7. Transaktionsmodelle

- Transaktionseigenschaften
- Probleme im Mehrbenutzerbetrieb
- Serialisierbarkeit
- Transaktionsabbruch und Fehlersicherheit
- Ausnutzung semantischer Informationen
- Erweiterte Transaktionsmodelle

Transaktionen im Mehrbenutzerbetrieb

Ablaufintegrität

- Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten
- mehrere Programme laufen simultan
  → nebenläufige, konkurrierende Prozesse
- *Transaktion* als Verarbeitungseinheit

Beispielszenarien

- Platzreservierung für Flüge quasi gleichzeitig aus vielen Reisebüros
  → Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren
- überschneidende Kontooperationen einer Bank
- statistische Datenbankoperationen
  → Ergebnisse sind verfälscht, wenn während der Berechnung Daten geändert werden

Transaktionseigenschaften

Eine *Transaktion* ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das *ACID-Prinzip* eingehalten werden muß.

ACID-Eigenschaften

- **A**tomicity (Atomarität):
  Transaktion wird entweder ganz oder gar nicht ausgeführt
- **C**onsistency (Konsistenz oder auch Integritätserhaltung):
  Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand
- **I**solation (Isolation):
  Nutzer, der mit einer Datenbank arbeitet, sollte den Eindruck haben, daß er mit dieser Datenbank alleine arbeitet
- **D**urability (Dauerhaftigkeit / Persistenz):
  nach erfolgreichem Abschluß einer Transaktion muß das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden

Kommandos einer Transaktionssprache

- Beginn einer Transaktion:
  Begin-of-Transaction-Kommando **BOT**
- **commit**: die Transaktion soll erfolgreich beendet werden
- **abort**: die Transaktion soll abgebrochen werden

Probleme im Mehrbenutzerbetrieb

- Inkonsistentes Lesen: Nonrepeatable Read
- Lesen inkonsistenter Zustände
- Abhängigkeiten von nicht freigegebenen Daten: Dirty Read
- Das Phantom-Problem
- Verlorengegangenes Ändern: Lost Update
- Integritätsverletzung durch Mehrbenutzer-Anomalie
- Probleme bei Cursor-Referenzen

Nonrepeatable Read

Beispiel:

- Zusicherung $X = A + B + C$ am Ende der Transaktion T_1
- X und Y seien lokale Variablen
- T_i ist die Transaktion i
- Integritätsbedingung $A + B + C = 0$

Beispiel für inkonsistentes Lesen

T_1	T_2
$X := A$;	
	$Y := A/2$;
	$A := Y$;
	$C := C + Y$;
	commit;
$X := X + B$;	
$X := X + C$;	
commit;	

Lesen inkonsistenter Zustände

Integritätsbedingung $X + Y = 0$

T_1	T_2
read(X);	
read(Y);	
write(X);	
$Y := Y + 1$;	
	read(X);
	read(Y);
write(Y);	
commit;	

Dirty Read

T_1	T_2
read(X);	
$X := X + 100$;	
write(X);	
	read(X);
	$Y := Y + X$;
	write(Y);
	commit;
abort;	

Das Phantom-Problem

T_1	T_2
select count (*)	
into X	
from `Mitarbeiter`;	
	insert
	into `Mitarbeiter`
	values $(Meier, 50000, \cdots)$;
	commit;
update `Mitarbeiter`	
set `Gehalt` $=$	
`Gehalt` $+10000/X$;	
commit;	

Lost Update

T_1	T_2	X
read(X);		10
	read(X);	10
$X := X + 1$;		10
	$X := X + 1$;	10
write(X);		11
	write(X);	11

Mehrbenutzer-Anomalie

Integritätsbedingung $A = B$

$$T_1 := \langle A := A + 10; B := B + 10 \rangle$$
$$T_2 := \langle A := A * 1.1; B := B * 1.1 \rangle$$

T_1 und T_2 erhalten isoliert die IB

Mehrbenutzer-Anomalie II

T_1	T_2	A	B
read(A);		10	10
$A := A + 10$;			
write(A);		20	
	read(A);		
	$A := A * 1.1$;		
	write(A);	22	
	read(B);		
	$B := B * 1.1$;		
	write(B);		11
read(B);			
$B := B + 10$;			
write(B);		22	21

Probleme bei Cursor-Referenzen

T_1	T_2
Positioniere Cursor C_1 auf nächstes Tupel mit Eigenschaft P (Tupel A)	
	Verändere Eigenschaft $P \rightarrow P'$ von A
Lies laufendes Tupel	

Probleme mit Cursor: SQL-Notation

T_1	T_2
declare C_1 **cursor for** **select** * **from** `Mitarbeiter` **where** `Abt = 'Verkauf';` ...	
	update `Mitarbeiter` **set** `Abt = 'Einkauf'` **where** `MName = ...;`
fetch C_1 **into** ...	

