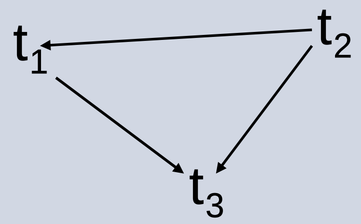
- Beispiel
  - $s = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$
  - $s \not\approx_c t_1\ t_2\ t_3$ und $s \notin$ CSR, aber
    $s \approx_v t_1\ t_2\ t_3$ und damit $s \in$ VSR
- Theorem
  - CSR ist monoton
  - $s \in$ CSR $\Leftrightarrow \Pi_T(s) \in$ VSR für alle $T \subseteq$ trans(s)
    (d.h., CSR ist die größte monotone Teilmenge von VSR)

Klasse CSR (7)

- 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 TA (conf(s)) und repräsentiert mehrfache Konflikte zwischen denselben (abgeschlossenen) TA durch eine einzige Kante

Klasse CSR (8)

- 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) =

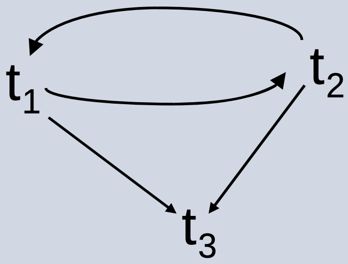


- Theorem ***Serialisierbarkeitstheorem***
  - Sei s eine Historie; dann gilt: $s \in CSR$ gdw G(s) azyklisch
- Aufgabe
  - Finden einer seriellen Historie, die konsistent mit allen Kanten in G(s) ist

Klasse CSR (9)

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

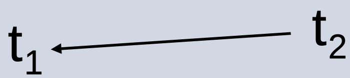
G(s) =



$s \notin CSR$

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

G(s') =



$s' \in CSR$

Klasse CSR (10)

- Korollar
  - Mitgliedschaft in CSR lässt sich in polynomialer Zeit in der Menge der am betreffenden Schedule teilnehmenden TA testen

- 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 TA 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 (in VSR) gdw sie konfliktserialisierbar (in CSR) ist!

Klasse CSR (11)

- 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

Klasse CSR (12)

- 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

Klasse CSR (13)

- Beispiel

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

- Definition *Kommutativitätsbasierte Äquivalenz*
  - Zwei Schedules s uns s' mit op(s) = op(s') sind kommutativitätsbasiert äquivalent, ausgedrückt durch s ~* s', wenn s nach s' transformiert werden kann durch eine endliche Anwendung der Regeln C1, C2, C3 und C4.

Klasse CSR (14)

- Theorem
  - s und s' seien Schedules mit op(s) = op(s')
  - Dann gilt s $\approx_c$ s' gdw s $\sim^*$ s'
- 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 $\in$ CSR

Klasse CSR (15)

- 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 (1)

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

$$t_3 \longrightarrow t_1 \longrightarrow t_2$$

- 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, 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 (siehe vorhergehende Folie) zeigt jedoch, dass die Inklusionsbeziehung echt ist: $s \in CSR - OCSR$

Klasse COCSR (1)

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

Klasse COCSR (2)

- Theorem
  - COCSR bezeichne die Klasse aller Historien, die „commit order-preserving conflict serializable“ sind; es gilt
  - COCSR $\subset$ CSR
- 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

Commit Serialisierbarkeit (1)

- Bisher Annahme,
  - dass jede betrachtete TA erfolgreich terminiert
- Anforderungen aufgrund möglicher Fehlerfälle
  1. Ein Korrektheitskriterium sollte ‚nur' erfolgreich abgeschlossene TA berücksichtigen
  2. Für jeden korrekten Schedule sollte jeder seiner Präfixe korrekt sein
- Definition *Hülleneigenschaften*
  - Sei E eine Klasse von Schedules
    1. E ist *präfix-abgeschlossen*, wenn für jeden Schedule s in E jeder Präfix von s auch in E ist
    2. E ist *commit-abgeschlossen*, wenn für jeden Schedule s in E auch CP(s), wobei CP(s) = $\Pi_{commit(s)}$ (s), in E ist

Commit Serialisierbarkeit (2)

