C | A | U
Christian-Albrechts-Universität zu Kiel

Christian Albrechts Universität zu Kiel
Institut für Informatik
AG Archäoinformatik (Data Science)

Skript zur Vorlesung

Informationssysteme

Wintersemester 2019/20

Kapitel 9: Transaktionssysteme
Einführung

Skript © 2019 Matthias Renz, CAU Kiel
(basiert auf dem Skript der LMU München)

Transaktionssystem

- Begriffserklärung und Beispiele

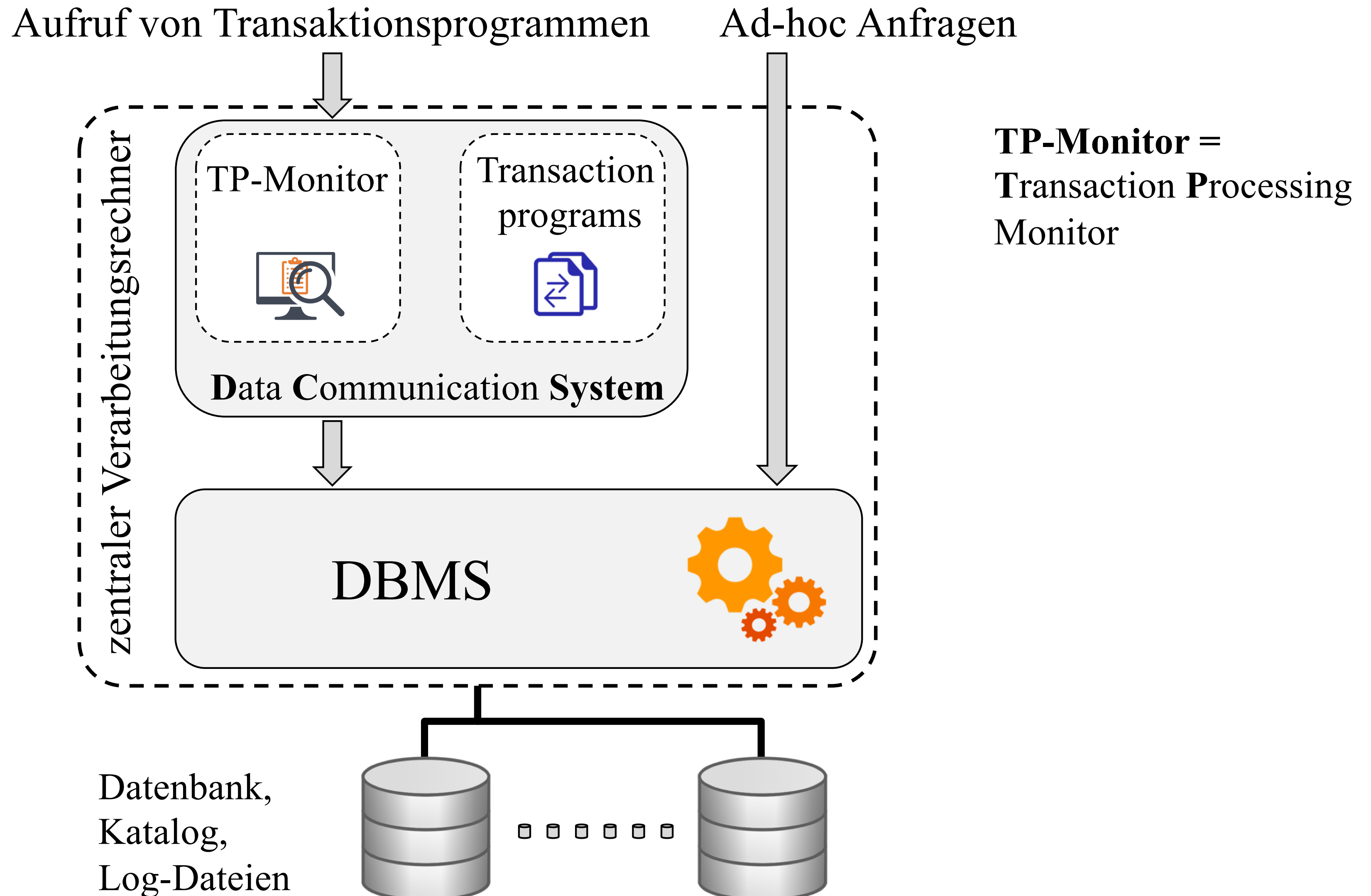
Allgemein:
*Ein **Transaktionssystem** ist ein System mit großen Datenbanken und Hunderten von Benutzern, die gleichzeitig Datenbanktransaktionen durchführen.*
[Elmasri, Navathe: Grundlagen von Datenbanksystemen, Pearson Studium]

- Typische Beispiele: Systeme für ..
  - Reservierungen,
  - Banktransaktionen,

In der Wirtschaftsinformatik:
*Ein **Transaktionssystem** ist ein Softwaresystem, mit dem betriebliche Transaktionen, vor allem in operativen Anwendungsbereichen, durchgeführt werden.*
[Roland Gabriel: Enzyklopädie der Wirtschaftsinformatik]

Transaktionssystem

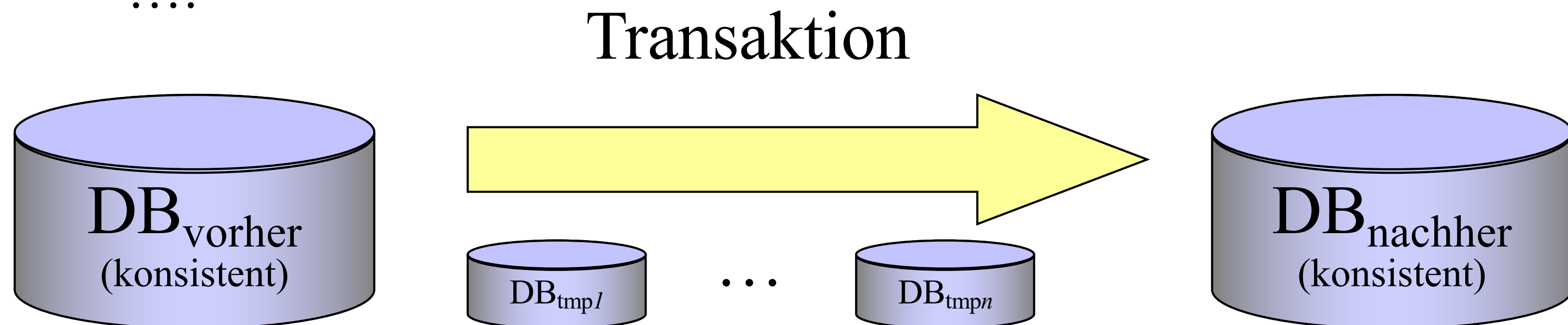
- (Grob-) Architektur eines zentralen Transaktionssystems



TP-Monitor =
Transaction **P**rocessing
Monitor

Transaktionskonzept

- Transaktion: Folge von Befehlen (*read, write*), die die DB von einen 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)
  - Sicherstellung der Korrektheit/Konsistenz der Daten/Datenbank
  - ….