Serialisierbarkeit

- Einführung in die Thematik
- Formalisierung von Abläufen (Schedules)
- Serialisierbarkeitsbegriffe
- Vergleich der Serialisierbarkeitsbegriffe

Einführung in die Serialisierbarkeit

T_1 : **read** A; $A := A - 10$; **write** A; **read** B;
$B := B + 10$; **write** B;

T_2 : **read** B; $B := B - 20$; **write** B; **read** C;
$C := C + 20$; **write** C;

Ausführungsvarianten für zwei Transaktionen:

- seriell, etwa T_1 vor T_2
- „gemischt“, etwa abwechselnd Schritte von T_1 und T_2

Beispiele für verschränkte Ausführungen

Ausführung 1		**Ausführung 2**		**Ausführung 3**	
T_1	T_2	T_1	T_2	T_1	T_2
read A		**read** A		**read** A	
$A - 10$			**read** B	$A - 10$	
write A		$A - 10$			**read** B
read B			$B - 20$	**write** A	
$B + 10$		**write** A			$B - 20$
write B			**write** B	**read** B	
	read B	**read** B			**write** B
	$B - 20$		**read** C	$B + 10$	
	write B	$B + 10$			**read** C
	read C		$C + 20$	**write** B	
	$C + 20$	**write** B			$C + 20$
	write C		**write** C		**write** C

Effekt unterschiedlicher Ausführungen

	A	B	C	$A + B + C$
initialer Wert	10	10	10	30
nach Ausführung 1	0	0	30	30
nach Ausführung 2	0	0	30	30
nach Ausführung 3	0	20	30	50

Vereinfachtes Modell

Lock-Unlock-Modell

T_1 : **lock** A; **unlock** A; **lock** B; **unlock** B;
T_2 : **lock** B; **unlock** B; **lock** C; **unlock** C;

Beispiele II

Ausführung 1		Ausführung 2		verbotene Ausführung	
T_1	T_2	T_1	T_2	T_1	T_2
lock A		**lock** A		**lock** A	
unlock A			**lock** B		**lock** B
lock B		**unlock** A		**unlock** A	
unlock B			**unlock** B	**lock** B	
	lock B	**lock** B			**unlock** B
	unlock B		**lock** C		**lock** C
	read C	**unlock** B		**unlock** B	
	unlock C		**unlock** C		**unlock** C

Semantik mit Berechnungsfunktionen I

T_1		T_2		T_3	
lock A		**lock** B		**lock** A	
lock B		**lock** C		**lock** C	
unlock A	$f_1(A,B)$	**unlock** B	$f_3(B,C)$	**unlock** C	$f_6(A,C)$
unlock B	$f_2(A,B)$	**lock** A		**unlock** A	$f_7(A,C)$
		unlock C	$f_4(A,B,C)$		
		unlock A	$f_5(A,B,C)$		

Semantik mit Berechnungsfunktionen II

Schritt		A	B	C
(1)	T_1: **lock** A	A_0	B_0	C_0
(2)	T_2: **lock** B	A_0	B_0	C_0
(3)	T_2: **lock** C	A_0	B_0	C_0
(4)	T_2: **unlock** B	A_0	$f_3(B_0, C_0) = \sigma_i$	C_0
(5)	T_1: **lock** B	A_0	σ_i	C_0
(6)	T_1: **unlock** A	$f_1(A_0, \sigma_i) = \sigma_{ii}$	σ_i	C_0
(7)	T_2: **lock** A	σ_{ii}	σ_i	C_0
(8)	T_2: **unlock** C	σ_{ii}	σ_i	$f_4(\sigma_{ii}, B_0, C_0) = \sigma_{iii}$
(9)	T_2: **unlock** A	$f_5(\sigma_{ii}, B_0, C_0) = \sigma_{iv}$	σ_i	σ_{iii}
(10)	T_3: **lock** A	σ_{iv}	σ_i	σ_{iii}
(11)	T_3: **lock** C	σ_{iv}	σ_i	σ_{iii}
(12)	T_1: **unlock** B	σ_{iv}	$f_2(A_0, \sigma_i)$	σ_{iii}
(13)	T_3: **unlock** C	σ_{iv}	$f_2(A_0, \sigma_i)$	$f_6(\sigma_{iv}, \sigma_{iii})$
(14)	T_3: **unlock** A	$f_7(\sigma_{iv}, \sigma_{iii})$	$f_2(A_0, \sigma_i)$	$f_6(\sigma_{iv}, \sigma_{iii})$

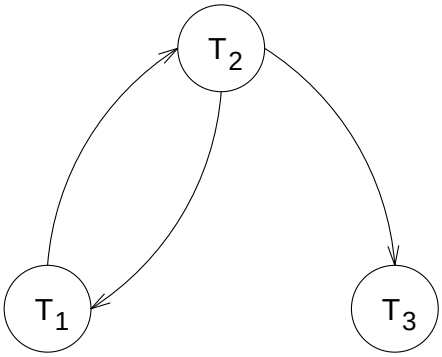
Letzter Wert für A voll ausgeschrieben

$$f_7(\sigma_{iv}, \sigma_{iii}) = \quad f_7(f_5(f_1(A_0, f_3(B_0, C_0)), B_0, C_0), f_4(f_1(A_0, f_3(B_0, C_0)), B_0, C_0))$$

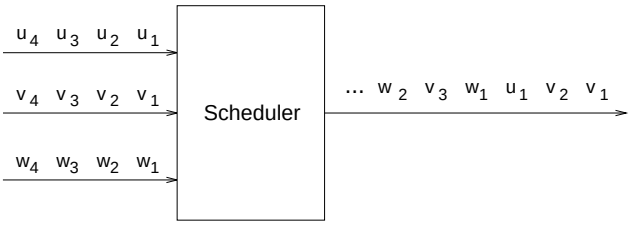
Serialisierbarkeit

Eine verschränkte Ausführung mehrerer Transaktionen heißt serialisierbar, wenn ihr Effekt identisch zum Effekt einer (beliebig gewählten) seriellen Ausführung dieser Transaktionen ist.

