Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Vorlesung

Datenbanksysteme I

Wintersemester 2014/2015

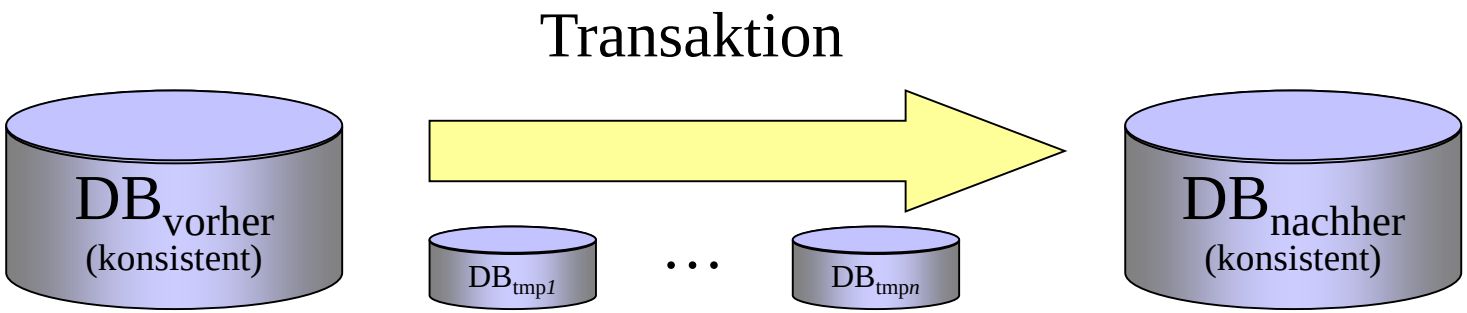
Kapitel 9: Transaktionen

Vorlesung: PD Dr. Arthur Zimek
Übungen: Sebastian Goebl, Jan Stutzki

http://www.dbs.ifi.lmu.de/Lehre/DBS

Transaktionskonzept

- Transaktion: Folge von Befehlen (*read*, *write*), die die DB von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt
- Transaktionen: Einheiten integritätserhaltender Zustandsänderungen einer Datenbank
- Hauptaufgaben der Transaktions-Verwaltung
  - Synchronisation (Koordination mehrerer Benutzerprozesse)
  - Recovery (Behebung von Fehlersituationen)



Transaktionskonzept

Beispiel Bankwesen:

Überweisung von Huber an Meier in Höhe von 200 €

- Mgl. Bearbeitungsplan:
  - (1) Erniedrige Stand von Huber um 200 €
  - (2) Erhöhe Stand von Meier um 200 €
- Möglicher Ablauf

Konto	Kunde	Stand
	Meier	1.000 €
	Huber	1.500 €

(1)

Konto	Kunde	Stand
	Meier	1.000 €
	Huber	1.300 €

(2)

System-absturz

Inkonsistenter DB-Zustand darf nicht entstehen bzw. darf nicht dauerhaft bestehen bleiben!

Eigenschaften von Transaktionen

- **ACID-Prinzip**
  - **Atomicity** (Atomarität)
    Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zum Tragen.
  - **Consistency** (Konsistenz, Integritätserhaltung)
    Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt.
  - **Isolation** (Isoliertheit, 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.
- Weitere Forderung: TA muss in endlicher Zeit bearbeitet werden können

Steuerung von Transaktionen

- **begin of transaction (BOT)**
  - markiert den Anfang einer Transaktion
  - Transaktionen werden implizit begonnen, es gibt kein `begin work` o.ä.
- **end of transaction (EOT)**
  - markiert das Ende einer Transaktion
  - alle Änderungen seit dem letzten BOT werden festgeschrieben
  - SQL: `commit oder commit work`
- **abort**
  - markiert den Abbruch einer Transaktion
  - die Datenbasis wird in den Zustand vor BOT zurückgeführt
  - SQL: `rollback oder rollback work`
- **Beispiel**

```
UPDATE Konto SET Stand = Stand-200 WHERE Kunde = ‘Huber’;
UPDATE Konto SET Stand = Stand+200 WHERE Kunde = ‘Meier’;
COMMIT;
```

