

# 184.686 VU Datenbanksysteme

## Transaktionen

Felix Winter (Folien von Katja Hose)

Institut für Logic and Computation

Sommersemester 2024



# Lernziele: Transaktionen und Schedules

## Lernziele

- Das Konzept der Transaktion verstehen
- Die ACID Eigenschaften verstehen
- Serialisierbarkeit verstehen
- Das Konzept von Schedules (Historien) verstehen
- Recoverable Schedules und Cascadeless Schedules verstehen

## Motivation

- Transaktionsgrenzen sind ein wichtiger Teil des System Designs
- Benutzer denken in Transaktionen
- Umgang mit Mehrbenutzersystemen

# Übersicht I

- 1 Transaktionen
  - Eigenschaften
  - Operationen auf Transaktionsebene
  - Umsetzung der ACID-Eigenschaften
- 2 Schedules und Serialisierbarkeit
  - Schedules
  - Conflict Serializability
  - Conflict Graphs (Precedence Graphs)
  - Recoverable Schedules und Cascadeless Schedules
- 3 Mehrbenutzersynchronisation
  - Sperrbasierte Synchronisation
  - Zwei-Phasen-Sperrprotokoll (2PL)
  - Lock Konvertierung
  - Erkennung von Deadlocks

# Übersicht II

- Vermeidung von Deadlocks

- 4 Recovery
  - Fehlerklassifikation
  - Datenspeicher
  - Log-Einträge
  - Log-Based Recovery

# Einführung

Beispiel einer typischen Bankanwendung:

- ① Lese den Kontostand von A in die Variable  $a$ :  $read(A, a);$
- ② Reduziere den Kontostand um 50 Euro:  $a := a - 50;$
- ③ Schreibe den neuen Kontostand in die Datenbank:  $write(A, a);$
- ④ Lese den Kontostand von B in die Variable  $b$ :  $read(B, b);$
- ⑤ Erhöhe den Kontostand um 50 Euro:  $b := b + 50;$
- ⑥ Schreibe den neuen Kontostand in die Datenbank:  $write(B, b);$

Was könnte ein Problem verursachen?

# Einführung

Beispiel einer typischen Bankanwendung:

- ① Lese den Kontostand von A in die Variable  $a$ :  $read(A, a);$
- ② Reduziere den Kontostand um 50 Euro:  $a := a - 50;$
- ③ Schreibe den neuen Kontostand in die Datenbank:  $write(A, a);$
- ④ Lese den Kontostand von B in die Variable  $b$ :  $read(B, b);$
- ⑤ Erhöhe den Kontostand um 50 Euro:  $b := b + 50;$
- ⑥ Schreibe den neuen Kontostand in die Datenbank:  $write(B, b);$

- Alle Schritte müssen als Einheit betrachtet werden  
“Alles oder nichts” Prinzip
- Sobald abgeschlossen, müssen die Änderungen permanent gespeichert werden
- ...

# Was ist eine Transaktion?

Eine **Transaktion** ist die Bündelung mehrerer Datenbankoperationen die eine **logische Einheit** an Arbeit bilden. In einer Transaktion wird auf verschiedene Datenbankeinträge zugegriffen, wobei einige Einträge möglicherweise verändert werden.

Transaktionsgrenzen werden durch den Benutzer  
(bzw. die Softwareanwendung) definiert!

# Eigenschaften von Transaktionen: ACID-Eigenschaften

## Atomicity (Atomarität)

- Entweder es werden alle Operationen einer Transaktion korrekt in der Datenbank umgesetzt oder gar keine.
- Wird typischerweise mit Log-Einträgen implementiert.

## Consistency (Konsistenz)

- Die Ausführung einer Transaktion in Isolation erhält den konsistenten Zustand der Datenbank.
- Entsprechend der definierten constraints, checks, assertions
- Auch die Anwendung definiert Konsistenz: z.B. sollten Banküberweisungen kein Geld generieren oder zerstören - die Gesamtsumme sollte vorher und nachher gleich sein.