Konfliktgraph



Der Begriff des Schedules



Das Read/Write-Modell

- *Transaktion* T ist eine endliche Folge von Operationen (Schritten) p_i der Form $r(x_i)$ oder $w(x_i)$:

$$T = p_1 p_2 p_3 \cdots p_n \text{ mit } p_i \in \{r(x_i), w(x_i)\}$$

- Vollständige Transaktion T hat als letzten Schritt entweder einen Abbruch a oder ein Commit c:

$$T = p_1 \cdots p_n a$$

oder

$$T = p_1 \cdots p_n c.$$

Verschränkte Ausführungen

SHUFFLE($\boldsymbol{T}$): *Menge aller verschränkten Ausführungen* der Einzelschritte aller in der Menge $\boldsymbol{T}$ enthaltenen Transaktionen T_i

- alle Schritte der Transaktionen T_i sind genau einmal enthalten
- relative Reihenfolge der Einzelschritte einer Transaktion wird beibehalten

$$\begin{aligned} T_1 &:= r_1(x)w_1(x) \\ T_2 &:= r_2(x)r_2(y)w_2(y) \end{aligned}$$

$$\begin{aligned} \text{SHUFFLE}(\boldsymbol{T}) = \quad & \{r_1(x)w_1(x)r_2(x)r_2(y)w_2(y), r_2(x)r_1(x) \\ & w_1(x)r_2(y)w_2(y),\ \dots\} \end{aligned}$$

Schedule

Ein *Schedule* ist ein Präfix eines vollständigen Schedules.

$$\underbrace{\underbrace{r_1(x)r_2(x)w_1(x)}_{\text{ein Schedule}}\, r_2(y)a_1w_2(y)c_2}_{\text{ein vollständiger Schedule}}$$

Serieller Schedule

Ein *serieller Schedule* s für $\boldsymbol{T}$ ist ein vollständiger Schedule der folgenden Form:

$s := T_{\rho(1)} \cdots T_{\rho(n)}$ für eine Permutation ρ von $\{1, \dots, n\}$

resultierende serielle Schedules für zwei Transaktionen $T_1 := r_1(x)w_1(x)c_1$ und $T_2 := r_2(x)w_2(x)c_2$:

$$s_1 := \underbrace{r_1(x)w_1(x)c_1}_{T_1}\,\underbrace{r_2(x)w_2(x)c_2}_{T_2}$$

$$s_2 := \underbrace{r_2(x)w_2(x)c_2}_{T_2}\,\underbrace{r_1(x)w_1(x)c_1}_{T_1}$$

Korrektheitskriterium

Ein Schedule s ist *korrekt*, wenn der Effekt des Schedules s (Ergebnis der Ausführung des Schedules) äquivalent dem Effekt eines (beliebigen) seriellen Schedules s' bzgl. derselben Menge von Transaktionen ist (in Zeichen $s \approx s'$).

Ist ein Schedule s äquivalent zu einem seriellen Schedule s', dann ist s *serialisierbar* (zu s').

Sichtserialisierbarkeit

Künstliche Zusatztransaktionen:

1. *Initialisierungstransaktion* T_0; schreibt am Anfang alle beteiligten Datenobjekte
2. *Terminierungstransaktion* T_∞; liest am Ende alle beteiligten Datenobjekte

Liest-von-Relation

Sei $\rightarrow_s$ die Relation „zeitlich vorher im Schedule s“, dann gilt „$r_j(x)$ liest x von T_i“ g.d.w.

- $w_i(x) \rightarrow_s r_j(x)$ und
- $\nexists k(w_i(x) \rightarrow_s w_k(x) \wedge w_k(x) \rightarrow_s r_j(x))$.

Liest-von-Relation $RF(s)$ für ein Schedule s:

$$RF(s) := \{ (T_i, x, T_j) \mid r_j(x) \text{ liest } x \text{ von } T_i \}$$

Sichtäquivalenz

Zwei Schedules s und s' sind *sichtäquivalent*, wenn gilt:

1. $op(s) = op(s')$
   $op(s)$: die Menge aller in s vorkommenden Schritte bezeichnet, einschließlich a und c (Mengen müssen für beide Schedules identisch sein)
2. $RF(s) = RF(s')$
   Schedules s und s' haben dieselbe „Liest-von-Relation“

Beispiel Sichtäquivalenz I

Geg.: zwei vollständige Schedules s_1 und s_2 bestehend aus zwei Transaktionen T_1 und T_2 mit:

$$\begin{aligned} s_1 &:= r_1(x) r_2(y) w_1(y) w_2(y) c_1 c_2 \\ s_2 &:= r_1(x) w_1(y) r_2(y) w_2(y) c_2 c_1 \end{aligned}$$

Beispiel Sichtäquivalenz II

- zur Berechnung der Liest-von-Relationen $RF(s_1)$ und $RF(s_2)$ werden s_1 und s_2 um die Initialisierungstransaktion T_0 und die Terminierungstransaktion T_∞ wie folgt erweitert:

$$\begin{aligned} s_1 &:= \underbrace{w_0(x) w_0(y) c_0}_{T_0}\, r_1(x) r_2(y) w_1(y) w_2(y) c_1 c_2\, \underbrace{r_\infty(x), r_\infty(y) c_\infty}_{T_\infty} \\ s_2 &:= \underbrace{w_0(x) w_0(y) c_0}_{T_0}\, r_1(x) w_1(y) r_2(y) w_2(y) c_2 c_1\, \underbrace{r_\infty(x), r_\infty(y) c_\infty}_{T_\infty} \end{aligned}$$

Beispiel Sichtäquivalenz III

- resultierende *Liest-von-Relationen*:

$$RF(s_1) := \{(T_0, x, T_1), (T_0, y, T_2), (T_0, x, T_\infty), (T_2, y, T_\infty)\}$$
$$RF(s_2) := \{(T_0, x, T_1), (T_1, y, T_2), (T_0, x, T_\infty), (T_2, y, T_\infty)\}$$

- *Sichtäquivalenz* von s_1 und s_2: Vergleich der die *Liest-von-Relationen*
  da $RF(s_1) \neq RF(s_2)$ gilt, sind s_1 und s_2 *nicht* sichtäquivalent

Sichtserialisierbarkeit

Ein Schedule s ist nun genau dann *sichtserialisierbar*, wenn s sichtäquivalent zu einem seriellen Schedule ist.

