Transaktionsverwaltung

1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 –
Informationssysteme
und Datenbanken

RWTH AACHEN
UNIVERSITY

Ziele dieses Kapitels

Ein DBMS hat die (Angriffs- und Missbrauchs-)Sicherheit und die Korrektheit des Datenbankzustandes unter realen Benutzungsbedingungen zu wahren.

Mit Techniken auf Basis der **Anfragebearbeitung (Views)** unterstützt das DBMS:

- **Integrität** (Integrity Constraints)
  Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
- **Datensicherheit** (Security, Access Rights)
  Schutz vor Zugriffen und Änderungen durch *unberechtigte* Benutzer.

Das Konzept der **Transaktionen** sichert

- **Synchronisation** (Concurrency Control)
  Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer
- **Fehlertoleranz** (Recovery)
  Schutz vor Datenverlust durch technische Fehler (z.B. Programmfehler, Systemabsturz), ohne Unterbrechung des laufenden Betriebs.

Transaktionsverwaltung

1. **Das Transaktionskonzept**
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

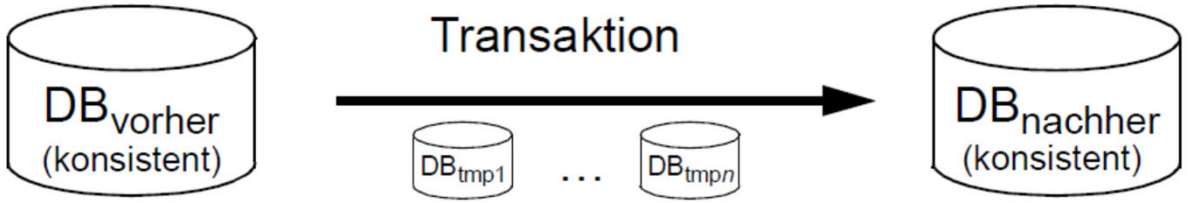
Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Motivation - Transaktionsmanagement

Prinzip	Annahme bisher	In Wahrheit
Isolation	Nur ein Nutzer greift auf die Datenbank lesend und schreibend zu.	Viele Nutzer und Anwendungen lesen und schreiben gleichzeitig.
Atomarität	Anfragen und Updates bestehen aus einer einzigen, atomaren Aktion. DBMS können nicht mitten in dieser Aktion ausfallen.	Auch einfache Anfragen bestehen oft aus mehreren Teilschritten. DBMS können jederzeit ausfallen.

Transaktionen

Transaktionen (TAs) sind die Einheiten *integritätserhaltender Zustandsänderungen* einer Datenbank:



- Eine Transaktion ist „eine Folge von Operationen […], welche eine gegebene Datenbank in ununterbrechbarer Weise von einem konsistenten Zustand in einen [anderen] (nicht notwendig verschiedenen) konsistenten Zustand überführt.“ [Vossen, 2008; S. 638]
- Integritätsaspekte:
  - Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
  - Ablaufintegrität: Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten vermeiden

Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

- Möglicher Bearbeitungsplan:
  1. Erniedrige Stand von “Huber” um 200
  2. Erhöhe Stand von “Meier” um 200

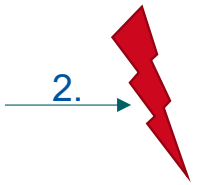
- Möglicher Ablauf:

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.500,00

1.

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.300,00

2. Systemabsturz



Wichtig: Inkonsistenter Datenbankzustand darf nicht entstehen bzw. nicht dauerhaft bestehen bleiben

ACID-Prinzip korrekter Transaktionsabwicklung

Eine grundlegende Charakterisierung von Transaktionsanforderungen ist durch das **ACID**-Prinzip (Härder/Reuter 1983) gegeben:

- **Atomicity** (Atomarität, “alles-oder-nichts”-Prinzip)
  - Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zu tragen
- **Consistency** (Konsistenz, Integritätserhaltung)
  - Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt
- **Isolation** (Isolation, logischer Einbenutzerbetrieb)
  - Innerhalb einer Transaktion nimmt ein Benutzer Änderungen durch andere Benutzer nicht wahr
- **Durability** (Dauerhaftigkeit, Persistenz)
  - Der Effekt einer abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten

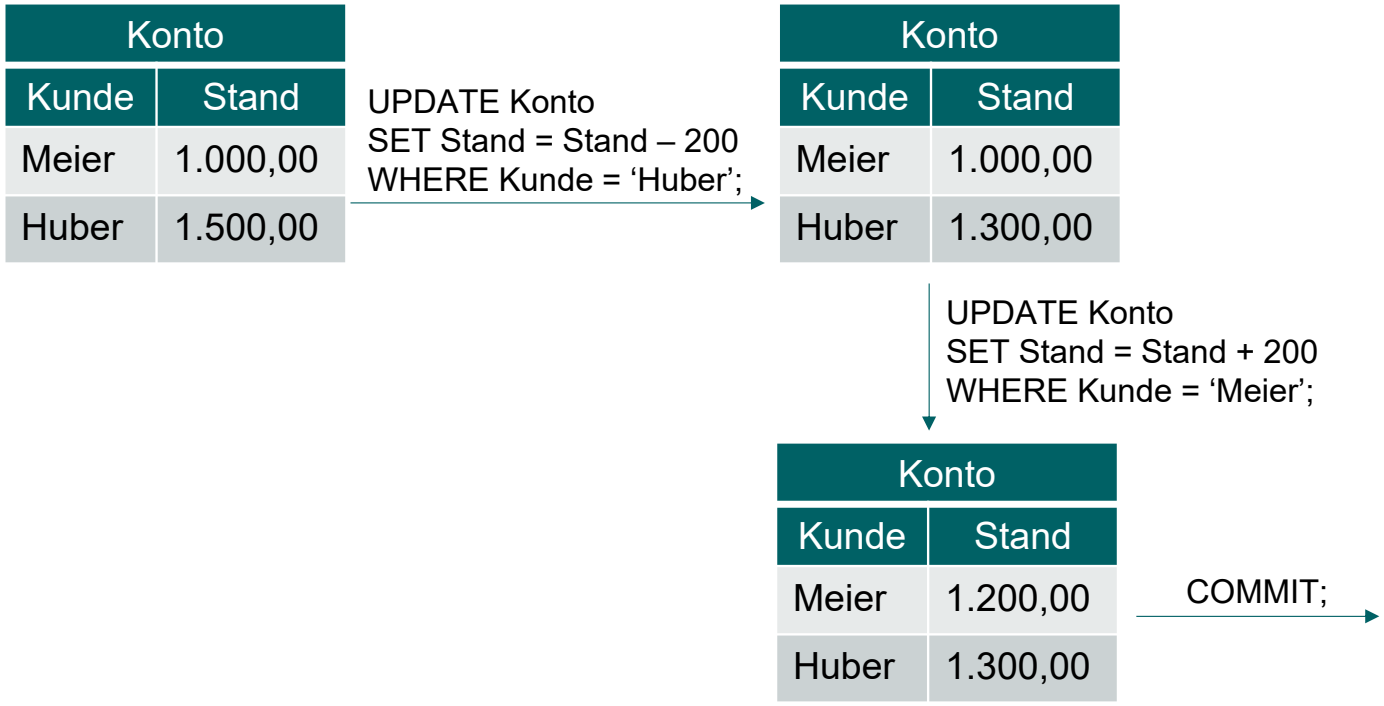
Jede Transaktion muss in endlicher Zeit ausgeführt werden, so dass die ACID Eigenschaften erhalten bleiben

Steuerung von Transaktionen

- **begin of transaction**
  - Beginn der Befehlsfolge einer neuen Transaktion
  - SQL: **begin transaction,** o.ä., oft auch implizit (hier auch: **BOT = Beginn of Transaction)**
- **end of transaction, commit**
  - Bestätigung der Transaktion durch den Benutzer
    - Die Änderungen seit Beginn der Transaktion werden endgültig bestätigt
    - Der jetzt erreichte Zustand soll dauerhaft gespeichert werden
    - Der Zustand wird durch den Benutzer für konsistent erklärt
  - SQL: **commit work** oder nur **commit**
- **abort transaction**
  - Abbruch der Transaktion durch das Programm bzw. den Benutzer (Rücksetzen, ABORT)
    - Die Änderungen der Transaktion werden zurückgesetzt
    - Der ursprüngliche Zustand vor der Transaktion wird wiederhergestellt
  - SQL: **rollback work** oder nur **rollback**

Steuerung von Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

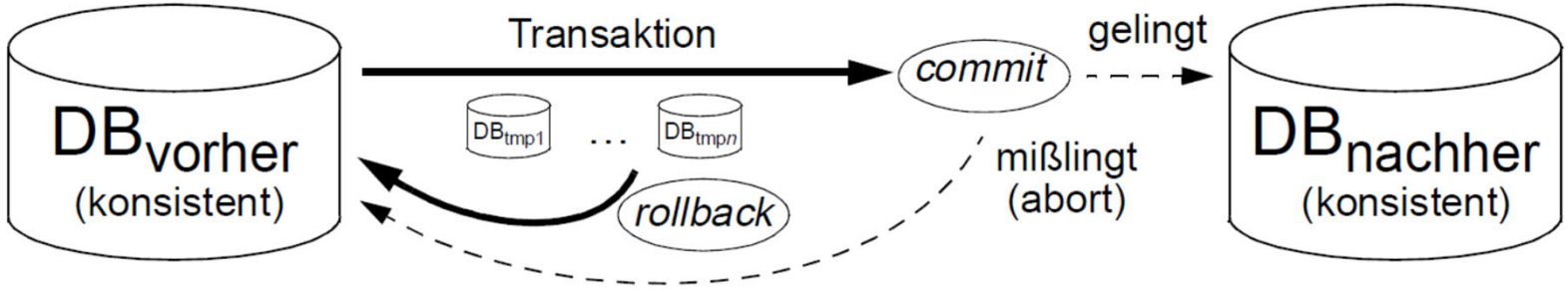


Ausführung einer Transaktion (commit)

Ein **COMMIT** kann …

- *gelingen*: Der neue Zustand wird dauerhaft gespeichert, oder
- *scheitern*: Der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann u.a. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.

Schematischer Ablauf:



Transaktionsverwaltung

1. Das Transaktionskonzept
2. **Synchronisation (Concurrency Control)**
3. Protokolle zur Synchronisation (Scheduler)
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Transaktionsmodell: Abstraktion auf Lese- und Schreiboperationen

- Eine **Transaktion** (t_i), d.h. ein DB-Programm während der Ausführung, wird vereinfacht betrachtet als ein Programm, das nur aus Lese- und Schreiboperationen auf einer Datenbank besteht
  - Leseoperation **read(x)** liest DB-Objekt x
  - Schreiboperation **write(x)** schreibt DB-Objekt x
- Die Beschränkung auf Lese- und Schreib-Operationen ist aus der Realität abstrahiert (z.B. Weiterverarbeitung von Daten) und dient zur vereinfachten Analyse
- Semantische Information über ein DB-Programm werden nicht mehr betrachtet
- Die Abstraktion ist hinreichend für viele praktische Anwendungen

Probleme beim Ausführen von Transaktionen im Mehrbenutzerbetrieb

Nach dem ACID-Prinzip “Isolation” sollen Transaktionen im logischen Einbenutzerbetrieb ablaufen, d.h. innerhalb einer Transaktion ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen.