# Eigenschaften von Transaktionen: ACID-Eigenschaften

## Isolation

- Jede Transaktion hat die DB “für sich allein”
- Zwischenresultate dürfen für andere Transaktionen nicht sichtbar sein
- Typischerweise mittels Locks (Sperren) implementiert

## Durability (Dauerhaftigkeit)

- Änderungen von erfolgreich abgeschlossenen Transaktionen dürfen auch bei Systemfehlern nicht verloren gehen
- Typischerweise mit Log-Einträgen implementiert

# Menti

## Frage 3

# Übersicht

- 1 Transaktionen
  - Eigenschaften
  - Operationen auf Transaktionsebene
  - Umsetzung der ACID-Eigenschaften

# Operationen auf Transaktionsebene

## begin of transaction (BOT)

Repräsentiert den Beginn einer Transaktion, d.h., alle Folgebefehle zusammen bilden eine Transaktion.

In SQL        `BEGIN;`

## commit

Repräsentiert das Ende einer Transaktion, d.h., alle Änderungen werden festgeschrieben und sind auch für andere sichtbar.

In SQL        `COMMIT;`

## rollback oder abort

Führt zu einem “roll back” der Transaktion, d.h., alle Änderungen werden zurückgesetzt/verworfen.

In SQL        `ROLLBACK;`

# Operationen auf Transaktionsebene

“autocommit” Modus

Jeder Befehl wird in einer eigenen Transaktion ausgeführt

## Einfache Konsistenzprüfungen (Checks)

```
CREATE TABLE emp(  
  eid      INT          PRIMARY KEY,  
  ename    VARCHAR(30) NOT NULL,  
  salary   INT          NOT NULL CHECK (salary > 0)  
);
```

```
-- primary key violation  
insert into emp values (11, 'Kim', 200);  
-- Not null constraint violation  
insert into emp values (44, NULL, 200);  
-- Check statement violation  
insert into emp values (44, 'Kim', -200);
```

- Viele Fehler können vom DBMS erkannt werden!
- Verwenden Sie diese Checks!

# Sicherungspunkte

Transaktionen von langer Dauer können zusätzlich Sicherungspunkte (Savepoints) definieren

`SAVEPOINT savepoint_name;`

Definiert einen Punkt/Zustand innerhalb einer Transaktion

Eine Transaktion kann bis zum Sicherungspunkt **teilweise zurückgerollt werden**.

`ROLLBACK TO savepoint_name;`

setzt die aktive Transaktion zurück bis zum Sicherungspunkt  
savepoint\_name

# Beispiel

```
BEGIN;  
INSERT INTO tab VALUES...  
SAVEPOINT A;  
INSERT INTO tab VALUES...  
SAVEPOINT B;  
SELECT * FROM tab;  
ROLLBACK TO A;  
SELECT * FROM tab;  
...
```