- Menge aller sichtserialisierbaren Schedules: **VSR** (für engl. *view serializability*)

Beispiel Sichtserialisierbarkeit

- mögliche seriellen Schedules s' und s'' für s_1:
  - $s' = T_1T_2 = r_1(x)w_1(y)c_1r_2(y)w_2(y)c_2$
    $RF(s') :=$
    $\{(T_0, x, T_1), (T_1, y, T_2), (T_0, x, T_\infty), (T_2, y, T_\infty)\}$
    $\rightarrow \quad RF(s') \neq RF(s_1)$
  - $s'' = T_2T_1 = r_2(y)w_2(y)c_2r_1(x)w_1(y)c_1$
    $RF(s'') :=$
    $\{(T_0, y, T_2), (T_0, x, T_1), (T_0, x, T_\infty), (T_1, y, T_\infty)\}$
    $\rightarrow \quad RF(s'') \neq RF(s_1)$

  $RF(s') \neq RF(s_1)$ und $RF(s'') \neq RF(s_1)$: Schedule s_1 ist nicht sichtserialisierbar!

Konfliktserialisierbarkeit

- Konfliktrelation C von s:

  $C(s) := \{ (p, q) \mid p, q$ sind in Konflikt und $p \rightarrow_s q \}$

- Konfliktmatrix:

	r_i(x)	w_i(x)
r_j(x)	$\surd$	–
w_j(x)	–	–

Bereinigte Konfliktrelation

- Mit conf(s) wird „bereinigte“ Konfliktrelation bezeichnet, in der keine abgebrochenen Transaktionen mehr vorkommen

  conf$(s) := C(s) - \{ (p, q) \mid (p \in t' \vee q \in t') \wedge t' \in$ **aborted**$(s) \}$

- **aborted**(s): Menge der abgebrochenen Transaktionen des Schedules s

Beispiel Konfliktrelation

- Geg.: Schedule s:

  $$s = r_1(x) w_1(x) r_2(x) r_3(y) w_2(y) c_2 a_1 c_3$$

- Konfliktrelation zu s:

  $$C(s) := \{(w_1(x), r_2(x)), (r_3(y), w_2(y))\}$$

- Entfernen der abgebrochene Transaktion T_1 aus s:

  $$\text{conf}(s) := \{(r_3(y), w_2(y))\}$$

Konfliktäquivalenz

- Zwei Schedules s und s' heissen *konfliktäquivalent* ($s \approx_c s'$) falls gilt:
  1. $op(s) = op(s')$
  2. conf(s) = conf(s')

Konfliktserialisierbarkeit

Ein Schedule s ist genau dann *konfliktserialisierbar*, wenn s konfliktäquivalent zu einem seriellen Schedule ist.

- Klasse aller konfliktserialisierbaren Schedules: **CSR** (für engl. *conflict serializabel*)

Beispiel Konfliktserialisierbarkeit

- Geg.: zwei Schedules s und s':

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

- Frage:
  Sind die Schedules s und s' konfliktäquivalent?
- 1. Schritt:
  $op(s) = op(s')$ gilt, da alle in s vorkommenden Datenbankoperationen auch in s' vorkommen; gilt auch umgekehrt

Beispiel Konfliktserialisierbarkeit II

- 2. Schritt: bereinigte Konfliktelationen

$$\begin{aligned}\text{conf}(s) &= \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)), \\ &\quad (w_1(y), w_2(y))\} \\ \text{conf}(s') &= \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)), \\ &\quad (w_1(y), w_2(y))\}\end{aligned}$$

  - Es gilt $\text{conf}(s) = \text{conf}(s')$; somit stimmen auch die Konfliktrelationen überein und damit sind s und s' konfliktäquivalent

Beispiel Konfliktserialisierbarkeit III

- Test auf Konfliktserialisierbarkeit durch Vergleich mit den seriellen Schedules
- Geg.: Schedule s

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

Beispiel Konfliktserialisierbarkeit IV

- bereinigte Konfliktrelation für s:

$$\begin{aligned}\text{conf}(s) &= \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), \\ &\quad w_2(y)), (w_1(y), w_2(y))\}\end{aligned}$$

- möglicher serieller Schedule s_1

$$s_1 = T_1T_2 = r_1(x)r_1(y)w_1(y)w_1(x)c_1w_2(x)r_2(z)w_2(y)c_2$$

  - Konfliktrelation von s_1 stimmt *nicht* mit der von s überein:

$$\begin{aligned}\text{conf}(s_1) &= \{(r_1(x), w_2(x)), (w_1(x), w_2(x)), \\ &\quad (r_1(y), w_2(y)), (w_1(y), w_2(y))\}\end{aligned}$$