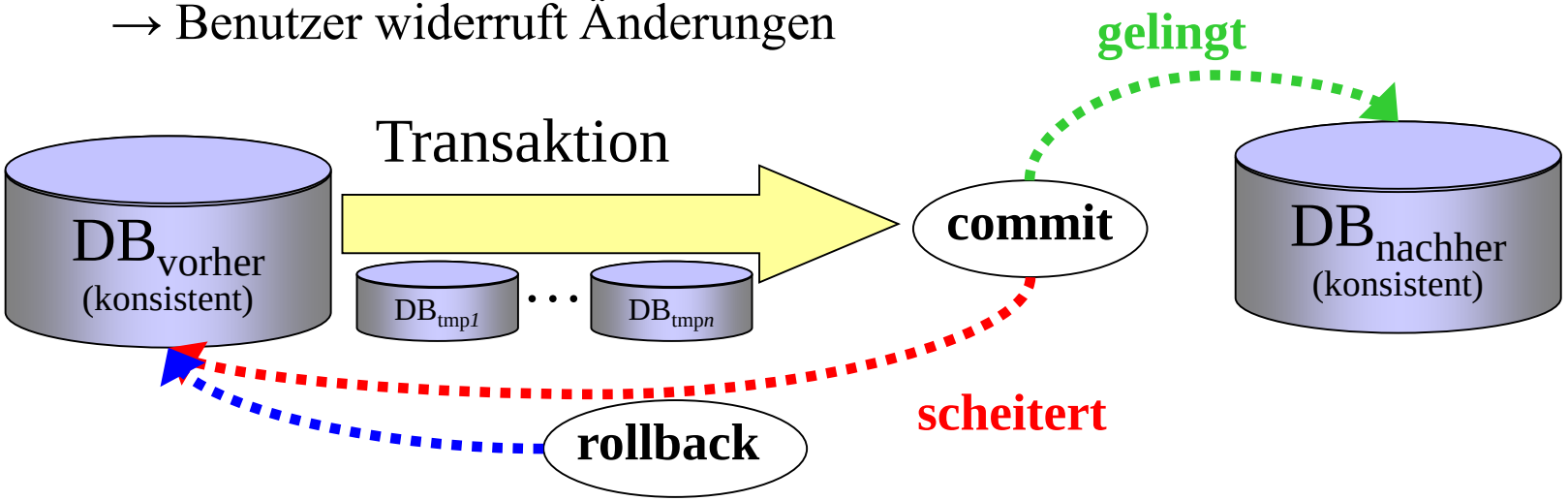
Steuerung von Transaktionen

Unterstützung langer Transaktionen durch

- **define savepoint**
  - markiert einen zusätzlichen Sicherungspunkt, auf den sich die noch aktive Transaktion zurücksetzen lässt
  - Änderungen dürfen noch nicht festgeschrieben werden, da die Transaktion noch scheitern bzw. zurückgesetzt werden kann
  - SQL: `savepoint <identifier>`
- **backup transaction**
  - setzt die Datenbasis auf einen definierten Sicherungspunkt zurück
  - SQL: `rollback to <identifier>`

Ende von Transaktionen

- COMMIT **gelingt**
  → der neue Zustand wird dauerhaft gespeichert.
- COMMIT **scheitert**
  → der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann z.B. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.
- ROLLBACK
  → Benutzer widerruft Änderungen



Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Klassifikation von Fehlern

- **Transaktionsfehler**
  Lokaler Fehler einer noch nicht festgeschriebenen Transaktion, z.B. durch
  - Fehler im Anwendungsprogramm
  - Expliziter Abbruch der Transaktion durch den Benutzer (`ROLLBACK`)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - Konflikte mit nebenläufigen Transaktionen (Deadlock)

Klassifikation von Fehlern

- **Systemfehler**
  Fehler mit Hauptspeicherverlust, d.h. permanente Speicher sind *nicht* betroffen, z.B. durch
  - Stromausfall
  - Ausfall der CPU
  - Absturz des Betriebssystems, …
- **Medienfehler**
  Fehler mit Hintergrundspeicherverlust, d.h. Verlust von permanenten Daten, z.B. durch
  - Plattencrash
  - Brand, Wasserschaden, …
  - Fehler in Systemprogrammen, die zu einem Datenverlust führen