CAU
Christian-Albrechts-Universität zu Kiel

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 um200 €
- 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;
```

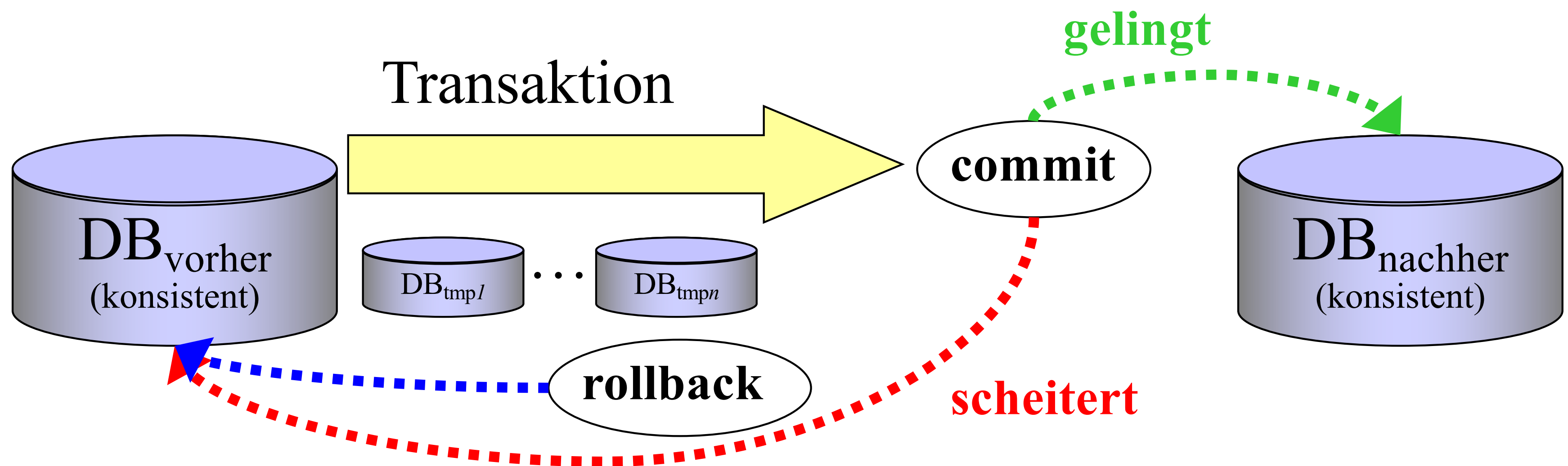
Steuerung von Transaktionen

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

Christian-Albrechts-Universität zu Kiel

Aufgaben eines DBMS

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

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

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

`ROLLBACK;`	

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

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

Phantomproblem

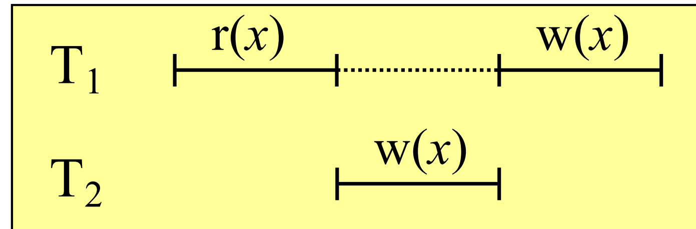
- Ausprägung des nicht-reproduzierbaren Lesen, bei der Aggregationsfunktionen beteiligt sind
- Bsp.:
  - T1 druckt die Passagierliste sowie die Anzahl der Passagiere für den Flug LH745
  - T2 bucht den Platz 7D auf dem Flug LH745 für Phantomas
- Mgl. Ablauf

T1	T2
`SELECT * FROM Passagiere WHERE FlugNr = „LH745";`	
	`INSERT INTO Passagiere VALUES (LH745, Phantomas, 7D, 2); COMMIT;`
`SELECT COUNT(*) FROM Passagiere WHERE FlugNr = „ LH745";`	

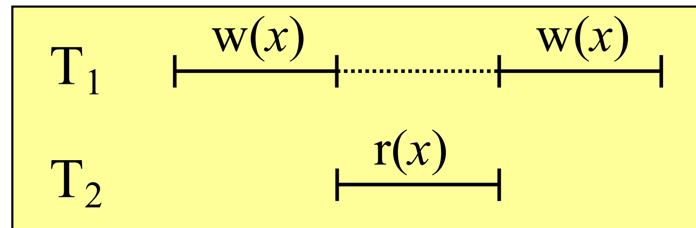
- Für Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere berücksichtigt ist

Muster der Anomalien

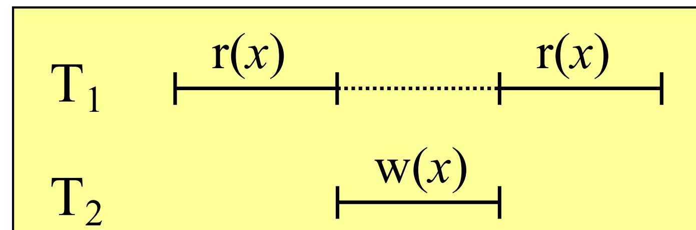
- Lost Update: S=($r_1(x)$, $w_2(x)$, $w_1(x)$)



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



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



Aufgaben eines DBMS

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

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

C|A|U
Christian-Albrechts-Universität zu Kiel

Christian Albrechts Universität zu Kiel
Institut für Informatik
AG Archäoinformatik (Data Science)

Skript zur Vorlesung

Informationssysteme

Wintersemester 2019/20

Kapitel 9.1: Transaktionssysteme Synchronisation

Skript © 2019 Matthias Renz, CAU Kiel
(basiert auf dem Skript der LMU München)

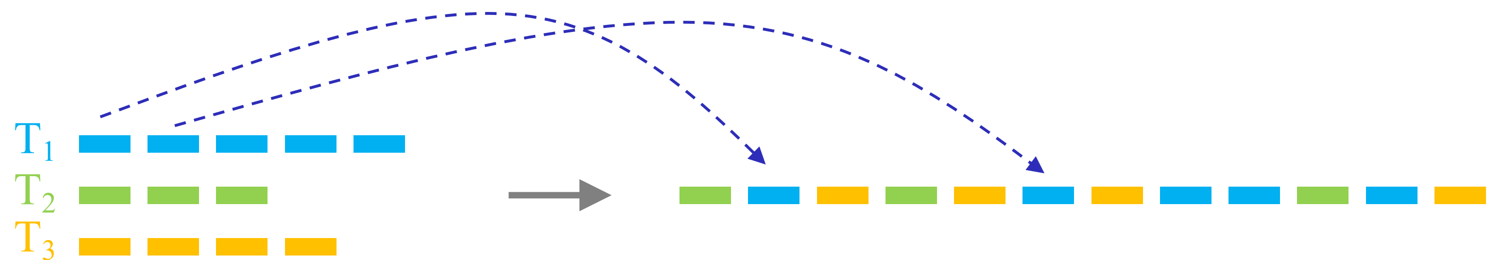
Synchronisation

- Motivation:
  - Verzahnung von Transaktionen im Mehrbenutzerbetrieb zur besseren Auslastung des Systems → Effizienzsteigerung.
  - Unkontrollierte Verzahnung von Transaktionen kann allg. zu Fehlern/Anomalien (Lost-Update, Dirty Read, …) führen → Verletzung des ACID Prinzips.
- Ziel:
  - Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer.
  - Einhaltung des ACID-Prinzips
- Aufgabe:
  - Kontrollierte Abarbeitung von Transaktionen (TAs), sodass sich die TAs nicht gegenseitig stören können.
  - Wirkung als ob die TAs (in einer beliebigen Reihenfolge) hintereinander ausgeführt wurden.

→logische *Serialisierbarkeit* mehrerer TAs

Schedules

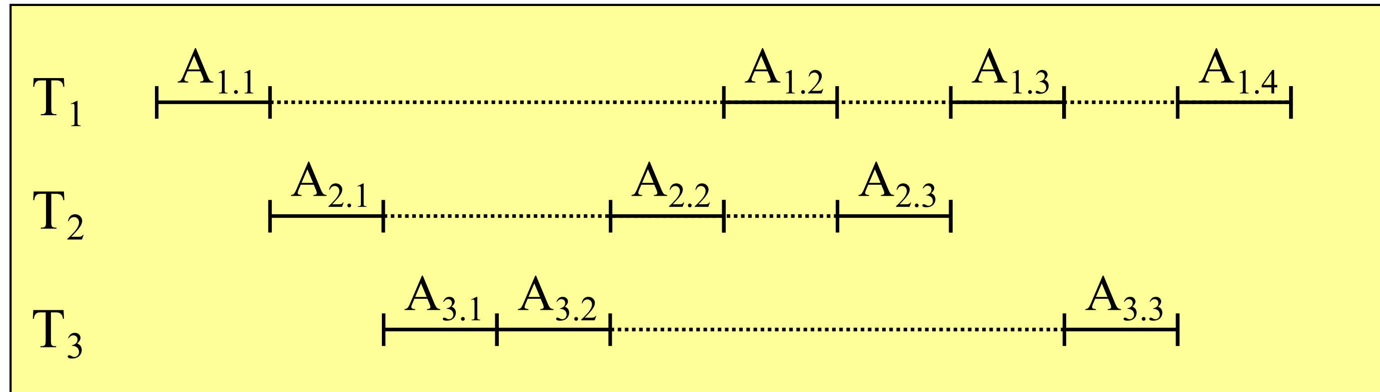
- **Begriffe**
  - Ein *Schedule* für eine Menge $\mathcal{T}=\{T_1, \ldots, T_n\}$ von Transaktionen ist eine beliebig gemischte Folge von Aktionen aus $\mathcal{T}$, wobei die Reihenfolge der Aktionen innerhalb einer Transaktion beibehalten wird.



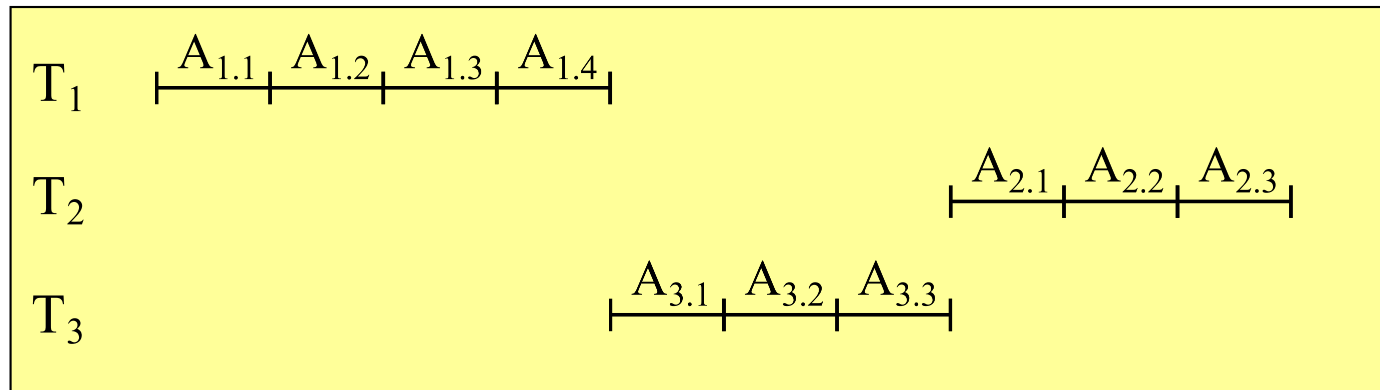
  - Ein *serieller Schedule* ist ein Schedule S von $\{T_1, \ldots, T_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken <u>hintereinander</u> ausgeführt werden.

Schedules

- **Beispiel:** $T_1 = < A_{1.1}, A_{1.2}, A_{1.3}, A_{1.4} >$, $T_2 = < A_{2.1}, A_{2.2}, A_{2.3} >$, $T_3 = < A_{3.1}, A_{3.2}, A_{3.3} >$