Beispiel Konfliktserialisierbarkeit V

- möglicher serieller Schedule s_2 s Kandidat:

$$s_2 = T_2T_1 = w_2(x)r_2(z)w_2(y)c_2r_1(x)r_1(y)w_1(y)w_1(x)c_1$$

  - auch Konfliktrelation von s_2 stimmt *nicht* mit der von s überein:

$$\begin{aligned}\text{conf}(s_2) \quad = \quad & \{(w_2(x), r_1(x)), (w_2(y), r_1(y)),\\ & (w_2(y), w_1(y)), (w_2(x), w_1(x))\}\end{aligned}$$

- somit gilt: $s \notin$ **CSR**, d.h., der Schedule s ist nicht konfliktserialisierbar

Beispiel Konfliktserialisierbarkeit VI

- Schedule

$$s_3 = r_1(x)r_2(x)w_2(y)c_2w_1(x)c_1$$

  ist trivialerweise konfliktserialisierbar, da nur ein einziger Konflikt auftritt

Graphbasierter Test

Konfliktgraph $G(s) = (V, E)$ von Schedule s:

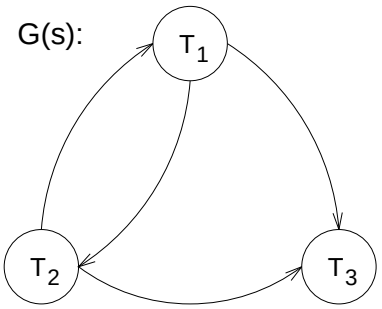
1. Knotenmenge V enthält alle in s vorkommende Transaktionen
2. Kantenmenge E enthält alle gerichteten Kanten zwischen zwei in Konflikt stehenden Transaktionen, also: $(t, t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t)(\exists q \in t')$ mit $(p, q) \in \text{conf}(s)$

Zeitlicher Verlauf dreier Transaktionen

T_1	T_2	T_3
r(y)		
		r(u)
	r(y)	
w(y)		
w(x)		
	w(x)	
	w(z)	
		w(x)

$$s = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$$

Konfliktgraph



Eigenschaften von Konfliktgraph $G(s)$

1. Ist s ein serieller Schedule, dann ist der vorliegende Konfliktgraph ein azyklischer Graph.
2. Für jeden azyklischen Graphen $G(s)$ lässt sich ein serieller Schedule s' konstruieren, so daß s konfliktserialisierbar zu s' ist (Test bspw. durch *topologisches Sortieren*)
3. Enthält ein Graph Zyklen, dann ist der zugehörige Schedule nicht konfliktserialisierbar.

Konfliktgraphen und -serialisierbarkeit

Für jeden Schedule s gilt:

$$G(s) \text{ azyklisch} \Leftrightarrow s \in \textbf{CSR}$$

Probleme zur Laufzeit

- zur Laufzeit nur unvollständige Schedules verfügbar $\rightsquigarrow$ Überwachung unvollständiger Schedules notwendig
- Transaktionen, die noch kein **commit** gemacht haben, können noch jederzeit abgebrochen werden

Abgeschlossenheitseigenschaften

1. *Präfix-Abgeschlossenheit*
   Falls eine Eigenschaft E für einen Schedule s gilt, dann gilt E auch für jeden Präfi x von s. Ist am Ende eines Schedules E erfüllt, dann darf E vorher nicht verletzt worden sein.
2. *Commit-Abgeschlossenheit*
   Gilt E für s, dann gilt E auch für $CP(s)$ („**c**ommitted **p**rojection“). Wenn E für eine Menge von Transaktionen gilt, dann gilt sie auch, wenn einige davon abgebrochen werden.
3. *Präfix-Commit-Abgeschlossenheit* (PCA)
   Präfi x-Commit-Abgeschlossenheit ist die Konjunktion.

Abgeschlossenheitseigenschaften

- **VSR**-Schedules sind **nicht** *präfix-commit-abgeschlossen*
- **CSR**-Schedules sind *präfix-commit-abgeschlossen*

Beispiel CSR versus VSR I

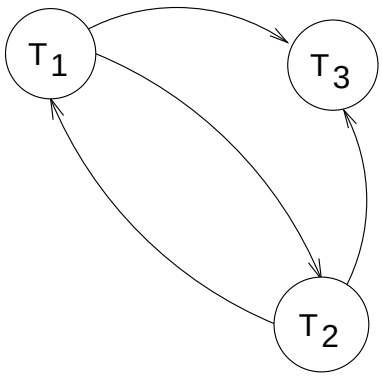
Gilt **CSR** $\subset$ **VSR** oder **CSR** $\subset$ **VSR**?

- Geg.: Schedule s:

  s = $r_1(y)r_3(w)\underbrace{r_2(y)w_1(y)}_{T_2 \to T_1}\underbrace{w_1(x)w_2(x)}_{T_1 \to T_2}w_2(z)w_3(x)c_2c_1c_3$

Beispiel CSR versus VSR II

Der Schedule s ist nicht konfliktserialisierbar, weil Konfliktgraph $G(s)$ einen Zyklus enthält



Beispiel CSR versus VSR III

Ist s *sichtserialisierbar*?