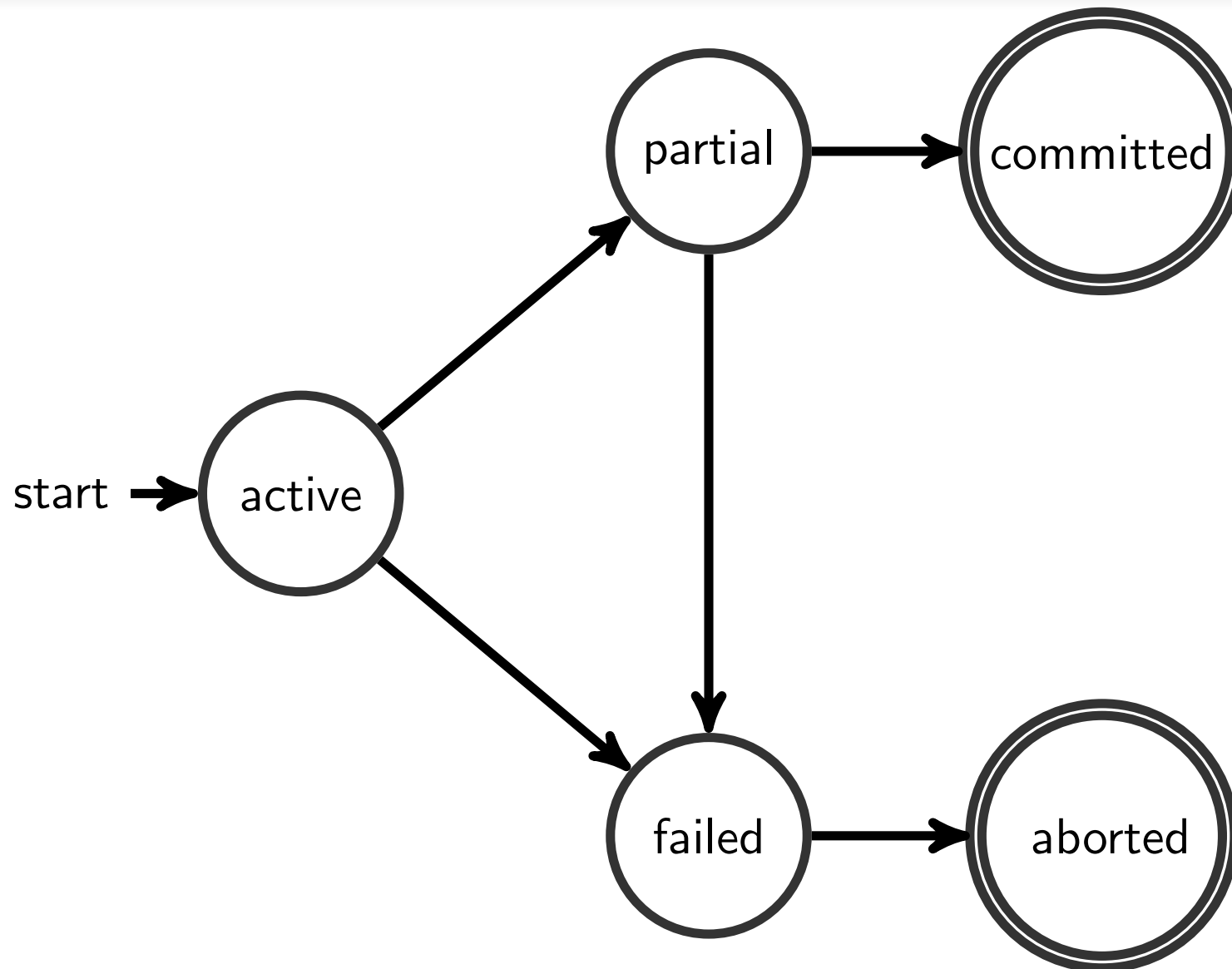


# Abschluss einer Transaktion

Für den Abschluss einer Transaktion gibt es drei Möglichkeiten:

- ① Den **erfolgreichen** Abschluss durch **commit**
- ② Den **erfolglosen** Abschluss durch ein **abort**
- ③ Den **erfolglosen** Abschluss durch einen **Fehler**

# Zustandsdiagramm für Transaktionen



# Wie werden Transaktionen vom DBMS realisiert?

Die beiden wichtigsten Komponenten der Transaktionsverwaltung sind

## Mehrbenutzersynchronisation (Isolation)

- Semantische Korrektheit bei Nebenläufigkeit  
Nebenläufigkeit erlaubt einen hohen Durchsatz
- Serialisierbarkeit
- Schwächere Isolationsstufen (Isolation Levels)

## Recovery (Atomicity und Durability)

- Zurücksetzen teilweise ausgeführter Transaktionen
- Wiederausführung von Transaktionen nach Ausfällen (Failures)
- Sicherstellen der Persistenz von Änderungen durch Transaktionen