- *Präfix-Commit-Abgeschlossenheit*
  - Erfüllung der beiden vorgenannten Abgeschlossenheits-eigenschaften
  - Falls Klasse E präfix-commit-abgeschlossen, dann gilt für jeden Schedule s in E, dass CP(s') in E für jeden Präfix s' von s
- FSR ist nicht präfix-commit-abgeschlossen
  - $s = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$
  - $s \approx_v t_1\ t_2\ t_3$, daher $s \in VSR$, daher $s \in FSR$
  - $s' = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$ ist Präfix von s
  - $CP(s') = s'$
  - $s' \not\approx_f t_1\ t_2$ und $s' \not\approx_f t_2\ t_1$, daher $s' \notin FSR$
- VSR ist schon deshalb nicht präfix-commit-abgeschlossen, da VSR nicht monoton

Commit Serialisierbarkeit (3)

- Theorem
  - CSR ist präfix-commit-abgeschlossen
  - Beweis
    - Sei s $\in$ CSR, daher ist G(s) azyklisch
    - Für jede Teilfolge s’ von s ist auch G(s’) azyklisch
    - Insbesondere G(CP(s’)) ist azyklisch
    - Damit CP(s’) $\in$ CSR
- Definition ***Commit-Serialisierbarkeit***
  - Ein Schedule s heißt *commit-serialisierbar*, wenn für jeden Präfix s’ CP(s’) serialisierbar ist.
- Klassen commit-serialisierbarer Schedules
  - CMFSR
  - CMVSR
  - CMCSR

Commit Serialisierbarkeit (4)

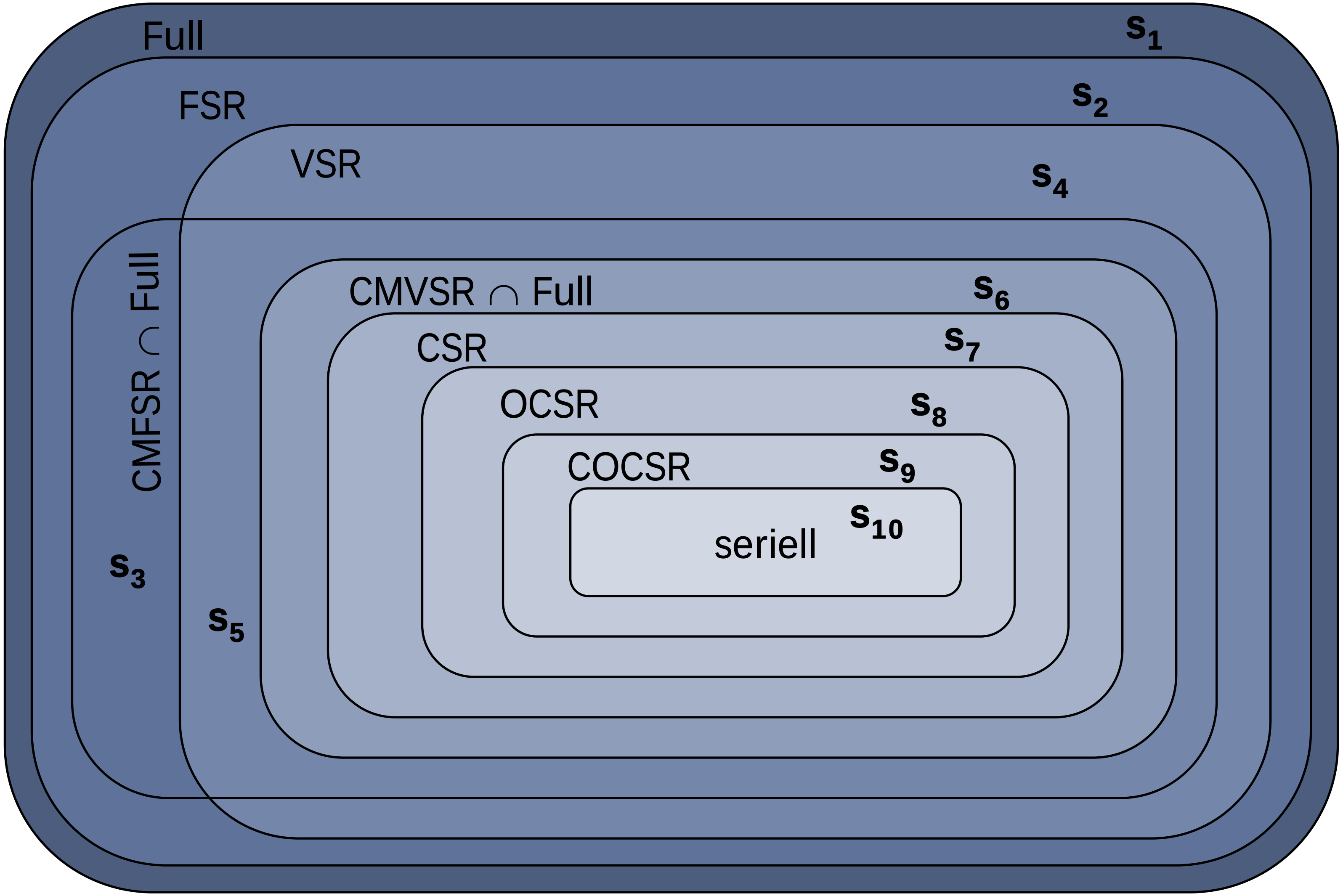
- Theorem
  1. CMFSR, CMVSR, CMCSR sind commit-abgeschlossen
  2. CMCSR $\subset$ CMVSR $\subset$ CMFSR
  3. CMFSR $\subset$ FSR
  4. CMVSR $\subset$ VSR
  5. **CMCSR = CSR**