Ist die Isolierung der Transaktionen in einem Datenbanksystem nicht sichergestellt, können verschiedene **Anomalien/Synchronisationsprobleme** auftreten:

- Verlorengegangene Änderungen (**Lost Updates**)
- Zugriff auf “schmutzige” Daten (**Dirty Read**, **Dirty Write**)
- Nicht-reproduzierbares Lesen
- Phantomproblem

Probleme beim Ausführung von Transaktionen

Die o.a. Anomalien werden unter anderem am Beispiel der folgenden Flugdatenbank erläutert:

Passagiere		
Flug#	Name	Platz
LH745	Müller	3A
LH745	Meier	6D
LH745	Huber	5C
BA932	Schmidt	9F
BA932	Huber	5C

Fluginfo	
Flug#	AnzPass
LH745	3
BA932	3

Verlorengegangene Änderungen (Lost Update)

Änderungen einer Transaktion können durch Änderungen anderer Transaktionen überschrieben werden und dadurch verloren gehen.

- Beispiel:
  Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus:
  UPDATE Fluginfo SET AnzPass = AnzPass + 1 WHERE Flug# = ‘BA932’;
- Möglicher Ablauf:

T1	T2
Read Fluginfo[Anzpass] → x	
	Read Fluginfo[Anzpass] → y
	$y := y + 1$
	Write y → Fluginfo[Anzpass]
x:= $x + 1$	
Write x → Fluginfo[Anzpass]	

- Ergebnis:
  Beide Transaktionen haben die Anzahl der Passagiere (für denselben Flug) jeweils um eins erhöht. Obwohl zwei Erhöhungen stattgefunden haben, ist in der Datenbank nur die Erhöhung von T1 wirksam. Die Änderung von T2 ist verlorengegangen.

Zugriff auf “schmutzige” Daten (Dirty Read, Dirty Write)

Als “schmutzige” Daten bezeichnet man Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden.

- Beispiel:
  - T1 erhöht das Gehalt um 500 Euro, wird aber später abgebrochen
  - T2 erhöht das Gehalt um 5% und wird erfolgreich abgeschlossen
- Möglicher Ablauf:

T1	T2
UPDATE …. $Gehalt := Gehalt + 500$;	
	UPDATE …. $Gehalt := Gehalt * 1.05$:
	COMMIT;
ROLLBACK;	

- Ergebnis:
  - Der Abbruch der ändernden Transaktion T1 macht die geänderten Werte ungültig, sie werden zurückgesetzt. Die Transaktion T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*).
  - Verstoß gegen *ACID*: Dieser Ablauf verursacht einen dauerhaften fehlerhaften Datenbankzustand (*Consistency*), bzw. T2 muss nach COMMIT zurückgesetzt werden (*Durability*).

Nicht-reproduzierbares Lesen

Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts.

- Beispiel:
  - T1: Gib die Fluginfo für alle Flüge und die Anzahl der Passagiere für BA932 aus
  - T2: Buche den Platz 3F auf dem Flug BA932 für Passagier Meier
- Möglicher Ablauf:

T1	T2
SELECT * FROM Fluginfo	
	INSERT INTO Passagiere(*) VALUES (‚BA932‘,‘Meier‘,‘3F‘);
	UPDATE Fluginfo SET AnzPass= AnzPAss +1 WHERE Flug# = ‚BA932‘;
	COMMIT;
SELECT AnzPass FROM Fluginfo	
Write x → Fluginfo[Anzpass] WHERE Flug# = ‚BA932‘;	

- Ergebnis:
  Die Anweisungen A1.1 und A1.2 liefern ein unterschiedliches Ergebnis für den Flug BA932, obwohl die Transaktion T1 den Datenbankzustand nicht geändert hat.

Phantomproblem

Das Phantomproblem ist ein nicht-reproduzierbares Lesen in Verbindung mit Aggregatfunktionen.

- Beispiel:
  - AnzPass werde jetzt durch COUNT(*) berechnet und nicht mehr in FlugInfo gespeichert
  - T1: Drucke die Passagierliste sowie die Fluginfo für den Flug LH745
  - T2: Buche den Platz 7D auf dem Flug LH745 für Phantomas
- Möglicher Ablauf:

T1	T2
SELECT * FROM Passagiere WHERE Flug# = ‘LH745‘;	
	INSERT INTO Passagiere(*) VALUES (‘LH745‘,‘Phantomas‘,‘7D‘);
	COMMIT;
SELECT AnzPass = COUNT(*) FROM Passagiere WHERE Flug# = ‘LH745‘;	

- Ergebnis:
  Für die Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere schon berücksichtigt ist.

- Grundannahme: Korrektheit
  - Jede Transaktion, isoliert ausgeführt auf einem konsistenten Zustand der Datenbank, hinterlässt die Datenbank wiederum in einem konsistenten Zustand.
  - Lösung aller obigen Probleme: Alle Transaktionen seriell ausführen.
  - Aber: Parallele Ausführung bietet Effizienzvorteile:
    - „Long-Transactions“ über mehrere Stunden hinweg
    - Cache ausnutzen
- Deshalb: Korrekte parallele Pläne (Schedules) finden
  - Korrekt = Serialisierbar

Transaktionen: Definition

- **Annahme**: eine Datenbank $\mathrm{DB} = \{x, y, z, \dots\}$ ist eine Menge von Objekten auf die nur mittels Schreib- und Leseoperationen zugegriffen wird
- Eine **Transaktion** t ist definiert als eine endliche Folge von Schreib- und Leseoperationen:

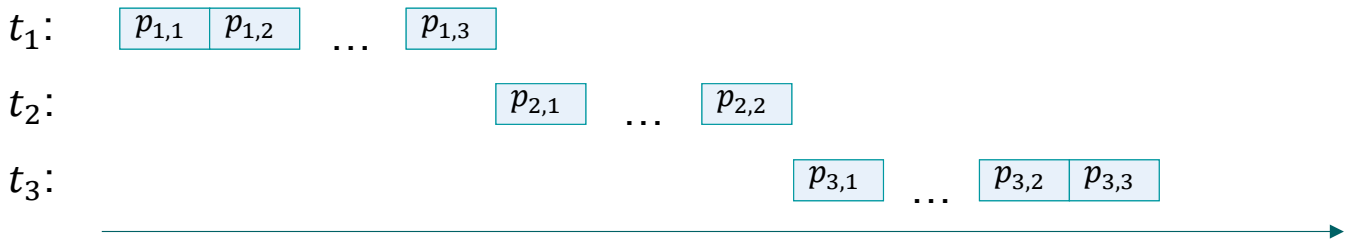
$$t = p_1, \dots, p_n$$

  - mit $n < \infty$
  - $p_i \in \{r(x) \mid x \in \mathrm{DB}\} \cup \{w(x) \mid x \in \mathrm{DB}\}$ für $1 \leq i \leq n$

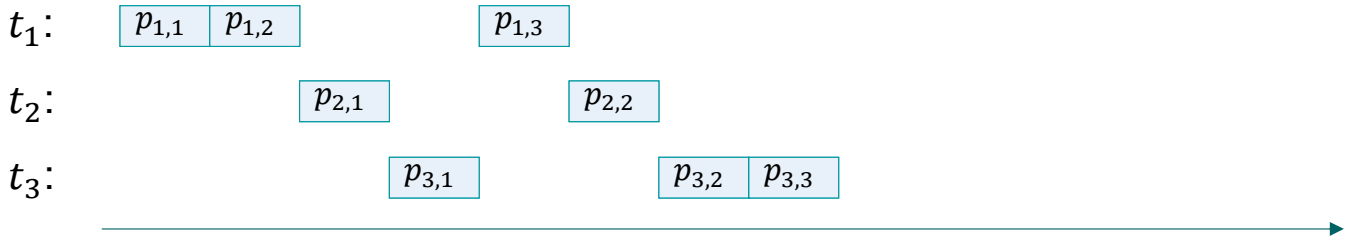
- Indizes kennzeichnen verschiedene (nebenläufige) Transaktionen, z.B.:
  - $t_1 = r_1(x)\, r_1(y)\, r_1(z)\, w_1(z)\, w_1(x)$
  - $t_2 = r_2(x)\, r_2(z)\, w_2(x)\, w_2(y)$

Bearbeitung mehrerer Transaktionen

- Wie können mehrere nebenläufige Transaktionen effizient ausgeführt werden?
- Serieller Ablaufplan (Schedule):



- Verzahnter Schedule:



Bearbeitung mehrerer Transaktionen

- Wie können mehrere nebenläufige Transaktionen effizient ausgeführt werden?

Serieller Ablaufplan (Schedule):

Schritt	t_1	t_2	t_3
1	$p_{1,1}$		
2	$p_{1,2}$		
3	$p_{1,3}$		
4		$p_{2,1}$	
5		$p_{2,2}$	
6		$p_{2,3}$	
7			$p_{3,1}$
8			$p_{3,2}$
9			$p_{3,3}$

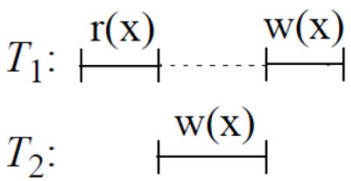
Verzahnter Schedule:

Schritt	t_1	t_2	t_3
1	$p_{1,1}$		
2	$p_{1,2}$		
3		$p_{2,1}$	
4			$p_{3,1}$
5	$p_{1,3}$		
6		$p_{2,2}$	
7			$p_{3,2}$
8		$p_{2,3}$	
9			$p_{3,3}$

Beobachtung:
mögliche Problemursache Nicht-serialisierbare Schedules

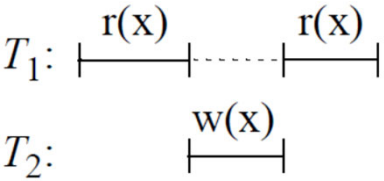
- Man hat früh beobachtet, dass viele Probleme offenbar dann auftreten, wenn Transaktionen sich beinflussen, dh., Schedules „nicht serialisierbar“ sind, z.B.

$s = r_1(x)\, w_2(x)\, w_1(x)$



Lost Update

$s = r_1(x)\, w_2(x)\, r_1(x)$



Non-repeatable Read

- Erster Lösungsansatz zur Formalisierung des Synchronisationsproblems:

Nur „serialisierbare“ Schedules dürfen zugelassen werden.

(Definition im folgenden)

Serielle und Serialisierbare Schedules

Serieller Schedule

Schritt	t_1	t_2
1	BOT	
2	read(A)	
3	write(A)	
4	read(B)	
5	write(B)	
6	commit	
7		BOT
8		read(C)
9		write(C)
10		read(A)
11		write(A)
12		commit

Serialisierbarer Schedule

Schritt	t_1	t_2
1	BOT	
2	read(A)	
3		BOT
4		read(C)
5	write(A)	
6		write(C)
7	read(B)	
8	write(B)	
9	commit	
10		read(A)
11		write(A)
12		commit

Schedules

Serialisierbar?

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Serialisierbar?

Abstrakt

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Gleicher Ablauf, konkret

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Serialisierbar?

Original

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Serialisierbar?