# Menti

Fragen 4–7

# Übersicht

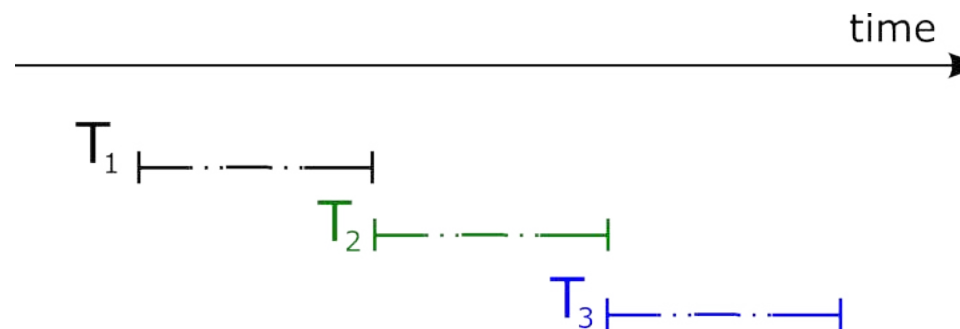
- 2 Schedules und Serialisierbarkeit
  - Schedules
  - Conflict Serializability
  - Conflict Graphs (Precedence Graphs)
  - Recoverable Schedules und Cascadeless Schedules

## Nebenläufigkeit (Parallelität)

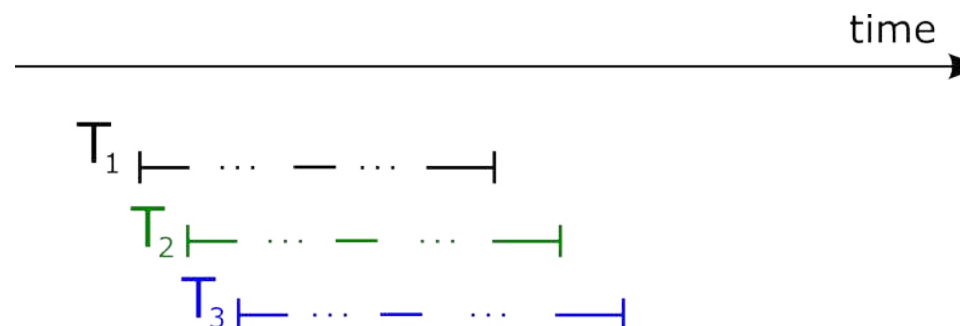
Das „I“ in ACID.

Ausführung der drei Transaktionen  $T_1$ ,  $T_2$ , und  $T_3$

(a) im Einzelbetrieb



(b) im (nebenläufigen) Mehrbenutzerbetrieb mit verzahnter Ausführung



# Probleme bei nebenläufiger Ausführung

Wo liegt hier das Problem?

| Schritt | $T_1$              | $T_2$              |
|---------|--------------------|--------------------|
| 1.      | read(A,a1)         |                    |
| 2.      | $a1 := a1 - 300$   |                    |
| 3.      |                    | read(A,a2)         |
| 4.      |                    | $a2 := a2 * 1.03$  |
| 5.      |                    | <b>write(A,a2)</b> |
| 6.      | <b>write(A,a1)</b> |                    |
| 7.      | read(B,b1)         |                    |
| 8.      | $b1 := b1 + 300$   |                    |
| 9.      | write(B,b1)        |                    |

# Probleme bei nebenläufiger Ausführung

**Lost Updates** (verlorengegangene Änderungen durch Überschreiben)

| Schritt | $T_1$                             | $T_2$                             |
|---------|-----------------------------------|-----------------------------------|
| 1.      | read(A, $a_1$ )                   |                                   |
| 2.      | $a_1 := a_1 - 300$                |                                   |
| 3.      |                                   | read(A, $a_2$ )                   |
| 4.      |                                   | $a_2 := a_2 * 1.03$               |
| 5.      |                                   | <b>write(A, <math>a_2</math>)</b> |
| 6.      | <b>write(A, <math>a_1</math>)</b> |                                   |
| 7.      | read(B, $b_1$ )                   |                                   |
| 8.      | $b_1 := b_1 + 300$                |                                   |
| 9.      | write(B, $b_1$ )                  |                                   |



# Probleme bei nebenläufiger Ausführung

Wo liegt hier das Problem?