Alle Klassen im Überblick (1)

- Historien
  - $s_1 = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$
  - $s_2 = w_1(x)\ r_2(x)\ w_2(y)\ c_2\ r_1(y)\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$
  - $s_3 = w_1(x)\ r_2(x)\ w_2(y)\ w_1(y)\ c_1\ c_2$
  - $s_4 = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$
  - $s_5 = w_1(x)\ r_2(x)\ w_2(y)\ w_1(y)\ c_1\ c_2\ w_3(x)\ w_3(y)\ c_3$
  - $s_6 = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ w_3(x)\ w_3(y)\ c_3\ w_1(z)\ c_1$
  - $\mathbf{s_7 = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(z)\ c_1}$
  - $\mathbf{s_8 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1}$
  - $\mathbf{s_9 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ w_1(y)\ c_1\ c_2}$
  - $\mathbf{s_{10} = w_1(x)\ w_1(y)\ c_1\ w_2(x)\ w_2(y)\ c_2}$

Alle Klassen im Überblick (2)

- Klassen-Übersicht



Zusammenfassung (1)

- Korrektheitskriterium der Synchronisation:
  - (Konflikt-)Serialisierbarkeit
- Theorie der Serialisierbarkeit
  - einfaches Read/Write-Modell (Syntaktische Behandlung)
  - Konfliktoperationen: reihenfolgeabhängige Operationen verschiedener Transaktionen auf denselben DB-Daten
  - Konflikt-Serialisierbarkeit
    - für praktische Anwendungen relevant (im Gegensatz zu Final-State- und View-Serialisierbarkeit)
    - effizient überprüfbar
    - es gilt: $CSR \subset VSR \subset FSR$
  - Serialisierbarkeitstheorem: Eine Historie s ist genau dann konfliktserialisierbar, wenn der zugehörige G(s) azyklisch ist

Zusammenfassung (2)

- Theorie der Serialisierbarkeit (Forts.)
  - CSR, obwohl weniger allgemein als VSR, ist am besten geeignet
    - aus Gründen der Komplexität
    - wegen Monotonizitätseigenschaft
    - wegen Verallgemeinerbarkeit für semantisch reichhaltigere Operationen
  - OCSR und COCSR haben weitere nützliche Eigenschaften
  - Commit-Serialisierbarkeit bezieht mögliche Abbrüche mit ein
- Serialisierbare Abläufe
  - Gewährleisten ‚automatisch' Korrektheit des Mehrbenutzerbetriebs
  - Anzahl der möglichen Schedules bestimmt erreichbaren Grad der Parallelität