- Ermitteln der Liest-von-Relation RF

$$RF(s) = \{(T_0, y, T_1), (T_0, w, T_3), (T_0, y, T_2), (T_3, x, T_\infty), (T_1, y, T_\infty), (T_2, z, T_\infty), (T_0, w, T_\infty)\}$$

- serieller Schedule $s' = T_2\ T_1\ T_3$:

$$s' = r_2(y)w_2(x)w_2(z)c_2r_1(y)w_1(y)w_1(x)c_1r_3(w)w_3(x)c_3$$

- Liest-von-Relation für Schedule s':

$$RF(s') = \{(T_0, y, T_1), (T_0, w, T_3), (T_0, y, T_2), (T_3, x, T_\infty), (T_1, y, T_\infty), (T_2, z, T_\infty), (T_0, w, T_\infty)\}$$

Beispiel CSR versus VSR IV

- da $RF(s) = RF(s')$ gilt, ist der Schedule s sichtserialisierbar
- daher: Konfliktserialisierbarkeit ist einschränkender als Sichtserialisierbarkeit

$$\neg(\textbf{CSR} \supset \textbf{VSR})$$

- Allgemein:

$$\textbf{CSR} \subset \textbf{VSR}$$

Transaktionsabbruch und Fehlersicherheit

Aus Sicht der Fehlersicherheit ist folgender Schedule s nicht akzeptabel:

s = $r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$

. . . aber serialisierbar in **VSR** *und* **CSR**!

Rücksetzbarkeit RC

- s heißt *rücksetzbar* (engl. *recoverable*), falls folgende Bedingung erfüllt ist:

$$(T_i \text{ liest von } T_j \text{ in } s) \wedge (c_i \in s) \Rightarrow (c_j \rightarrow_s c_i)$$

Beispiel Rücksetzbarkeit

$$s_1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)c_2w_1(z)c_1$$

In s_1 liest T_2 das Datenobjekt y von T_1, aber c_2 kommt vor c_1
$\rightsquigarrow$ s_1 ist nicht rücksetzbar

$$s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)c_1c_2$$

s_2 ist rücksetzbar

Aber: Probleme bei Abbruch von T_1 anstelle von c_1 (dirty read)!

Vermeidung kaskadierender Abbrüche

- Schedule s *vermeidet kaskadierende Abbrüche* (engl. *avoiding cascading aborts* **ACA**), falls folgende Bedingung erfüllt ist:

$$(T_i \text{ liest } x \text{ von } T_j \text{ in } s\ ) \Rightarrow (c_j \rightarrow_s r_i(x))$$

  $\rightsquigarrow$ eine Transaktion darf nur Daten lesen, die zuletzt von einer bereits abgeschlossenen Transaktion geschrieben wurden

Beispiel: Vermeidung kaskad. Abbrüche

- Schedule s_2 des letzten Beispiels gehört nicht in die Klasse **ACA**
- s_3 jedoch vermeidet kaskadierende Abbrüche:

$$s_3 = w_1(x)w_1(y)r_2(u)w_2(x)w_1(z)c_1r_2(y)w_2(y)c_2$$

- Daher gilt: $s_3 \in$ **ACA**

Probleme mit Before-Images

DB-Inhalt	Operation
$x = 1$ (Anfangswert)	
$x = 2$	w_1(x $\leftarrow$ 2) [$BF_{x,T_1} = 1$]
$x = 3$	w_2(x $\leftarrow$ 3) [$BF_{x,T_2} = 2$]
	a_1 Rücksetzen von $w_1(x \leftarrow 2)$ mit $BF_x := 1$. Überschreiben durch T_2 muss erhalten bleiben!
$x = 3$	
	a_2 Rücksetzen von $w_2(x \leftarrow 3)$ mit $BF_x :=$??

Striktheit ST

- Schedule s heißt *strikt* (engl. *strict*), falls folgende Bedingung gilt:

$$(w_j(x) \rightarrow_s p_i(x) \quad \wedge \quad j \neq i) \Rightarrow (a_j <_s p_i(x) \vee c_j <_s p_i(x), (p \in \{r, w\}))$$

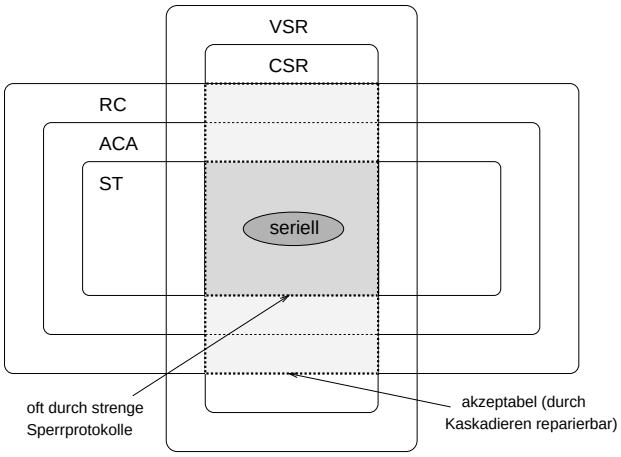
$\rightsquigarrow$ es darf kein „geschriebenes“ Objekt einer noch nicht beendeten Transaktion gelesen oder überschrieben werden

Beispiel Striktheit

- $s_3 \notin$ **ST**
- s_4 ist strikt, also $s_4 \in$ **ST**:

$$s_4 = w_1(x)w_1(y)r_2(u)w_1(z)c_1w_2(x)r_2(y)w_2(y)c_2$$

Zusammenhang zwischen den Klassen



Operationen: Benutzerkommandos

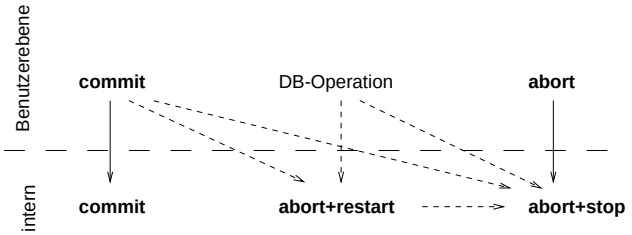
Benutzerkommandos, die den Abbruch einer Transaktion beeinflussen können:

- **commit** versucht ein Commit durchzuführen $\leadsto$ gelingt aber nicht immer (Integritätsverletzung) $\leadsto$ tatsächliches Commit ist erst duch die Erfolgsmeldung des DBMS garantiert
- **abort** erzwingt endgültigen Abbruch einer Transaktion
- andere Datenbank-Operationen, die zum Abbruch führen können: Division durch Null, Integritätsverletzungen

Operationen: Intern

- **commit** führt intern ein endgültiges Commit durch
- für **abort** zwei Varianten:
  - **abort+restart** bricht die Transaktion ab, aber versucht durch ein erneutes Starten, diese Transaktion zu einem erfolgreichen Ende zu führen (durch Eingriff des Schedulers; bei wiederholten erfolglosen **abort+restart** auch endgültiger Abbruch möglich)
  - **abort+stop** realisiert den endgültigen Abbruch (expliziter Benutzerwunsch oder bei nichtreparablen Integritätsverletzungen)

Operationen: Zusammenhang



Ausnutzung semantischer Informationen

- Bisher: Read/Write-Modell
  - low-level Operationen
  - keine spezielle Optimierung möglich
- Ausnutzung semantischer Informationen
- höhere Operationen
- Vertauschbarkeitsrelationen

Vertauschbarkeit von Operationen

1. Transaktionen $T_1,\ T_2$ lesen zwei gleiche oder verschiedene Datenobjekte:

$$r_i(x); r_j(y) \ \equiv \ r_j(y); r_i(x) \quad \text{falls} \quad i \neq j$$

2. Transaktion T_1 liest Wert, Transaktion T_2 schreibt anderen Wert:

$$r_i(x); w_j(y) \ \equiv \ w_j(y); r_i(x) \quad \text{falls} \quad i \neq j, x \neq y$$

3. Transaktionen $T_1,\ T_2$ schreiben zwei unterschiedliche Datenbankobjekte:

$$w_i(x); w_j(y) \ \equiv \ w_j(y); w_i(x) \quad \text{falls} \quad i \neq j, x \neq y$$

Neuer Serialisierbarkeitsbegriff

Ein Schedule s ist *konfliktserialisierbar* wenn er durch eine endliche Folge von erlaubten Vertauschungen $p; q \ \rightarrow \ q; p$ in einen seriellen Schedule überführbar ist.

Beispiel 1

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

- Anwendung der Regel (1):

$$s' = r_2(x) r_1(x) w_1(x) w_2(x)$$

- keine weiteren Vertauschungsregeln (Basisregeln) anwendbar
  $\leadsto$ Schedule s' ist kein serieller Schedule
  $\leadsto$ Schedule s ist nicht serialisierbar

Beispiel 2

$$s = r_1(x) r_2(x) w_1(x) w_2(y)$$

- Anwendung der Regeln (1) und (3):

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

- Anwendung der Regel (2):

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

- Schedule s'' ist serieller Schedule
  $\rightsquigarrow$ Schedule s serialisierbar

Vertauschbarkeitstabelle

$x \neq y$	$r_1(x)$	$r_1(y)$	$w_1(x)$	$w_1(y)$
$r_2(x)$	$\surd$	$\surd$	–	$\surd$
$r_2(y)$	$\surd$	$\surd$	$\surd$	–
$w_2(x)$	–	$\surd$	–	$\surd$
$w_2(y)$	$\surd$	–	$\surd$	–

Vereinfachte Vertauschbarkeitstabelle

	$r_1(x)$	$w_1(x)$
$r_2(x)$	$\surd$	–
$w_2(x)$	–	–

Beispielszenario

- Bank mit zwei Arten von Konten:
  - Konten für normale Buchungen, sowie
  - ein Konto, welches die Summe der Einlage bestimmter Konten enthält
- bei jeder Buchung muss Konto mit der Summe der Einlagen aktualisiert werden $\leadsto$
  - fast jede Transaktion muss auf das Summen-Objekt (Summen-Konto) zugreifen $\leadsto$ *Hot-Spot-Objekt*

Neue Operationen für Hot-Spot-Objekt

- $incr$ (Hinzuaddieren eines Wertes)
- $decr$ (Abziehen eines Wertes)

Vertauschbarkeitstabelle

	$r_1(x)$	$w_1(x)$	$incr_1(x)$	$decr_1(x)$
$r_2(x)$	$\surd$	–	–	–
$w_2(x)$	–	–	–	–
$incr_2(x)$	–	–	$\surd$	$\surd$
$decr_2(x)$	–	–	$\surd$	$\surd$

Verträglichkeit der $incr$- und $decr$-Operationen!

Verträglichkeit der Operation

$$incr_1(5); decr_2(4) \equiv decr_2(4); incr_1(5)$$

$\Rightarrow$ vertauschbar!