Original

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4		BOT		
5		read(A): A=A*1.1		
6		write(A)	121	
7		read(B); B=B+20		
8		write(B)		120
9		commit		
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		121	144

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Serialisierbar?

Serialisierbar?
Nein, denn Effekt entspricht weder dem seriellen Schedule $t_1; t_3$ noch dem seriellen Schedule $t_3; t_1$

Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	

Noch mal die beiden Schedules. Was fällt auf?

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Noch mal die beiden Schedules. Was fällt auf?

Schedule: $t_1; t_3$

Schritt	t_1	t_3	A	B
1	BOT		100	100
2	read(A); A=A+10			
3	write(A)		110	
4	read(B); B = B * 1.2			
5	write(B)			120
6	commit			
7		BOT		
8		read(A): A=A*1.1	121	120
9		write(A)		
10		read(B); B=B+20		
11		write(B)		140
12		commit	121	140

Schedule: $t_3; t_1$

Schritt	t_1	t_3	A	B
1		BOT	100	100
2		read(A): A=A*1.1		
3		write(A)	110	
4		read(B); B=B+20		
5		write(B)		120
6		commit		
7	BOT			
8	read(A); A=A+10			120
9	write(A)		120	
10	read(B); B = B * 1.2			
11	write(B)			144
12	commit		120	144

Ergebnisse von $t_1; t_3 \neq t_3; t_1$. Macht das was?

Transaktionsverwaltung

1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. **Protokolle zur Synchronisation (Scheduler)**
4. Fehlertoleranz (Recovery)
5. Datensicherheit

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Formaler Korrektheitsbegriff 1: Serialisierbarkeit von Schedules

- Ein **Schedule** für eine Menge $\{t_1, \ldots, t_n\}$ von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der t_i entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.

- Ein **serieller Schedule** ist ein Schedule s von $\{t_1, \ldots, t_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden. Serielle Schedules erfüllen offensichtlich die Isolationsbedingung des ACID-Prinzips.

- Ein Schedule s von $\{t_1, \ldots, t_n\}$ ist **serialisierbar**, wenn er „äquivalent“ ist zu einem (beliebigen) seriellen Schedule (Permutation von $\{t_1, \ldots, t_n\}$).

Um Serialisierbarkeit sicherzustellen, brauchen wir einen **Äquivalenzbegriff**, der
  - möglichst viel Parallelität gestattet (Mehrbenutzerfähigkeit!),
  - auch bei großen Nutzerzahlen effizient zu testen, möglichst automatisch zu garantieren
  - zu jedem Zeitpunkt der Lebensdauer eines DBMS anwendbar ist.
- Wir werden das Konzept „**Konfliktserialisierbarkeit“** im folgenden erarbeiten!

Schedule: Definition

- Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.
- Das **Shuffle-Produkt** $shuffle(T)$ von T bezeichnet die Menge aller Folgen von Aktionen, in welchen die gegebenen Transaktionen $t_1, \ldots, t_n$ als Teilfolgen auftreten und keine weiteren Aktionen vorkommen.

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$

- Dann gilt für die folgende Folge von Aktionen:
  - $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z) \in shuffle(T)$

Schedule: Definition

Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.

- Ein **vollständiger Schedule** s für T ist eine Folge $s \in shuffle(T)$ mit den zusätzlichen Pseudoaktionen c_i) commit) und a_i (abort) für jedes $t_i \in T$ entsprechend den folgenden Regeln:
  - $c_i \in s \Leftrightarrow a_i \notin s$ für alle $1 \leq i \leq n$
  - c_i oder a_i steht in s hinter der letzten Aktion von t_i für alle $1 \leq i \leq n$
- Die **Menge aller vollständigen Schedules** wird als $shuffle_{ac}(T)$ bezeichnet.
- Erweiterung der Definition: Ein **Schedule** ist der Präfix eines Elementes von $shuffle_{ac}(T)$

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\, w_1(x)\, r_1(y)\, w_1(y)$
  - $t_2 = r_2(z)\, w_2(x)\, w_2\ (z)$
  - $t_3 = r_3(x)\, r_3(y)\, w_3(z)$

- Dann gilt für das folgende Schedule:
  - $s_1 = r_1(x)\, r_2(z)\, r_3(x)\, w_2(x)\, w_1(x)\, r_3(y)\, r_1(y)\, w_1(y)\, w_2(z)\, w_3(z) \in shuffle(T)$
  - $s_2 = s_1\, c_1\, c_2\, c_3 \in shuffle_{ac}(T)$
  - $s_3 = r_1(x)\, r_2(z)\, r_3(x)$ ist ein Schedule
  - $s_4 = s_1\, c_1$ ist ein Schedule

Schedule: Definition

- Sei $T = \{t_1, \ldots, t_n\}$ eine endliche Menge von Transaktionen.
- Ein vollständiger Schedule $s \in shuffle_{ac}(T)$ ist **seriell**, wenn für jeweils zwei t_i und t_j $(i \neq j)$ alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$
- Dann gilt für die folgenden Schedules:
  - $s_5 = t_1\ c_1\ t_3\ a_3\ t_2\ c_2$ ist ein serieller Schedule

Notationen für Transaktionszustands-Klassen

Sei s ein Schedule. Dann definieren wir die folgenden Mengen für s:

- Die Menge aller Aktionen:
  - $op(s) = \text{Menge aller Aktionen in } s$
- Die Menge aller Transaktionen:
  - $trans(s) = \{t_i \mid s \text{ enthält Aktionen aus } t_i\}$
- Die Menge aller bestätigten Transaktionen:
  - $commit(s) = \{t_i \mid t_i \in trans(s) \wedge c_i \in s\}$
- Die Menge aller abgebrochenen Transaktionen:
  - $abort(s) = \{t_i \mid t_i \in trans(s) \wedge a_i \in s\}$
- Die Menge aller aktiven Transaktionen:
  - $active(s) = trans(s) - \big(commit(s) \cup abort(s)\big)$