- (Beliebiger) Schedule:



- Serieller Schedule:



Schedules

- **Eigenschaften:**
  - **Allgemeine Schedules** bieten offenbar eine beliebige Verzahnung und sind daher **aus Performanz-Gründen erwünscht**.
  - Aus Sicht des **ACID**-Prinzips (insb. Isolation) sind **serielle Schedules erwünscht**.

- Wünschenswert:
  Kompromiss zwischen Performanz (allg. Schedules) und Isolation (seriellen Schedules), d.h. Schedules mit verzahnten Transaktionen aber gleicher Wirkung

  → **Serialisierbarer Schedule**

Serialisierbarkeit von Transaktionen

- **Begriffe (cont.)**
  - Ein Schedule S von $\{T_1, \ldots, T_n\}$ ist *serialisierbar*, wenn er **dieselbe Wirkung** auf den resultierenden Datenbankinhalt hat wie ein beliebiger serieller Schedule von $\{T_1, \ldots, T_n\}$.

Nur serialisierbare Schedules dürfen zugelassen werden!

- Frage: Wann haben zwei Schedules S1 und S2 die gleiche Wirkung auf den resultierenden Datenbankinhalt?
- Achtung: Gleiches Ergebnis …
  - kann auch Ergebnis eines Zufalls sein.
  - kann nur nachträglich (nach der Ausführung von S1 und S2) festgestellt werden.

Serialisierbarkeit von Transaktionen

Beispiel:

- Im Folgenden verwendete Notation:
  $r_1(x)$:= Transaktion T_1 liest aus Datenblock x (liest Objekt x).
  $w_1(x)$:= Transaktion T_1 schreibt auf Datenblock x (schreibt Objekt x).

- 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

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

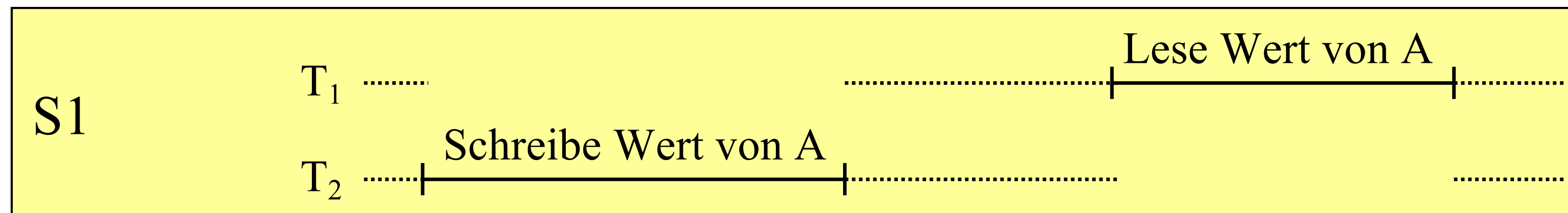
Frage: Ist S_{allgem} auch serialisierbar?

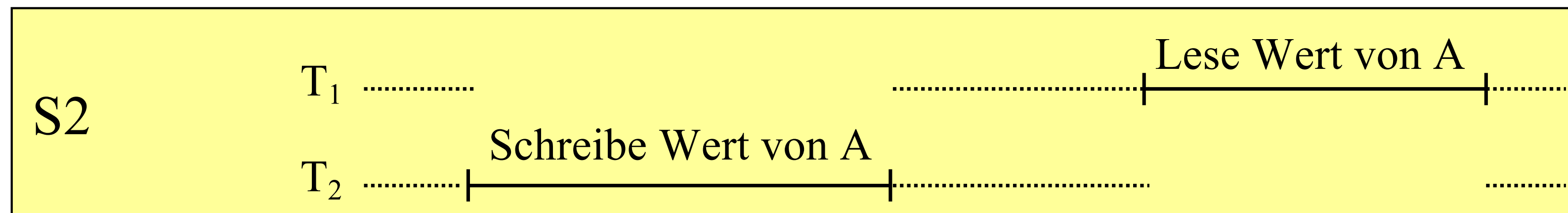
Abhängigkeiten und Konflikte

Wir benötigen ein objektivierbares Kriterium:

Konflikt Äquivalenz

- Idee:
  Wenn in Schedule S1 eine Transaktion T_1 z.B. einen Wert liest, den T_2 geschrieben hat, dann muss das auch in S2 so sein.





- Wir sprechen hier von einer Schreib-Lese-Abhängigkeit (bzw. Konflikt) zwischen T2 und T1 (in Schedule S1 und S2)

Abhängigkeiten und Konflikte

Abhängigkeiten (im Konflikt liegende Aktionen)

- Sei S ein Schedule mit (mind.) zwei Transaktionen T_1 und T_2.
- Wir sprechen von einer
  - Schreib-Lese-Abhängigkeit von $T_1 \rightarrow T_2$
    - Es existiert Objekt x, so dass in S **$w_1(x)$ vor $r_2(x)$** kommt
    - Abkürzung: $wr_{1,2}(x)$
  - Lese-Schreib-Abhängigkeit von $T_1 \rightarrow T_2$
    - Es existiert Objekt x, so dass in S **$r_1(x)$ vor $w_2(x)$** kommt
    - Abkürzung: $rw_{1,2}(x)$
  - Schreib-Schreib-Abhängigkeit von $T_1 \rightarrow T_2$
    - Es existiert Objekt x, so dass in S **$w_1(x)$ vor $w_2(x)$** kommt
    - Abkürzung: $ww_{1,2}(x)$

Warum keine Lese-Lese-Abhängigkeiten?

Christian-Albrechts-Universität zu Kiel

Abhängigkeiten und Konflikte

Konfliktäquivalenz von Schedules

- Zwei Schedules S1 und S2 heißen konfliktäquivalent, wenn
    1. S1 und S2 die **gleichen Transaktionen** (und Aktionsmengen) besitzen, d.h. wenn beide Schedules dieselben Operationen ausführen (innerhalb einer TA auch gleiche Reihenfolge der Aktionen (Operationen)!!!).
    2. S1 und S2 die **gleichen Abhängigkeitsmengen** besitzen, d.h. wenn in der Abhängigkeitsmenge von S1 z.B. die Schreib-Lese-Abhängigkeit ‘‘$w_1(x)$ vor $r_2(x)$" vorkommt (für ein Objekt x), dann muss diese auch in der Abhängigkeitsmenge von S2 vorkommen.

- Zwei konflikt-äquivalente Schedules haben die gleiche Wirkung auf den Datenbank-Inhalt.