$\rightarrow$ Ausführung der Operationen $incr$ und $decr$ ist beliebig

Kompensierende Aktionen

Operation	Kompensation
incr(x)	decr(x)
decr(x)	incr(x)

Einsatz kompensierender Aktionen

Abbruch einer Transaktion $\rightarrow$ statt des **aborts** kompensierende Aktionen ausführen:

- statt: $incr_i(x)a_i$
- jetzt: $incr_i(x)$ ... $decr_i(x)c_i$

Einsatz Termersetzungssemantik

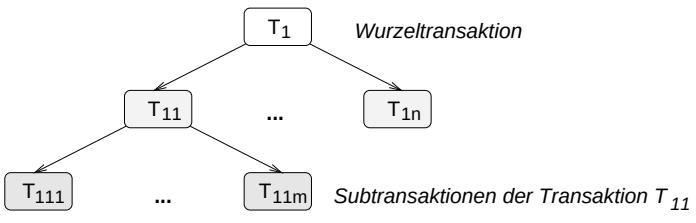
- Termersetzung auf Schedules definiert Serialisierbarkeit, da fehlende Konflikte durch Vertauschungsersetzungen erklärt werden können
- Termersetzung erweitert um Kompensationsregeln vereinfacht das Rücksetzen auf Hot-Spot-Objekten

Erweiterte Transaktionsmodelle

Prinzipien anhand zweier Modelle:

- Geschachtelte Transaktionen (engl. *nested transactions*):
  hierarchische Ansammlung von Vater- / Sohntransaktionen
- *Sagas*:
  Zwischenergebnisse bereits durch ein Commit anderen Transaktionen verfügbar, aber trotzdem bei einem späteren Abbruch wieder rückgängig gemacht

Transaktionsbaum



ACID-Eigenschaften von Tx-Bäumen

- *Isolation*:
  Ergebnisse einer Subtransaktion an die Vatertransaktion weitergeleitet, nicht sichtbar für andere nebenläufige Transaktionen
- *Atomarität*:
  entweder alle Transaktionen des Transaktionsbaums enden erfolgreich oder brechen gemeinsam ab

Geschlossen geschachtelte Transaktionen

- Weitergabe der Sperren einer Subtransaktion an die Vatertransaktion, Sperren werden also in der Hierarchie nach „oben“ (in Richtung der Wurzel) weitergereicht
- Ergebnisse der geschlossen geschachtelten Transaktion werden erst mit dem Commit der Wurzeltransaktion freigegegeben

Geschlossen gesch. Transaktionen (II)

- Atomarität:
  1. Abbruch einer Vatertransaktion erzwingt Abbruch aller Subtransaktionen
  2. Transaktion des Transaktionsbaumes kann nur erfolgreich enden, wenn alle Subtransaktionen erfolgreich waren
  3. Abbruch einer Subtransaktion führt zum Abbruch der Vatertransaktion

Offen geschachtelte Transaktionen ONT

- Ergebnisse werden bereits bei Commit der Subtransaktion freigegeben
- Atomarität offen geschachtelter Transaktionen

  Geg.: zwei Transaktionen t_i und t_j wobei t_j Sohn von t_i ist

  1. **abort**(t_i) erzwingt einen **abort**(t_j)
  2. **commit**(t_i) ist nur möglich nach einem **commit**(t_j)
- Klassen von Subtransaktionen:
  vitale / nicht-vitale Transaktionen, Ersatztransaktionen

Reaktion bei abort(t_j) bei ONT

1. Ignorieren (**ignore**) für „nicht lebenswichtige“ (nicht-vitale) Subtransaktionen
2. Erneutes Starten der abgebrochenen Subtransaktion: **retry** t_j, evtl. in Abhängigkeit von der Ursache des Abbruchs
3. Versuch der Ausführung (**try**) einer *Ersatztransaktion* (engl. *contingency transaction*) (alternatives Ausführen von Ersatztransaktionen im Falle eines Abbruchs oder der Nichtausführbarkeit einer Transaktion)
4. Abbruch des Vaters: **abort**(t_i)

Sagas

Bestandteile einer Saga:

- Eine Menge von Transaktionen T
- Für jede Transaktion $T_i \in T$ eine *kompensierende Transaktion* C_i, die den Zustand vor der Transaktion T_i semantisch rückgängig macht

Saga $=$ spezielle ONT der Tiefe 1

Erlaubte Ausführungshistorien einer Saga

- Erlaubte Ausführungen einer Saga:

  $T_1 T_2 \ldots T_n$ (korrekte Ausführung)

  oder

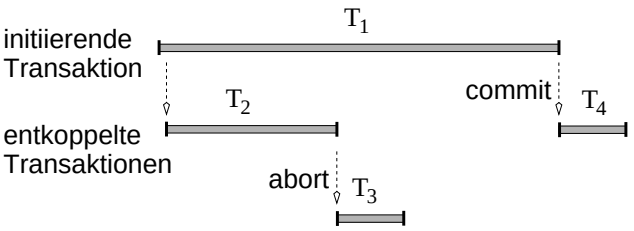
  $T_1 T_2 \ldots, T_i C_i C_{i-1} \ldots C_2 C_1$ (kontrollierter Abbruch)

- zweiter Fall tritt ein, wenn die Subtransaktion $T_{i+1}, 1 \leq i < n$, abgebrochen wurde

Beispiel für Ablauf einer Saga

1. Transaktion T_1: Hebe 10 Euro ab
2. Transaktion T_2: Kaufe Gegenstand
3. **abort** von T_2
4. Kompensierende Transaktion C_1: Zahle 10 Euro ein

Entkoppelte Subtransaktionen



Entkoppelte Subtransaktionen (II)

- Subtransaktionen können außerhalb des zeitlichen Rahmens der Wurzeltransaktionen laufen
- eine Subtransaktion kann ein Commit machen, auch wenn Vatertransaktion abbricht
- neben expliziten Aufruf einer Subtransaktion kann Subtransaktion auch durch das Commit oder Abort einer anderen Transaktion aktiviert werden
  - entspricht Spezialfällen des Regelmodus **detached but causally dependent detached but causally dependent** in aktiven Datenbanken
  - spezielle Abhängigkeiten sind **parallel**, **sequential** (Start der Transaktion nach erfolgreichem Ende der triggernden Transaktion) und **exclusive** (Start nur nach Abbruch der triggernden Transaktion)