Notationen für Transaktionszustands-Klassen

Sei s ein Schedule. Dann definieren wir die folgenden Mengen für s:

- Die Menge aller Aktionen:
  - $op(s) = \text{Menge aller Aktionen in } s$
- Die Menge aller Transaktionen:
  - $trans(s) = \{t_i \mid s \text{ enthält Aktionen aus } t_i\}$
- Die Menge aller bestätigten Transaktionen:
  - $commit(s) = \{t_i \mid t_i \in trans(s) \wedge c_i \in s\}$
- Die Menge aller abgebrochenen Transaktionen:
  - $abort(s) = \{t_i \mid t_i \in trans(s) \wedge a_i \in s\}$
- Die Menge aller aktiven Transaktionen:
  - $active(s) = trans(s) - \big(commit(s) \cup abort(s)\big)$

Beispiel:

- Seien die folgenden Transaktionen $T = \{t_1, t_2, t_3\}$ wie folgt gegeben:
  - $t_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)$
  - $t_2 = r_2(z)\ w_2(x)\ w_2\ (z)$
  - $t_3 = r_3(x)\ r_3(y)\ w_3(z)$
- Dann gilt für die folgenden Schedules:
  - $s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z) \in shuffle(T)$
  - $s_2 = s_1\ c_1\ c_2\ c_3 \in shuffle_{ac}(T)$
  - $s_3 = r_1(x)\ r_2(z)\ r_3(x)$ ist ein Schedule
  - $s_4 = s_1\ c_1$ ist ein Schedule
  - $s_5 = t_1\ c_1\ t_3\ a_3\ t_2\ c_2$ ist ein serieller Schedule

$$
\begin{aligned}
trans(s_2) &= \{t_1, t_2, t_3\} \\
trans(s_4) &= \{t_1, t_2, t_3\} \\
commit(s_4) &= \{t_1\} \\
commit(s_5) &= \{t_1, t_2\} \\
abort(s_4) &= \emptyset \\
abort(s_5) &= \{t_3\} \\
active(s_2) &= \emptyset \\
active(s_4) &= \{t_2, t_3\}
\end{aligned}
$$

Wann sind Operationen zweier Transaktionen in Konflikt?

Gegeben Transaktionen t_i und t_k
- $r_i(x)$ und $r_k(x)$ stehen nicht in Konflikt
- $r_i(x)$ und $w_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $w_i(x)$ und $r_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $w_i(x)$ und $w_k(y)$ stehen nicht in Konflikt (falls $x \neq y$)
- $r_i(x)$ und $w_k(x)$ stehen in Konflikt
- $w_i(x)$ und $r_k(x)$ stehen in Konflikt

Zusammengefasst: Konflikt herrscht falls zwei Aktionen
- das gleiche Datenbankelement betreffen,
- und mindestens eine der beiden Aktionen ein *write* ist.

Konfliktmenge

- Sei s ein Schedule und $t, t' \in trans(s)$ zwei Transaktionen mit $t \neq t'$.
- Zwei Datenoperationen $p \in t$ und $q \in t'$ in s stehen in **Konflikt**, falls sie auf demselben Objekt arbeiten und mindestens eine Datenoperation eine Schreiboperation ist.
- Die **Konfliktmenge** von Schedule s ist definiert als:

$$C(s) = \{(p, q) \mid p, q \text{ in } s \text{ stehen in Konflikt und } p \text{ steht vor } q \text{ in } s\}$$

- Die Elemente der **Konfliktmenge** heißen **Konfliktbeziehungen**.
- Die bereinigte Konfliktmenge $conf(s)$ bezeichnet im Folgenden die Menge der Konfliktmenge eines Schedules s, *bereinigt von abgebrochenen Transaktionen*.
  - Randbemerkung: Wir betrachten hier also nur laufende und erfolgreich abgeschlossene Transaktionen. *Dirty Read-Probleme* können in diesem Modell NICHT formuliert werden, weil sie ja nur Abbruch von Transaktionen entstehen.

Konfliktmenge: Beispiel

- Sei der folgende Schedule gegeben:

$$s = w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(y)\ w_3(x)\ w_3(y)\ c_1\ a_2\ c_3$$

- Dann sind die Konfliktmengen wie folgt gegeben:

$$C(s) = \left\{ \begin{matrix} (w_1(x), r_2(x)), & (w_1(x), w_3(x)), & (r_2(x), w_3(x)), \\ (w_2(y), r_1(y)), & (w_2(y), w_1(y)), & (w_2(y), w_3(y)), \\ (r_1(y), w_3(y)), & (w_1(y), w_3(y)) & \end{matrix} \right\}$$

t_2 wird abgebrochen, daher ist die bereinigte Konfliktmenge:

$$conf(s) = \{(w_1(x), w_3(x)), \quad (r_1(y), w_3(y)), \quad (w_1(y), w_3(y))\}$$

Konfliktäquivalenz

- Zwei Schedules s und s' heißen **konfliktäquivalent**, bezeichnet mit $s \approx_c s'$, falls folgendes gilt:
  - Die Menge der Aktionen sind gleich, d.h.:
    - $op(s) = op(s')$
  - Die bereinigten Konfliktmengen sind gleich, d.h.:
    - $conf(s) = conf(s')$

- Konfliktäquivalenz kann wie folgt überprüft werden:
  - Für zwei gegebene Schedules (mit derselben Menge von Operationen), bestimme die Menge der bereinigten Konfliktbeziehungen und überprüfe sie auf Gleichheit.