Recovery-Techniken

- **Rücksetzen** (bei Transaktionsfehler)
  - *Lokales UNDO*: der ursprüngliche DB-Zustand wie zu BOT wird wiederhergestellt, d.h. Rücksetzen aller Aktionen, die diese Transaktion ausgeführt hat
  - Transaktionsfehler treten relativ häufig auf
    → Behebung innerhalb von Millisekunden notwendig

Recovery-Techniken

- **Warmstart** (bei Systemfehler)
  - *Globales UNDO*: Rücksetzen aller noch nicht abgeschlossenen Transaktionen, die bereits in die DB eingebracht wurden
  - *Globales REDO*: Nachführen aller bereits abgeschlossenen Transaktionen, die noch nicht in die DB eingebracht wurden
  - Dazu sind Zusatzinformationen aus einer Log-Datei notwendig, in der die laufenden Aktionen, Beginn und Ende von Transaktionen protokolliert werden
  - Systemfehler treten i.d.R. im Intervall von Tagen auf → Recoverydauer einige Minuten

Recovery-Techniken

- **Kaltstart** (bei Medienfehler)
  - Aufsetzen auf einem früheren, gesicherten DB-Zustand (Archivkopie)
  - *Globales REDO:* Nachführen aller Transaktionen, die nach dem Erzeugen der Sicherheitskopie abgeschlossenen wurden
  - Medienfehler treten eher selten auf (mehrere Jahre) → Recoverydauer einige Stunden / Tage
  - Wichtig: regelmäßige Sicherungskopien der DB notwendig

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Datenintegrität

- **Beispiel**

  Kunde(KName, KAdr, Konto)
  Auftrag(KName, Ware, Menge)
  Lieferant(LName, LAdr, Ware, Preis)

- **Mögliche Integritätsbedingungen**
  - Kein Kundenname darf mehrmals in der Relation „Kunde“ vorkommen.
  - Jeder Kundenname in „Auftrag“ muss auch in „Kunde“ vorkommen.
  - Kein Kontostand darf unter -100 € sinken.
  - Das Konto des Kunden Huber darf überhaupt nicht überzogen werden.
  - Es dürfen nur solche Waren bestellt werden, für die es mindestens einen Lieferanten gibt.
  - Der Brotpreis darf nicht erhöht werden.

Statische vs. dynamische Integrität

- **Statische Integritätsbedingungen**
  - Einschränkung der möglichen Datenbankzustände
  - z.B. „Kein Kontostand darf unter -100 € sinken.“
  - In SQL durch Constraint-Anweisungen implementiert (`UNIQUE, DEFAULT, CHECK, ….`)
  - Im Bsp.:

```
create table Kunde(
   kname varchar(40) primary key,
   kadr varchar(100),
   konto float check konto >= -100,
   );
```

Statische vs. dynamische Integrität

- **Dynamische Integritätsbedingungen**
  - Einschränkung der möglichen Zustandsübergänge
  - z.B. „Der Brotpreis darf nicht erhöht werden.“
  - In Oracle durch sog. Trigger implementiert
    - PL/SQL-Programm*, das einer Tabelle zugeordnet ist und durch ein best. Ereignis ausgelöst wird
    - Testet die mögliche Verletzung einer Integritätsbedingung und veranlasst daraufhin eine bestimmte Aktion
    - Mögliche Ereignisse: insert, update oder delete
    - **Befehls-Trigger** (statement-trigger): werden einmal pro auslösendem Befehl ausgeführt
    - **Datensatz-Trigger** (row-trigger): werden einmal pro geändertem / eingefügtem / gelöschtem Datensatz ausgeführt
    - Mögliche Zeitpunkte: vor (BEFORE) oder nach (AFTER) dem auslösenden Befehl oder alternativ dazu (INSTEAD OF)

*PL/SQL ist eine ORACLE-eigene prozedurale Erweiterung von SQL (Schleifen, Fallunterscheidung…)

Modellinhärente Integrität

Durch das Datenmodell vorgegebene Integritätsbedingungen

- **Typintegrität**
  Beschränkung der zulässigen Werte eines Attributs durch dessen Wertebereich
- **Schlüsselintegrität**
  DB darf keine zwei Tupel mit gleichem Primärschlüssel enthalten, z.B. „Kein Kundenname darf mehrmals in der Relation Kunde vorkommen.“.
- **Referentielle Integrität** (Fremdschlüsselintegrität)
  Wenn Relation R einen Schlüssel von Relation S enthält, dann muss für jedes Tupel in R auch ein entsprechendes Tupel in S vorkommen, z.B. „Jeder Kundenname in Auftrag muss auch in Kunde vorkommen.“.