Gilt die Umkehrung?

Abhängigkeiten und Konflikte

Beispiel:

$S_1 = (r_1(x), r_1(y), r_2(x), w_2(x), w_1(x), w_1(y))$

$S_2 = (r_2(x), r_1(x), r_1(y), w_2(x), w_1(x), w_1(y))$

$S_3 = (r_1(x), r_1(y), r_2(x), w_1(x), w_2(x), w_1(y))$

$S_4 = (r_2(x), r_1(y), r_1(x), w_2(x), w_1(y), w_1(x))$

- Transaktionen von S1, S2 und S3 sind identisch, aber Aktionen in T1 in S4 sind vertauscht im Vergleich zu T1 in S1/S2/S3.
- Abhängigkeitsmengen:

$A_{S1} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{2,1}(x)\}$
$A_{S2} = \{rw_{2,1}(x), rw_{1,2}(x), ww_{2,1}(x)\}$
$A_{S3} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{1,2}(x)\}$

Schedule S1 und S2 sind konfliktäquivalent

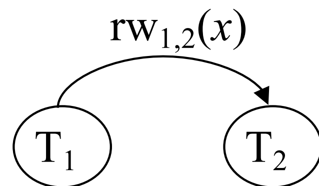
Schedule S1 und S3, bzw. S2 und S3 sind nicht konfliktäquivalent

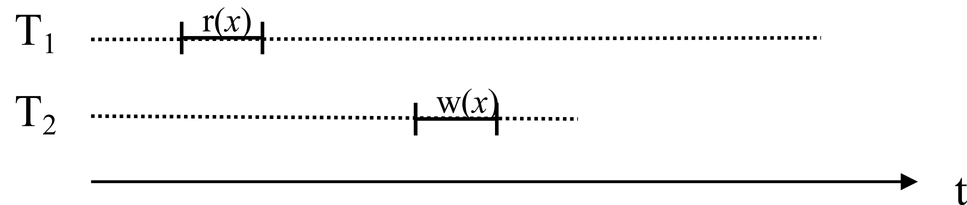
Serialisieruns-Graph

Um festzustellen ob ein Schedule $\{T_1, \ldots, T_n\}$ serialisierbar ist konstruiert man einen

Serialisierungs-Graphen

- Die beteiligten Transaktionen $\{T_1, \ldots, T_n\}$ sind die *Knoten* des Graphen
- Die Abhängigkeiten zwischen den Transaktionen (d.h. im Konflikt liegende Aktionen) beschreiben die *Kanten* des Graphen:
  Eine Kante $T_i \rightarrow T_j$ wird eingetragen, falls im Schedule
  - $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr_{i,j}(x)$
  - $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw_{i,j}(x)$
  - $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww_{i,j}(x)$
- Beispiel:

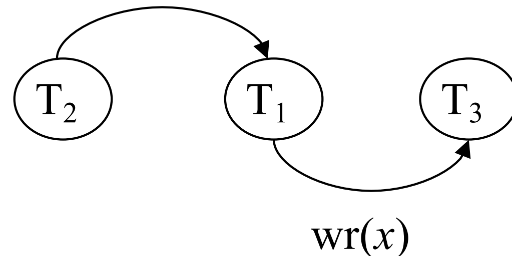


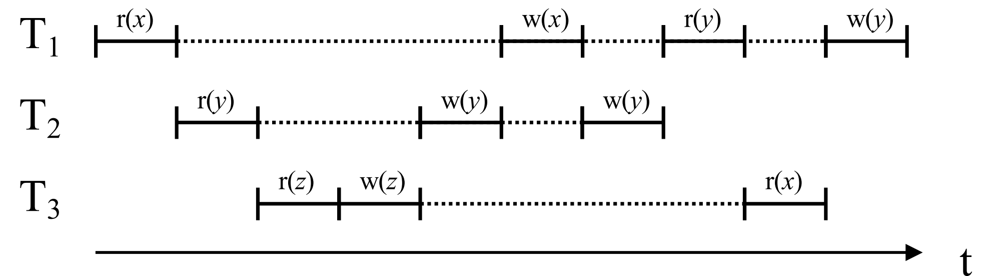


Serialisierbare Schedules

- Ein Schedule ist **serialisierbar**, falls der Serialisierungs-graph *zyklenfrei* ist.

- Den zu einem Schedule S zugehörigen konfliktäquivalenten seriellen Schedule S‘ erhält man durch topologisches Sortieren des Graphen.
- Die topologische Reihenfolge der Knoten des Graphen bestimmt die Reihenfolge der Transaktionen in dem seriellen Schedule S‘. (Serialisierungsreihenfolge)
- Es kann i.A. mehrere serielle Schedules geben.
- Beispiel
  - $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$



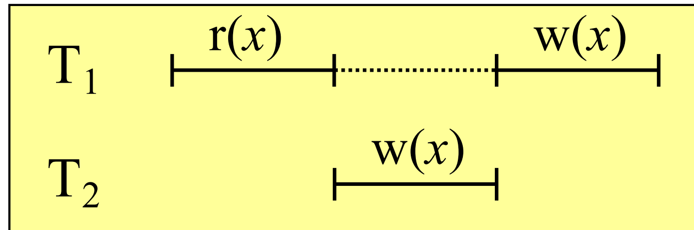


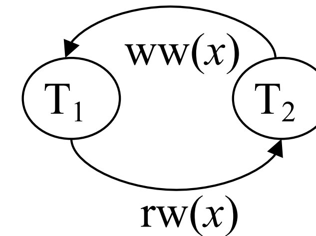
  - Serialisierungsreihenfolge: (T_2, T_1, T_3)

Nicht-serialisierbare Schedules

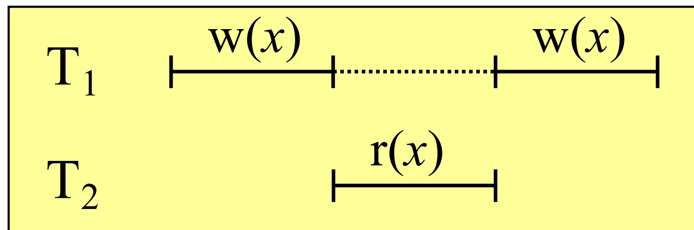
Beispiele für nicht-serialisierbare Schedules

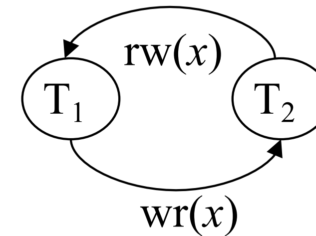
Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



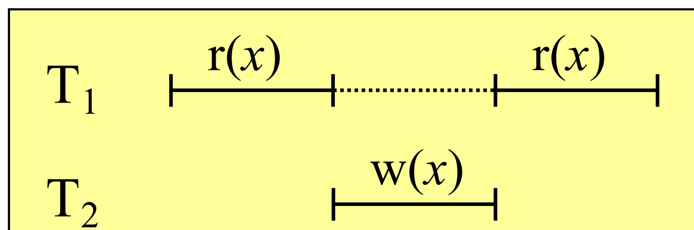


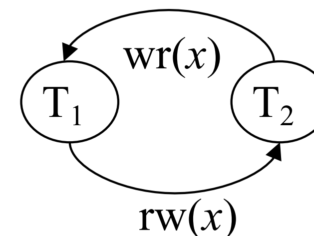
Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$





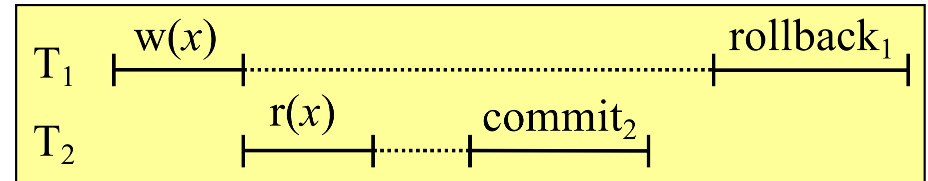
Non-Repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$



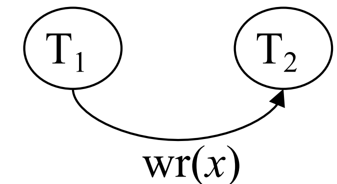


Rücksetzbare Schedules

- Bisher: Betrachtung der Serialisierbarkeit unter Beachtung von Schreib-Lese-Aktionen
- Frage: Was passiert, wenn eine Transaktion (z.B. auf eigenen Wunsch) zurückgesetzt wird (rollback)?
- Beispiel:
  - T_1 schreibt Datensatz (Objekt) x
  - T_2 liest Datensatz x
  - T_2 führt COMMIT aus
    Schedule ist serialisierbar,



Der Serialisierungs-Graph ist zyklenfrei



ABER

- T_1 wird zurückgesetzt (d.h. Datensatz x wird wieder auf den Ursprungswert zurückgesetzt)
- T_2 müsste eigentlich auch zurückgesetzt werden, hat aber schon COMMIT ausgeführt