Konfliktäquivalenz: Beispiel

- Seien die folgenden Schedules gegeben:

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

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

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

- Die beiden Schedules sind konfliktäquivalent, d.h. es gilt $s \approx_c s'$.

Konfliktserialisierbarkeit

- Ein vollständiger Schedule s heißt **konfliktserialisierbar**, falls ein serieller Schedule s' existiert mit $s \approx_c s'$.
- Die **Klasse aller konfliktserialisierbaren Schedules** bezeichnen wir mit *CSR*.

- Beispiele:
  - $s = r_2(y)\ w_1(x)\ w_1(y)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - $s' = r_1(x)\ r_2(x)\ w_2(y)\ c_2\ w_1(x)\ c_1 \in CSR$

Betrachte:

- $conf(s) = \{(r_2(y), w_1(y)), (w_1(x), w_2(y))\}$
- $conf(s') = \{(r_2(x), w_1(x))\}$

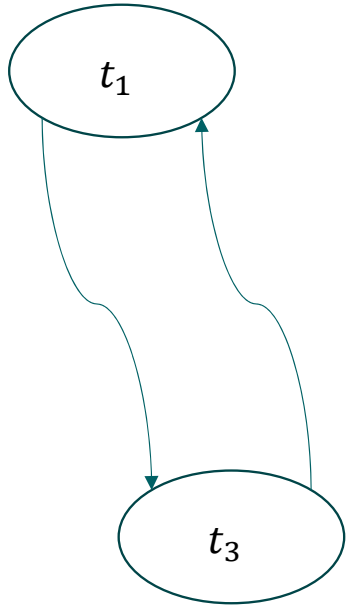
In der Konfliktmenge von s ist ein Zyklus!

- Ob ein Schedule zur Klasse der konfliktserialisierbaren Schedules gehört kann einfach geprüft werden.

Serialisierbar?

Serialisierbar?
Nein, denn Effekt entspricht weder dem seriellen Schedule $t_1; t_3$ noch dem seriellen Schedule $t_3; t_1$

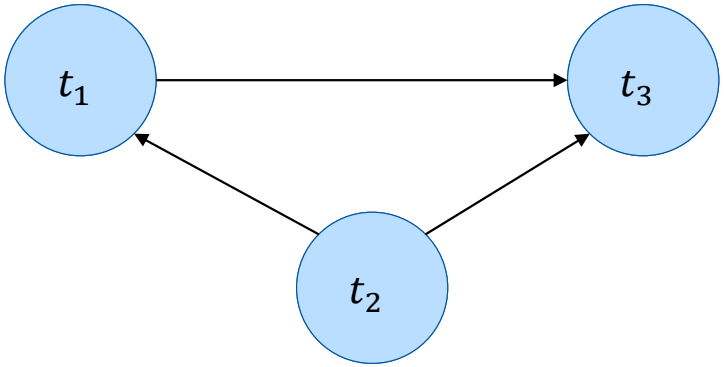
Schritt	t_1	t_3
1	BOT	
2	read(A)	
3	write(A)	
4		BOT
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		commit
10	read(B)	
11	write(B)	
12	commit	



Konfliktgraph

- Sei s ein Schedule. Der **Konfliktgraph** $G(s) = (V, E)$ von s ist wie folgt definiert:
  - Kontenmenge $V = commit(s)$
  - Kantenmenge $E = \{(t, t') \mid t \neq t' \wedge \exists p \in t, \exists q \in t' : (p, q) \in conf(s)\}$

- Beispiel:
  - Konfliktgraph für Schedule $s = r_1(x)\ r_2(x)\ w_1(x)\ r_3(x)\ w_3(x)\ w_2(y)\ c_3\ c_2\ w_1(y)\ c_1$:



Serialisierbarkeitssatz

- Die Zugehörigkeit eines Schedules s zur Klasse der konfliktserialisierbaren Schedules *CSR* kann durch die folgende Aussage bestimmt werden:

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

- Beweis wird mittels topologischem Sortieren geführt

- Korollar: Zugehörigkeit zu CSR kann in polynomieller Zeit geprüft werden

Beweis

Konfliktgraph ist zykelfrei ⟺ Schedule ist konfliktserialisierbar

- Konfliktgraph ist zykelfrei ⇐ Schedule ist konfliktserialisierbar
  - Leicht: Konfliktgraph hat Zykel ⟹ Schedule ist nicht konfliktserialisierbar
  - $t_1 \rightarrow t_2 \rightarrow \ \ldots \mathrm{t_n} \ \rightarrow t_1$
- Konfliktgraph ist zykelfrei ⇒ Schedule ist konfliktserialisierbar
  - Induktion über Anzahl der Transaktionen n
  - *n* = 1: Graph und Schedule haben nur eine Transaktion.
  - *n* = *n* + 1:
    - Graph ist zykelfrei
      ⇒ mindestens ein Knoten t_i ohne eingehende Kante
      ⇒ es gibt keine Aktion einer anderen Transaktion, die vor einer Aktion in t_i ausgeführt wird und mit dieser Aktion in Konflikt steht
    - Alle Aktionen aus t_i können an den Anfang bewegt werden (Reihenfolge innerhalb t_i bleibt erhalten).
    - Restgraph ist wieder azyklisch (Entfernung von Kanten aus einem azyklischen Graph kann ihn nicht zyklisch machen).
    - Restgraph hat *n*-1 Transaktionen