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
2. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer
3. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

Synchronisation (Concurrency Control)

- **Serielle Ausführung** von Transaktionen ist unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist (niedriger Durchsatz, hohe Wartezeiten)
- **Mehrbenutzerbetrieb** führt i.A. zu einer besseren Auslastung des Systems (z.B. Wartezeiten bei I/O-Vorgängen können zur Bearbeitung anderer Transaktionen genutzt werden)
- **Aufgabe der Synchronisation**
  Gewährleistung des logischen Einbenutzerbetriebs, d.h. innerhalb einer TA ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen

Anomalien bei unkontrolliertem Mehrbenutzerbetrieb

- Verloren gegangene Änderungen (*Lost Updates*)
- Zugriff auf „schmutzige“ Daten (*Dirty Read / Dirty Write*)
- Nicht-reproduzierbares Lesen (*Non-Repeatable Read*)
- Phantomproblem
- **Beispiel:** Flugdatenbank

Passagiere	FlugNr	Name	Platz	Gepäck
	LH745	Müller	3A	8
	LH745	Meier	6D	12
	LH745	Huber	5C	14
	BA932	Schmidt	9F	9
	BA932	Huber	5C	14

Lost Updates

- Änderungen einer Transaktion können durch Änderungen anderer Transaktionen überschrieben werden und dadurch verloren gehen
- Bsp.: Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus
  - T1: `UPDATE Passagiere SET Gepäck = Gepäck+3 WHERE FlugNr = LH745 AND Name = „Meier“;`
  - T2: `UPDATE Passagiere SET Gepäck = Gepäck+5 WHERE FlugNr = LH745 AND Name = „Meier“;`
- Mgl. Ablauf:

T1	T2
`read(Passagiere.Gepäck, x1);`	`read(Passagiere.Gepäck, x2);`
	`x2 := x2 + 5;`
	`write(Passagiere.Gepäck, x2);`
`x1 := x1+3;`	
`write(Passagiere.Gepäck, x1);`	

- In der DB ist nur die Änderung von T1 wirksam, die Änderung von T2 ist verloren gegangen → Verstoß gegen *Durability*

Dirty Read / Dirty Write

- Zugriff auf „schmutzige“ Daten, d.h. auf Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden
- Bsp.:
  - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
  - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen
- Mgl. Ablauf:

T1	T2

`UPDATE Passagiere`
`SET Gepäck = Gepäck+3;`
 	
`ROLLBACK;`
	
	`UPDATE Passagiere`
	`SET Gepäck = Gepäck+5;`
	`COMMIT;`

- Durch den Abbruch von T1 werden die geänderten Werte ungültig. T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*)
- Verstoß gegen ACID: Dieser Ablauf verursacht einen inkonsistenten DB-Zustand (*Consistency*) bzw. T2 muss zurückgesetzt werden (*Durability*).

Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Bsp.:
  - T1 liest das Gepäckgewicht der Passagiere auf Flug BA932 zweimal
  - T2 bucht den Platz 3F auf dem Flug BA932 für Passagier Meier mit 5kg Gepäck
- Mgl. Ablauf

T1	T2

`SELECT Gepäck FROM Passagiere`
`WHERE FlugNr = „BA932“;`
 	
`SELECT Gepäck FROM Passagiere`
`WHERE FlugNr = „BA932“;`
	`INSERT INTO Passagiere`
	`VALUES (BA932, Meier, 3F, 5);`
	`COMMIT;`

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl die T1 den DB-Zustand nicht geändert hat → Verstoß gegen *Isolation*

Phantomproblem

- Ausprägung des nicht-reproduzierbaren Lesens, bei der Aggregationsfunktionen beteiligt sind
- Bsp.:
  - T1 druckt die Passagierliste sowie die Anzahl der Passagiere für den Flug LH745
  - T2 bucht den Platz 7D auf dem Flug LH745 für Phantomas
- Mgl. Ablauf