| Steps | $T_1$                             | $T_2$                            |
|-------|-----------------------------------|----------------------------------|
| 1.    | read( $A, a_1$ )                  |                                  |
| 2.    | $a_1 := a_1 - 300$                |                                  |
| 3.    | <b>write(<math>A, a_1</math>)</b> |                                  |
| 4.    |                                   | <b>read(<math>A, a_2</math>)</b> |
| 5.    |                                   | $a_2 := a_2 * 1.03$              |
| 6.    |                                   | write( $A, a_2$ )                |
| 7.    | read( $B, b_1$ )                  |                                  |
| 8.    | ...                               |                                  |
| 9.    | abort                             |                                  |

# Probleme bei nebenläufiger Ausführung

**Dirty Read** (Abhängigkeit von nicht freigegebenen Änderungen)

| Steps | $T_1$                             | $T_2$                            |
|-------|-----------------------------------|----------------------------------|
| 1.    | read( $A, a_1$ )                  |                                  |
| 2.    | $a_1 := a_1 - 300$                |                                  |
| 3.    | <b>write(<math>A, a_1</math>)</b> |                                  |
| 4.    |                                   | <b>read(<math>A, a_2</math>)</b> |
| 5.    |                                   | $a_2 := a_2 * 1.03$              |
| 6.    |                                   | write( $A, a_2$ )                |
| 7.    | read( $B, b_1$ )                  |                                  |
| 8.    | ...                               |                                  |
| 9.    | abort                             |                                  |

## Probleme bei nebenläufiger Ausführung

Wo liegt hier das Problem?

| $T_1$   | $T_2$  |
|---|--|
| <b>update</b> account<br><b>set</b> balance=42000<br><b>where</b> accountID=12345 | <b>select</b> sum(balance)<br><b>from</b> account<br><br><b>select</b> sum(balance)<br><b>from</b> account |

## Probleme bei nebenläufiger Ausführung

**Non-Repeatable Read** (Abhängigkeit von anderen Updates)

| $T_1$   | $T_2$  |
|---|--|
| <b>update</b> account<br><b>set</b> balance=42000<br><b>where</b> accountID=12345 | <b>select</b> sum(balance)<br><b>from</b> account<br><br><b>select</b> sum(balance)<br><b>from</b> account |

# Probleme bei nebenläufiger Ausführung

Wo liegt hier das Problem?

| $T_1$  | $T_2$   |
|--|---|
| <b>insert into account<br/>values (C,1000,...)</b> | <b>select</b> sum(balance)<br><b>from</b> account |
|  | <b>select</b> sum(balance)<br><b>from</b> account |

# Probleme bei nebenläufiger Ausführung

**Phantomproblem** (Abhängigkeit von neuen/gelöschten Tupeln)

| $T_1$  | $T_2$  |
|--|--|
| <b>insert into account<br/>values (C,1000,...)</b> | <b>select sum(balance)<br/>from account</b><br><br><b>select sum(balance)<br/>from account</b> |

# Übersicht

- 2 Schedules und Serialisierbarkeit
  - Schedules
  - Conflict Serializability
  - Conflict Graphs (Precedence Graphs)
  - Recoverable Schedules und Cascadeless Schedules

# Nebenläufigkeit und Korrektheit

Zentrales System mit gleichzeitigem Zugriff durch mehrere Benutzer

- Datenbank besteht aus zwei Einträgen: X und Y
- Einziges Kriterium für Korrektheit:  $X = Y$
- Folgende Transaktionen:

$$\begin{array}{ll} T_1 & X \leftarrow X + 1 \\ & Y \leftarrow Y + 1 \end{array} \quad \begin{array}{ll} T_2 & X \leftarrow 2 * X \\ & Y \leftarrow 2 * Y \end{array}$$

- Initial:  $X=10$  und  $Y=10$ .
- $T_1$  gefolgt von  $T_2 \Rightarrow X = 22$  and  $Y = 22$
- $T_2$  gefolgt von  $T_1 \Rightarrow X = 21$  and  $Y = 21$



# Ein Beispiel

Wert von X: 20

Wert von Y: 10

| schedule $S_0$ |                   |
|----------------|-------------------|
| $T_1$          | $T_2$             |
|                | read(X, x)        |
|                | $x \leftarrow 2x$ |
|                | write(X, x)       |
|                | read(Y, y)        |
|                | $y \leftarrow 2y$ |

# Ein Beispiel

Wert von X: 21

Wert von Y: 20

| schedule $S_0$  |  |
|---|--|
| $T_1$   | $T_2$  |
|   | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) |
| read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow y+1$ |  |

# Ein Beispiel

Wert von X: 21

Wert von Y: 21

| schedule $S_0$   |  |
|--|--|
| $T_1$  | $T_2$  |
|  | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) |
| read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) |  |

# Formale Definition eines Schedules

Ein **Schedule** (Historie) ist eine **Sequenz von Operationen** von einer oder mehreren Transaktionen.