Einige Synchronisationsprobleme

- Welche Synchronisationsprobleme werden durch *CSR* verhindert?
  - Lost Update
    - $s = r_1(x)\ r_2(x)\ w_1(x)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - Dirty Read
    - $s = r_1(x)\ w_1(x)\ r_2(x)\ \boldsymbol{a_1}\ w_2(x)\ \boldsymbol{c_2} \in \boldsymbol{CSR}$
  - Phantom
    - $s = r_1(x)\ r_1(y)\ r_2(z)\ w_2(z)\ r_2(x)\ w_2(x)\ c_2\ r_1(z)\ c_1 \notin CSR$

Konfliktserialisierbarkeit und Fehlersicherheit

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

  - aus Sicht der Fehlersicherheit nicht akzeptabel, da t_2 Objekt x von t_1 liest und t_1 danach abbricht (Dirty Read)
  - Datenbank muss dafür sorgen, dass $w_1(x)$ nicht durchgeführt wird und t_2 nicht von t_1 liest
- **Fehlersicherheit** eines Schedules ist die Eigenschaft, dass dieser Schedule sich bei Abbruch einer oder mehreren Transaktionen genauso verhält wie ein ähnlicher Schedule, der ausschließlich die nicht abgebrochenen Transaktionen enthält. Fehlersicherheit ist eine „orthogonale“ Eigenschaft, die erst zusammen mit Konfliktserialisierbarkeit zu korrekter Synchronisation führt

„liest-von“-Notation

- Sei s ein Schedule, dann bezeichnet $p <_s q$ das Auftreten von Aktion p vor q
- Seien $t_i, t_j \in trans(s)$:

1) $\boldsymbol{t_i}$ **liest** $\boldsymbol{x}$ **von** $\boldsymbol{t_j}$ **in** $\boldsymbol{s}$, falls alle drei Bedingungen gelten:
   a) $w_j(x) <_s r_i(x)$
      - Eine Aktion $r_i(x)$ liest x von $w_j(x)$
   b) $a_j \not<_s r_i(x)$
      - t_j ist zum Zeitpunkt des Lesens nicht abgebrochen
   c) $w_j(x) <_s w_k(x) <_s r_i(x) \Rightarrow a_k <_s r_i(x)$
      - $w_j(x)$ ist der letzte echte Schreiber auf x vor $r_i(x)$
2) $\boldsymbol{t_i}$ **liest von** $\boldsymbol{t_j}$ **in** $\boldsymbol{s}$, falls t_i irgendein x von t_j in s liest

„liest-von“-Notation: Beispiel

- Sei s ein Schedule mit Transaktionen $t_1, t_2, t_3 \in trans(s)$:

$$s = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ r_3(x)\ w_2(z)\ a_2\ r_1(z)\ c_1\ c_3$$

- Dann gilt:
  - t_3 liest x von t_2 aber nicht von t_1
  - t_2 liest y von t_1
  - t_1 liest z nicht von t_2

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Eine Transaktion wird freigegeben (committed), wenn alle anderen Transaktionen von denen sie gelesen hat, auch freigegeben wurden.
- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit: Beispiel

Seien s_I, s_{II} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_I) \cup trans(s_{II})$:

$$s_I = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$$

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

- Dann gilt:
  - t_2 liest y von t_1 in s_I und $c_2 \in s_I$, aber $c_1 \not\prec_{s_I} c_2$. Hieraus folgt $s_I \notin RC$.
  - $s_{II} \in RC$, da die Commit-Operation von t_2 hinter der von t_1 steht.

- Weiteres Problem tritt auf in s_{II}, falls t_1 direkt nach $r_2(y)$ abbrechen würde
  - ➢ Dirty Read-Situation würde Abbruch von t_2 nach sich ziehen

Vermeidung kaskadierender Aborts

- Ein Schedule s **vermeidet kaskadierende Aborts**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest } x \text{ von } t_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

- Eine Transaktion darf nur Werte von bereits erfolgreich abgeschlossenen Transaktionen lesen.
- Die **Klasse aller Schedules, welche kaskadierende Aborts vermeiden**, bezeichnen wir mit ACA (avoids cascading aborts).

Vermeidung kaskadierender Aborts: Beispiel

Seien s_{II}, s_{III} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{II}) \cup trans(s_{III})$:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

- Dann gilt:
  - $s_{II} \notin ACA$
  - $s_{III} \in ACA$

- Weiteres Problem tritt auf in s_{III}, falls t_1 direkt nach $w_2(x)$ abstürzt
  - t_2 braucht nicht abgebrochen zu werden
  - Objekt x muss auf den richtigen Zustand zurückgesetzt werden

Striktheit

- Ein Schedule s heißt **strikt**, falls für alle Transaktionen $t_i, \in trans(s)$ und für alle Aktionen $p_i(x) \in op(t_i)$ gilt:

$$w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$$

- Kein Objekt wird gelesen oder überschrieben, bis die Transaktion, welche es zuletzt geschrieben hat, (erfolgreich oder erfolglos) beendet ist.
- Die **Klasse aller strikten Schedules** bezeichnen wir mit ST.

Striktheit: Beispiel

Seien s_{III}, s_{IV} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{III}) \cup trans(s_{IV})$:

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

$$s_{IV} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$$

- Dann gilt:
  - $s_{III} \notin ST$
  - $s_{IV} \in ST$

Korrektheit von Schedules

- Fehlersicherheit (ST, ACA, RC) und Konfliktserialisierbarkeit (CSR) sind „orthogonale“ Anforderungen an Schedules

- Ein Schedule s heißt **korrekt**, falls er sowohl konfliktserialisierbar als auch fehlersicher ist, d.h. falls er in der Klasse CSR und in einer der Klassen RC, ACA oder ST liegt.

Fehlersichere Schedules

- Es gilt der folgende Zusammenhang: $ST \subset ACA \subset RC$

CSR : Konfliktserialisierbarkeit
RC : Recoverable
ACA : avoids cascading aborts
ST : Strikt