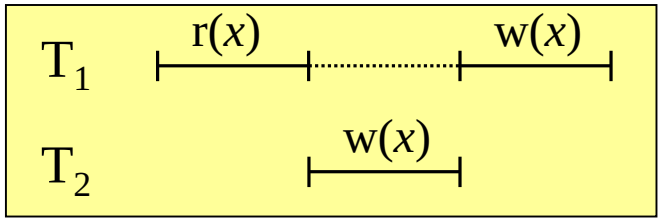
T1	T2

`SELECT * FROM Passagiere`
`WHERE FlugNr = „LH745“;`
 	
`SELECT COUNT(*) FROM Passagiere`
`WHERE FlugNr = „ LH745“;`
	`INSERT INTO Passagiere`
	`VALUES (LH745, Phantomas, 7D, 2);`
	`COMMIT;`

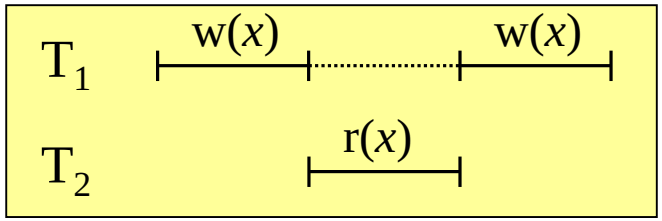
- Für Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere berücksichtigt ist

Muster der Anomalien

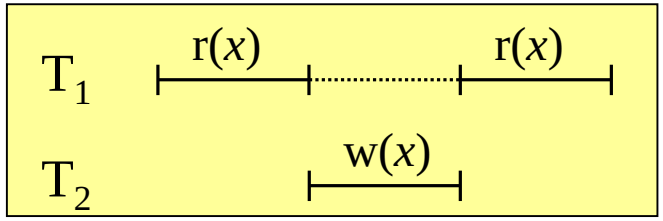
- Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



- Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$



- Non-repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$



Serialisierbarkeit von Transaktionen

- Die nebenläufige Bearbeitung von Transaktionen geschieht für den Benutzer transparent, d.h. als ob die Transaktionen (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden
- **Begriffe**
  - 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.
  - Ein Schedule S von $\{T_1, \ldots, T_n\}$ ist *serialisierbar*, wenn er dieselbe Wirkung hat wie ein beliebiger serieller Schedule von $\{T_1, \ldots, T_n\}$.

Nur serialisierbare Schedules dürfen zugelassen werden!

Schedule

Gegeben seien folgende Transaktionen:

- $T_1 = (r_1(x), w_1(x), r_1(y), w_1(y))$
- $T_2 = (r_2(y), w_2(y), w_2(y))$
- $T_3 = (r_3(z), w_3(z), r_3(x))$

Beispiele für einen allgemeinen und einen seriellen Schedule

- $S_{allgem} = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$
- $S_{seriell} = (r_2(y), w_2(y), w_2(y), r_3(z), w_3(z), r_3(x), r_1(x), w_1(x), r_1(y), w_1(y))$

Frage: Ist S_{allgem} auch serialisierbar?

Wie viele Schedules existieren bei n Transaktionen mit jeweils m_i Aktionen?

- Serielle Schedules: $n!$ $(=1 \cdot 2 \cdot 3 \cdot \ldots \cdot n)$
- Allgemeine Schedules: $\frac{(m_1 + m_2 + \ldots + m_n)!}{m_1! \cdot m_2! \cdot \ldots \cdot m_n!}$

Serialisierbarkeit von Transaktionen

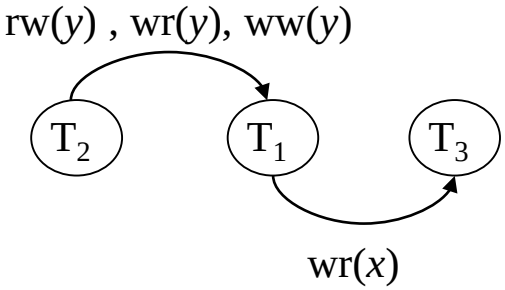
Mit Hilfe von Serialisierungsgraphen kann man prüfen, ob ein Schedule $\{T_1, \ldots, T_n\}$ serialisierbar ist