Rücksetzbare Schedules

- Also: Serialisierbarkeit alleine reicht leider nicht aus, wenn TAs zurückgesetzt werden können

- Rücksetzbarer Schedule:
  Eine Transaktion T_i darf erst dann ihr COMMIT durchführen, wenn alle Transaktionen T_j, von denen sie Daten gelesen hat, beendet sind.

- Andernfalls entsteht das Problem:
  Falls ein T_j noch zurückgesetzt wird, müsste auch T_i zurückgesetzt werden, was nach COMMIT (T_i) nicht mehr möglich wäre.

Rücksetzbare Schedules

Noch schlimmer:

- Rücksetzbare Schedules können eine Lawine weiterer Rollbacks in Gang setzen

Schritt	T_1	T_2	T_3	T_4	T_5
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_5(D)$	
8.					$r_5(D)$
9.	$abort_1$				

- Schedule ohne kaskadierendes Rücksetzen:
  Änderungen werden erst nach dem COMMIT für andere Transaktionen zum Lesen freigegeben

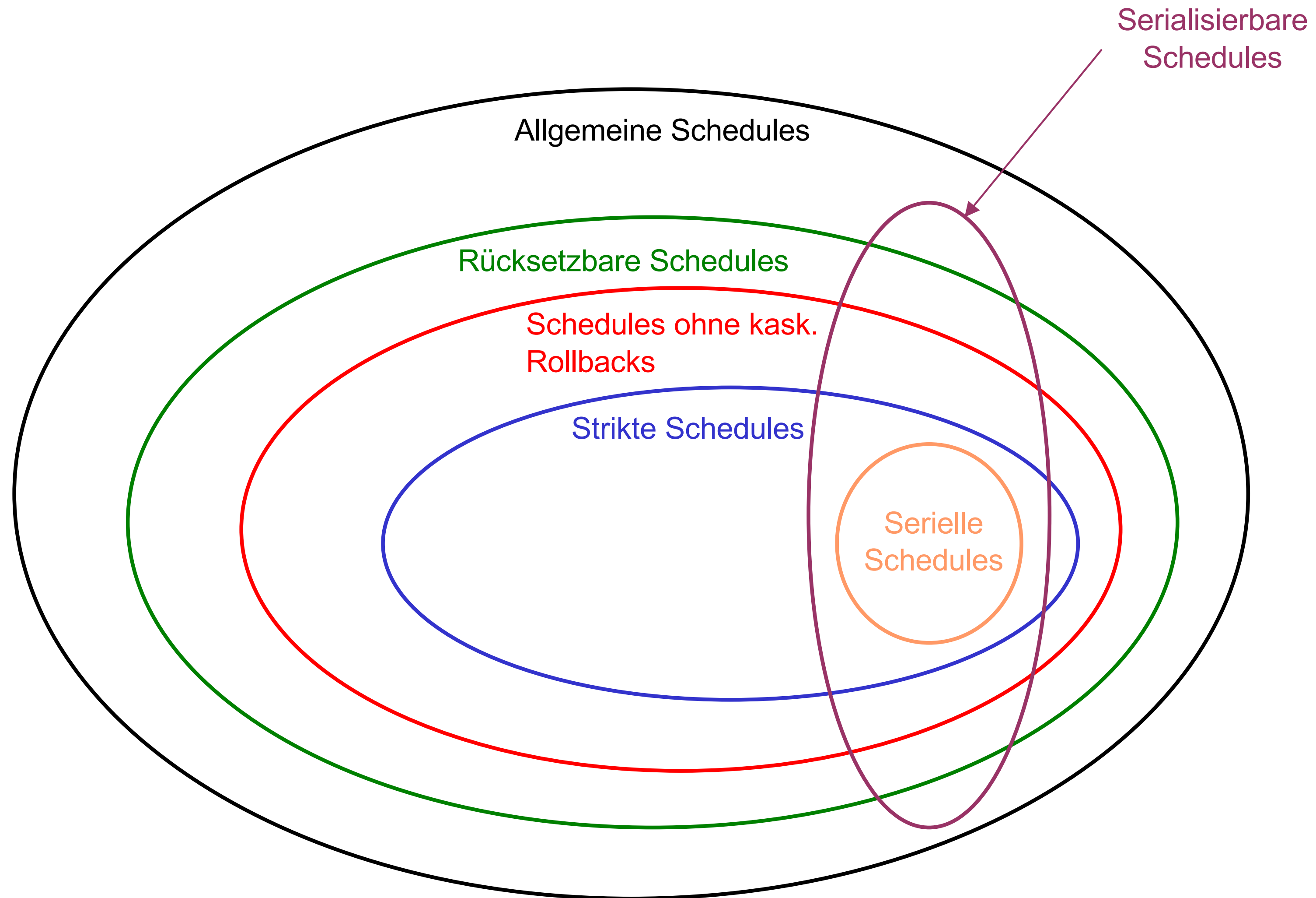
Scheduleklassen

Überblick von Scheduleklassen:

- Serieller Schedule
  - TAs in einzelnen Blöcken, phys. Einbenutzerbetrieb.
- Serialisierbarer Schedule
  - Konfliktäquivalent zu einem seriellen Schedule.
- Rücksetzbarer Schedule
  - TA darf erst committen, wenn alle TAs von denen sie Daten gelesen hat committed haben.
- Schedule ohne kaskadierendes Rollback
  - Veränderte Daten einer noch laufenden TA dürfen nicht gelesen werden.
- Strikter Schedule
  - Zusätzlich dürfen veränderte Daten einer noch laufenden TA nicht überschrieben werden.

Scheduleklassen

Beziehungen zwischen Scheduleklassen:



CAU
Christian-Albrechts-Universität zu Kiel

Ablaufsteuerungen

- Verwaltungsaufwand für Serialisierungsgraphen ist in der Praxis oft zu hoch → Sperr-Verfahren (Locking)

Klassen von Ablaufsteuerungen:

- **Pessimistische Ablaufsteuerung** (Locking, Sperren)
  - 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**
  - 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

Ablaufsteuerungen

- 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

Sperrverfahren (Locking)

Allgemeines:

- Sperrverfahren entsprechen der pessimistischen Ablaufsteuerung
- Sperrverfahren ist **das** Standardverfahren zur Synchronisation in relationalen DBMS
- Sperre (Lock)
  - Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
  - Anforderung einer Sperre durch LOCK, z.B. L(x) für LOCK auf Objekt x
  - Freigabe durch UNLOCK, z.B. U(x) für UNLOCK von Objekt x
  - LOCK / UNLOCK erfolgt atomar (also nicht unterbrechbar!)
  - Sperrgranularität (Objekte, auf denen Sperren gesetzt werden):
    - Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
  - Sperrenverwalter führt Tabelle für aktuell gewährte Sperren

Sperrverfahren (Locking)

Legale Schedules

- Vor jedem Zugriff auf ein Objekt wird eine geeignete Sperre gesetzt.
- Keine Transaktion fordert eine Sperre an, die sie schon besitzt.
- Spätestens bei Transaktionsende werden alle Sperren zurückgegeben.
- Sperren werden respektiert, d.h. eine mit gesetzten Sperren unverträgliche Sperranforderung (z.B. exklusiver Zugriff auf Objekt x) muss warten.

Bemerkungen

- Anfordern und Freigeben von Sperren sollte das DBMS implizit selbst vornehmen.
- Die Verwendung legaler Schedules garantiert noch nicht die Serialisierbarkeit (siehe folgendes Beispiel).

Sperrverfahren (Locking)

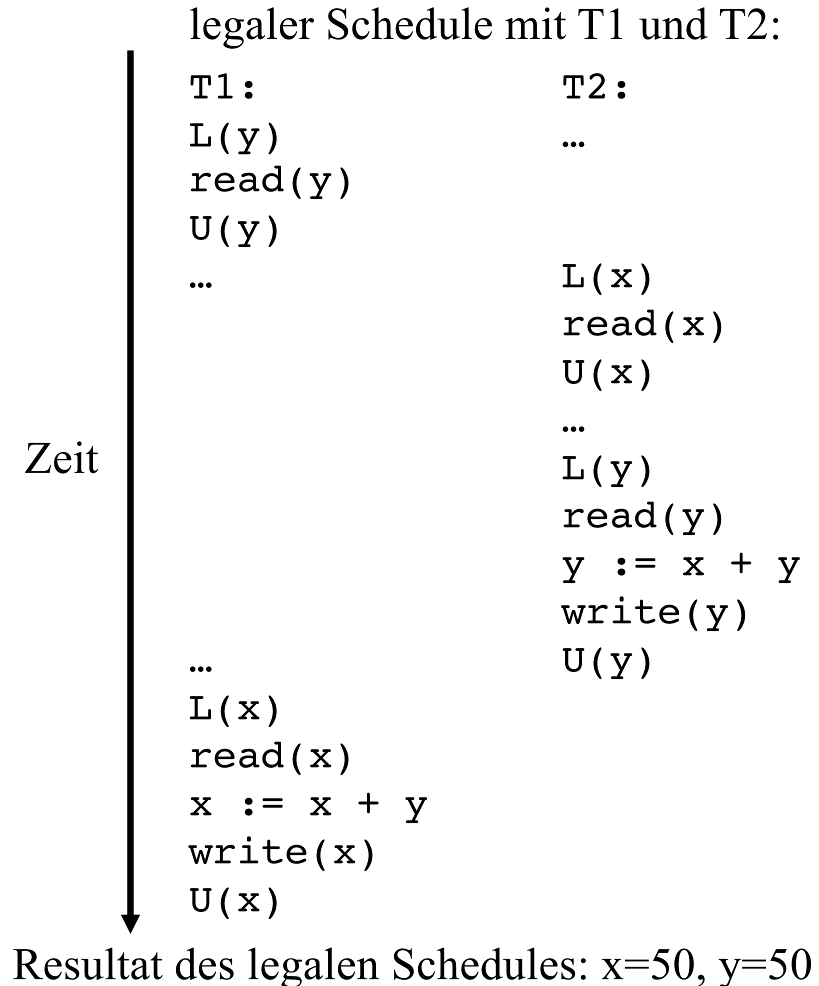
Beispiel: Legaler Schedule der nicht serialisierbar ist.