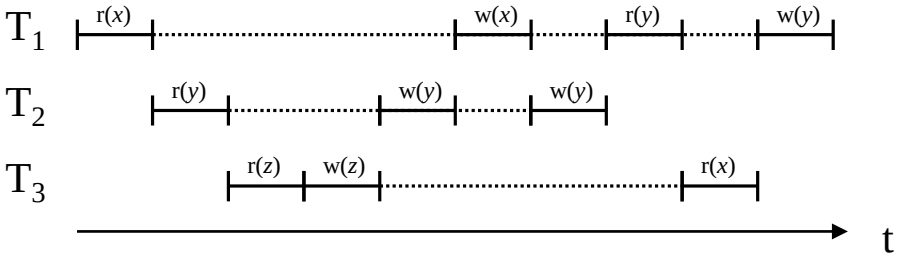
- Die beteiligten Transaktionen $\{T_1, \ldots, T_n\}$ sind die *Knoten* des Graphen
- Die *Kanten* beschreiben die Abhängigkeiten der Transaktionen:
  Eine Kante $T_i \rightarrow T_j$ wird eingetragen, falls im Schedule
  - $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr(x)$
  - $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw(x)$
  - $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww(x)$
- Ein Schedule ist serialisierbar, falls der Serialisierungs-graph *zyklenfrei* ist
- Einen zugehörigen seriellen Schedule erhält man durch topologisches Sortieren des Graphen

Kriterium für Serialisierbarkeit

Beispiele

- $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$

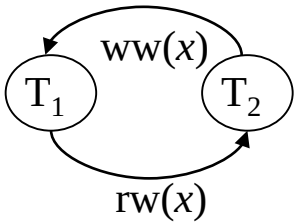




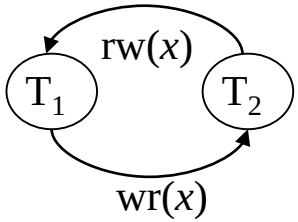
Serialisierungsreihenfolge: (T_2, T_1, T_3)

- Nicht-serialisierbare Schedules

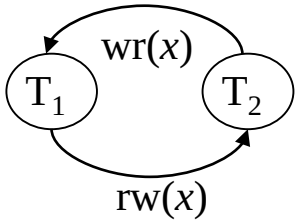
$S=(r_1(x), w_2(x), w_1(x))$
Lost Update



$S=(w_1(x), r_2(x), w_1(x))$
Dirty Read



$S=(r_1(x), w_2(x), r_1(x))$
Non-Repeatable Read



Techniken zur Synchronisation

- Die nebenläufige Bearbeitung von Transaktionen geschieht für den Benutzer transparent, d.h. als ob die Transaktionen (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden
- **Pessimistische Ablaufsteuerung** (Locking)
  - Konflikte werden vermieden, indem Transaktionen durch Sperren blockiert werden
  - Nachteil: ggf. lange Wartezeiten
  - Vorteil: I.d.R. nur wenig Rücksetzungen aufgrund von Synchronisationsproblemen nötig
  - Standardverfahren
- **Optimistische Ablaufsteuerung** (Zeitstempelverfahren)
  - Transaktionen werden im Konfliktfall zurückgesetzt
  - Transaktionen arbeiten bis zum COMMIT ungehindert. Anschließend erfolgt Prüfung anhand von Zeitstempeln, ob ein Konflikt aufgetreten ist
  - Nur geeignet, falls Konflikte zwischen Schreibern eher selten auftreten

Techniken zur Synchronisation

- Nur-lesende Transaktionen können sich gegenseitig nicht beeinflussen, da Synchronisationsprobleme nur im Zusammenhang mit Schreiboperationen auftreten.
- Es gibt deshalb die Möglichkeit, Transaktionen als nur-lesend zu markieren, wodurch die Synchronisation vereinfacht und ein höherer Parallelitätsgrad ermöglicht wird:
  - `SET TRANSACTION READ-ONLY`:
    kein INSERT, UPDATE, DELETE
  - `SET TRANSACTION READ-WRITE`:
    alle Zugriffe möglich