```
T1:             T2:
L(y)            L(x)
read(y)         read(x)
U(y)            U(x)
…               …
L(x)            L(y)
read(x)         read(y)
x := x + y      y := x + y
write(x)        write(y)
U(x)            U(y)
```

Anfangswerte: x=20, y=30

Resultat des seriellen Schedules
T1 gefolgt von T2:
x = 50, y = 80

Resultat des seriellen Schedules
T2 gefolgt von T1:
x = 70, y = 50

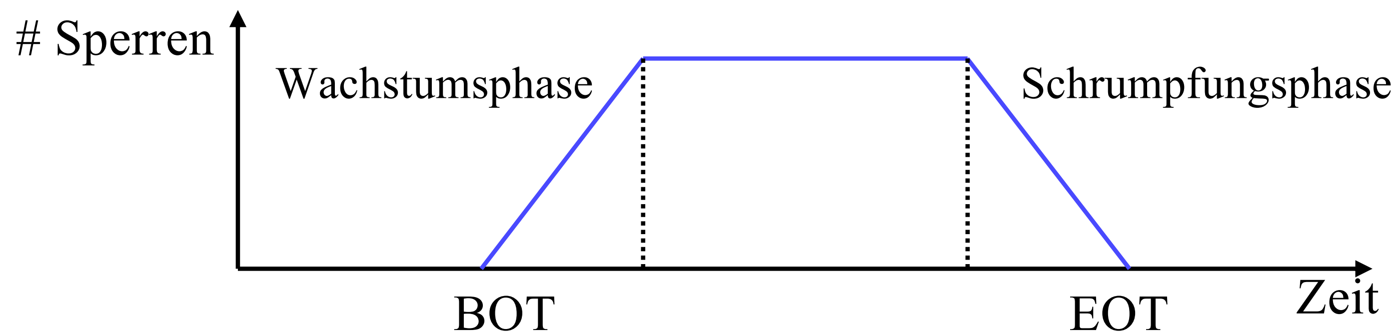


Resultat des legalen Schedules: x=50, y=50

Sperrverfahren (Locking)

Zwei-Phasen-Sperrprotokoll (2PL)

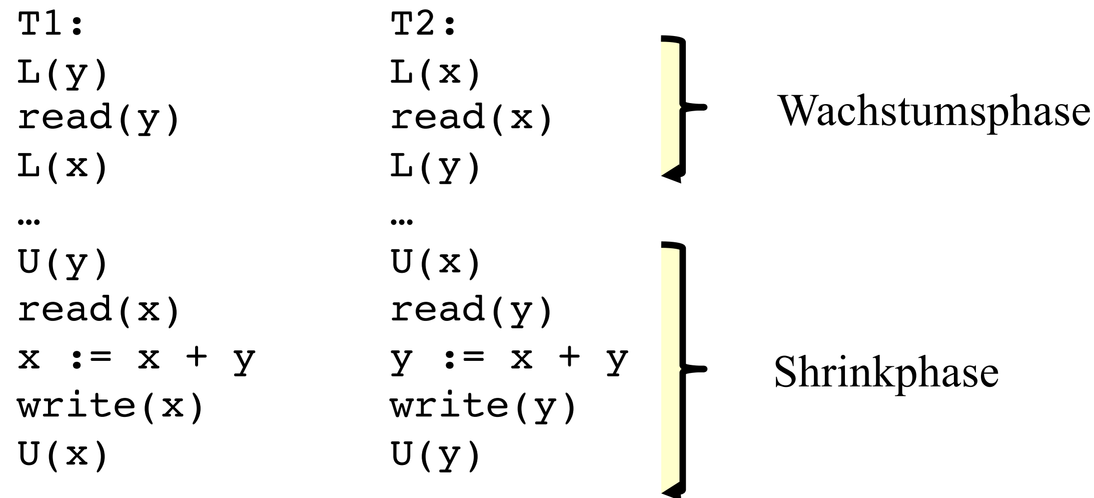
- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- Merkmal: keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
  - Wachstumsphase: Anforderungen der Sperren
  - Schrumpfungsphase: Freigabe der Sperren



CAU Christian-Albrechts-Universität zu Kiel

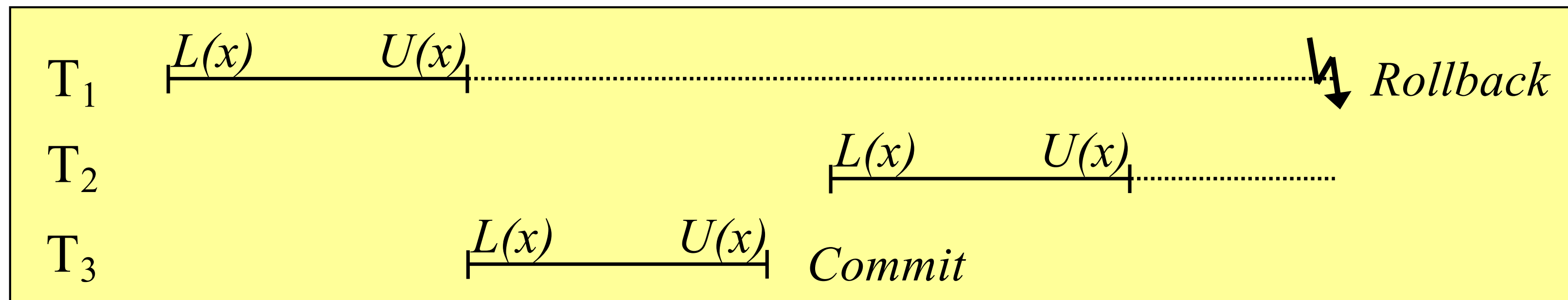
Sperrverfahren (Locking)

Beispiel: T1‘ und T2‘ entsprechen dem Zwei-Phasen-Sperrprotokoll:



Sperrverfahren (Locking)

- Zwei-Phasen-Sperrprotokoll (2PL)
  - Serialisierbarkeit ist gewährleistet, da Serialisierungsgraphen keine Zyklen enthalten können
  - Problem : Gefahr des kaskadierenden Rücksetzens im Fehlerfall (bzw. sogar nicht-rücksetzbar)



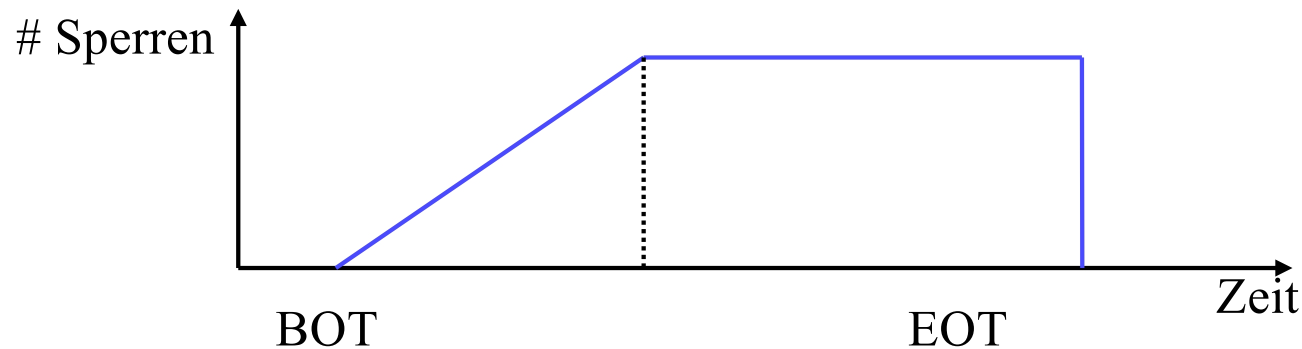
  - Transaktion T1 wird nach U(x) zurückgesetzt
  - T2 hat “schmutzig” gelesen und muss zurückgesetzt werden
  - Sogar T3 muss zurückgesetzt werden

    → Verstoß gegen die Dauerhaftigkeit (ACID) des COMMIT!

Sperrverfahren (Locking)

Striktes Zwei-Phasen-Sperrprotokoll

- Abhilfe durch striktes (oder strenges) Zwei-Phasen-Sperrprotokoll:
  - Alle Sperren werden bis zum COMMIT gehalten
  - COMMIT wird atomar (d.h. nicht unterbrechbar) ausgeführt



Sperrverfahren (Locking)

Erhöhung des Parallelisierungsgrads

- Striktes 2PL erzwingt serialisierbare, rücksetzbare Schedules
- ABER: Parallelität der TAs wird dadurch stark eingeschränkt
  - Objekt ist entweder gesperrt (und dann bis zum Commit der entspr. TA) oder zur Bearbeitung frei

    => kein paralleles Lesen oder Schreiben möglich
- Beobachtung: Parallelität unter Lesern könnte man eigentlich erlauben, da hier die Isoliertheit der beteiligten TAs nicht verletzt wird
- Daher statt 1 nun 2 Arten von Sperren
  - Lesesperren oder R-Sperren (read locks)
  - Schreibsperren oder X-Sperren (exclusive locks)

Sperrverfahren (Locking)

- Sperr-Typen:
  - Binäres Sperrschema:
    - Höchstens eine Transaktion darf exklusiv ein Objekt sperren.
    - Zu restriktiv bei sich konkurrierenden Transaktionen.
    - Z.B. wenn mehrere Transaktionen ein Objekt nur lesen wollen (vgl. S. 50).
    - => Mehrfachmodussperre
  - Mehrfachmodus-Sperrschema (RX-Sperrverfahren):
    - Erlaubt gemeinsamen Zugriff von mehreren Transaktionen die nur lese-Aktionen haben.
    - Nur exklusive Sperre wenn eine Transaktion ein Objekt schreiben will.
    - Unterscheidung zwischen gemeinsamen Sperren und exklusiven Sperren.
    - Ein Objekt x hat drei Sperr-Zustände (Zustände von LOCK(x)):
      - read_lock(x) (Lesesperre, gemeinsame Sperre)
      - write_lock(x) (Schreibsperre, exklusive Sperre)
      - unlock(x) (entsperrt)

Sperrverfahren (Locking)

RX-Sperrverfahren

- R- und X-Sperren
- Parallelität unter Lesern erlaubt
- Verträglichkeit der Sperrentypen (siehe Tabelle rechts)

angeforderte Sperre \ *bestehende Sperre*	***R***	***X***
R	+	-
X	-	-

Serialisierungsreihenfolge bei RX

- RX-Sperrverfahren meist in Verbindung mit striktem 2PL um nur kaskadenfreie rücksetzbare Schedules zu erhalten
- Zur Erinnerung: Die Reihenfolge der Transaktionen im „äquivalenten seriellen Schedule“ ist die Serialisierungsreihenfolge.
- Bei RX-Sperrverfahren (in Verbindung mit striktem 2PL) wird die Serialisierungsreihenfolge durch die erste auftretende Konfliktoperation festgelegt.

Sperrverfahren (Locking)

- Verwaltung der Sperren im Mehrfachmodus-Sperrschema
  – Jedem Datensatz/Objekt x ist ein Array mit vier Feldern zugeordnet:

x:

OID	LOCK	# Leseoperationen	Sperrende Transaktionen

- OID: Objektname.
- LOCK: Sperrzustand (read_lock, write_lock, unlock).
- # Leseoperationen: Anzahl von aktuellen Leseoperationen (read_locks).
- Sperrende Transaktionen: Liste von TAs die gemeinsame Sperre (bei nur Lesezugriffen) registriert haben oder nur eine Transaktion mit exklusiver Sperre (bei Schreibzugriff).

Sperrverfahren (Locking)

– Regeln für (Ent-)Sperrbeantragung jeder Transaktion T:
  1. read_lock(x) oder write_lock(x) **vor** read(x) von T ($r_T(x)$).
  2. write_lock(x) **vor** write(x) von T ($w_T(x)$).
  3. unlock(x) **nachdem** alle read(x) und write(x) in T abgeschlossen sind.
  4. Kein read_lock(x) beantragen wenn bereits read_lock(x) o. write_lock(x) vorhanden. (kann entschärft werden, s.u.)
  5. Kein write_lock(x) beantragen wenn bereits read_lock(x) o. write_lock(x) vorhanden. (kann entschärft werden, s.u.)
  6. Nur unlock(x) wenn read_lock(x) o. write_lock(x) vorhanden.

– Manchmal ist auch eine Änderung des Sperrentyps während dem halten einer Sperre erlaubt, z.B. read_lock→ write_lock, oder write_lock→read_lock.

Sperrverfahren (Locking)

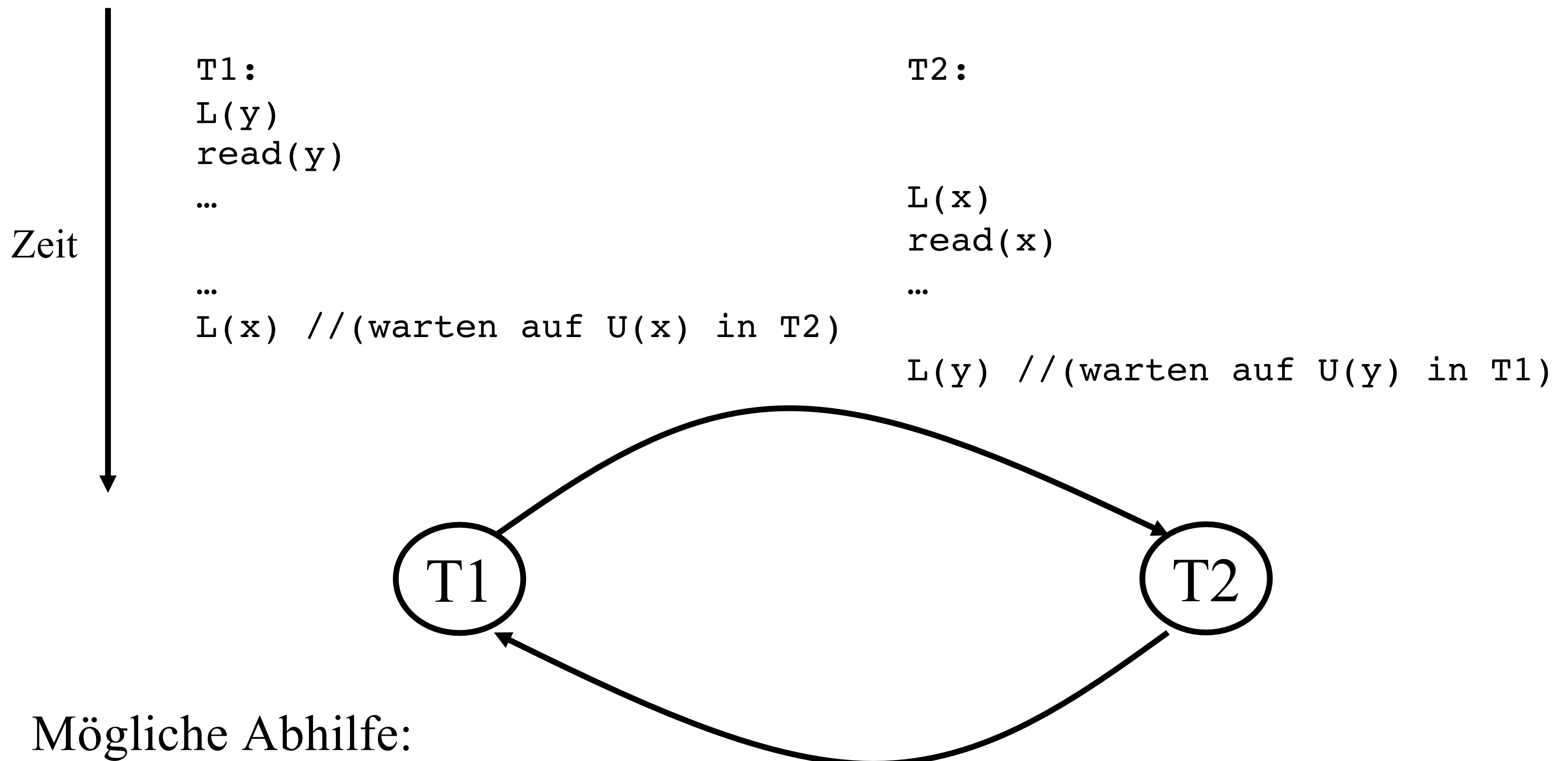
Beispiel (Serialisierungsreihenfolge bei RX):

- Situation:
  - T_1 schreibt ein Objekt x
  - Danach möchte T_2 Objekt x lesen
- Folge:
  - T_2 muss auf das COMMIT von T_1 warten, d.h. der serielle Schedule enthält T_1 vor T_2.
  - Da T_2 wartet, kommen auch alle weiteren Operationen erst nach dem COMMIT von T_1.
- Achtung:

  Grundsätzlich sind zwar auch Abhängigkeiten von T_2 nach T_1 denkbar (z.B. auf einem Objekt y), diese würden aber zu einer Verklemmung (Deadlock, gegenseitiges Warten) führen.

Sperrverfahren (Locking)

Beispiel für Verklemmung:



Mögliche Abhilfe:

Alle Transaktionen fordern gleich zu Beginn eine Sperre auf alle Objekte an, auf die sie später zugreifen (möglichst atomar).

Sperrverfahren (Locking)

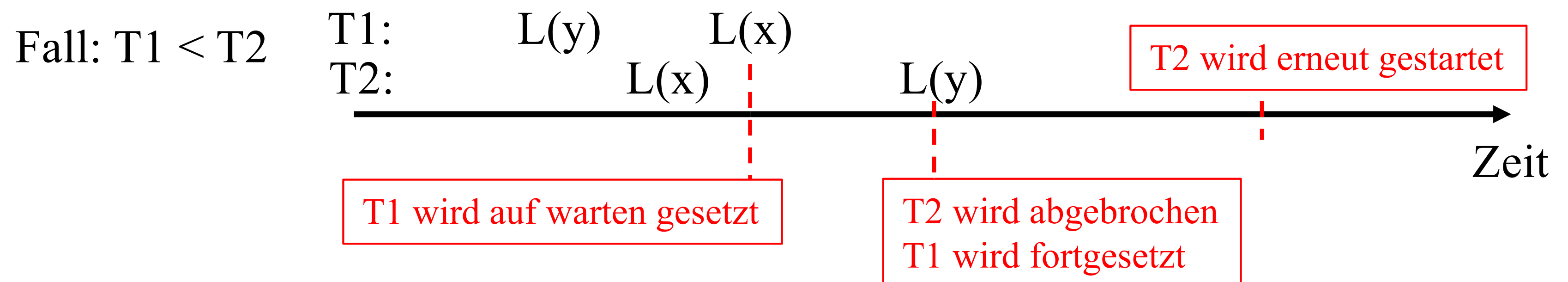
Konzepte zur Vermeidung von Verklemmung:

- Protokolle/Methoden:
  - Konservatives Zwei-Phasen-Sperrtheorem
    - TA sperrt alle Objekte auf die TA zugreift im voraus.
    - Falls eines dieser Objekte nicht verfügbar:
      - Kein Objekt wird gesperrt
      - TA wartet etwas und startet später einen neuen Versuch.
    - Zu hohe Einschränkung des Mehrbenutzerbetriebs → nicht praktikabel
  - Sperrprotokoll über globale Ordnung der Objekte
    - Alle Objekte werden global in der DB geordnet.
    - TAs dürfen die Objekte nur gemäß dieser Ordnung sperren.
    - Ebenfalls hohe Einschränkung des Mehrbenutzerbetriebs.

Sperrverfahren (Locking)

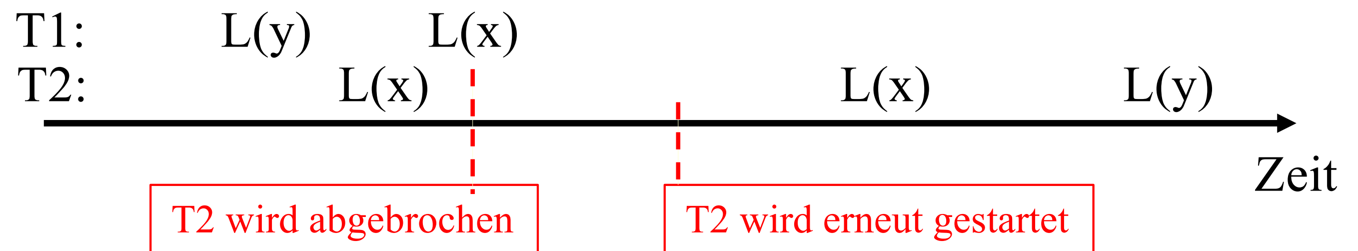
Konzepte zur Vermeidung von Verklemmung:

- Protokolle/Methoden (cont.):
  - Konzepte basierend auf Transaktionszeitstempel (TS)
    - TS(T1) < TS(T2) bedeutet Transaktion T1 startet vor T2.
    - Ausgangslage: T1 versucht Sperre auf Objekt x anzufordern, wobei Objekt x bereits von T2 gesperrt ist.
    - Schema - **Wait/Die** (Warten/Sterben):
      - Wenn TS(T1) < TS(T2), dann darf T1 warten.
      - Wenn TS(T1) > TS(T2), dann wird T1 abgebrochen und T1 startet später erneut (aber mit gleichem Zeitstempel).
      - Beispiel:



Sperrverfahren (Locking)

- Schema - **Wound/Wait** (Verwunden/Warten):
  - Wenn TS(T1) < TS(T2), dann wird T2 abgebrochen und startet später erneut mit gleichem Zeitstempel.
  - Wenn TS(T1) > TS(T2), dann darf T1 warten.
  - Beispiel:



- Beide Schemen Wait/Die und Wound/Wait sind verklemmungsfrei.

Wie kann man das zeigen?

Nachteil dieser Konzepte:

TAs können unnötig abgebrochen werden.

Sperrverfahren (Locking)

- Konzept zur Verklemmungserkennung mittels Warte-Graphen
  - Idee:
    - Bilde Graphen über Warte-Abhängigkeiten zwischen TAs
    - Überprüfe Graphen auf Zyklen
  - Warte-Graph (Wait-for Graph)
    - Knoten: TAs
    - Kante: Eine TA versucht Sperre auf Objekt x anzufordern, wobei Objekt x bereits von einer anderen TA gesperrt ist.
  - Sehr praktikable Lösung wenn wenige Konfliktabhängigkeiten auftreten, z.B. wenn Transaktionen sehr kurz sind.
- Einfaches Konzept: Abbruch nach langem warten!!!

- Man spricht von Verhungern, wenn TAs längere Zeit auf andere TAs durch unfaire Warteschema warten müssen.

→ faires Warteschema: z.B. FIFO (first-come-first-serve).