Bei nebenläufigen Transaktionen, können deren Operationen verzahnt sein.

## Operationen

- **read(Q, q)**  
Liest den Wert des Datenobjekts Q und speichert diesen in der lokalen Variable q.
- **write(Q, q)**  
Schreibt den Wert der lokalen Variable q in das Datenobjekt Q.
- **Arithmetische Operationen**
- **commit**
- **abort**

# Formale Definition eines Schedules

Ein **Schedule** (Historie) ist eine **Sequenz von Operationen** von einer oder mehreren Transaktionen.  
Bei nebenläufigen Transaktionen, können deren Operationen verzahnt sein.

## Serieller Schedule

Operationen der Transaktionen werden sequenziell und ohne zeitliche Überlappung ausgeführt.

## Nebenläufiger Schedule

Operationen der Transaktionen werden zeitlich überlappend ausgeführt.

## Gültiger Schedule

Ein Schedule ist gültig (valid), wenn das Resultat der Ausführung “korrekt” ist.

# Menti

Fragen 8–10

## Beispiele:

| schedule $S_0$   |  | schedule $S_{0'}$  |  | schedule $S_1$   |  |
|--|--|--|--|--|--|
| $T_1$  | $T_2$  | $T_1$  | $T_2$  | $T_1$  | $T_2$  |
| read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) | read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) | read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) |

Sind diese Schedules gültige nebenläufige Schedules, ungültige nebenläufige Schedules, oder serielle Schedules? Initial:  $X=Y=10$ , Korrektheitskriterium:  $X=Y$

## Beispiele:

| schedule $S_0$   |  | schedule $S_{0'}$  |  | schedule $S_1$   |  |
|--|--|--|--|--|--|
| $T_1$  | $T_2$  | $T_1$  | $T_2$  | $T_1$  | $T_2$  |
| read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) | read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) | read(X, x)<br>$x \leftarrow x+1$<br>write(X, x)<br><br>read(Y, y)<br>$y \leftarrow y+1$<br>write(Y, y) | read(X, x)<br>$x \leftarrow 2x$<br>write(X, x)<br>read(Y, y)<br>$y \leftarrow 2y$<br>write(Y, y) |

- $X = 21, Y = 21$

- **Serieller Schedule**

- $X = 21, Y = 21$

- **Nebenläufiger Schedule**

- $X = 22, Y = 21$

- **Ungültiger Schedule**



# Korrektheit

## Definition D1

Eine nebenläufige Ausführung von Transaktionen muss die Datenbank in einem konsistentem Zustand hinterlassen.

Geht davon aus, dass jede Transaktion, sofern diese in einem konsistenten Zustand gestartet wurde, die Datenbank in einem konsistentem Zustand hinterlässt.

## Definition D2

Eine nebenläufige Ausführung von Transaktionen muss ergebnisäquivalent zu einer seriellen Ausführung der Transaktionen sein.

Ergebnisäquivalent heißt, dass die finalen Datenbankzustände identisch sein müssen.

# Menti

## Frage 11

# Beispiel

| schedule $S_2$   |  |
|--|--|
| $T_3$  | $T_4$  |
| $\text{read}(X, x)$<br>$x \leftarrow x+1$<br><br><b><math>\text{write}(X, x)</math></b><br><br><br>$\text{read}(Y, y)$<br>$y \leftarrow y+1$<br>$\text{write}(Y, y)$ | <b><math>\text{read}(X, x)</math></b><br><br>$x \leftarrow 2x$<br><b><math>\text{write}(X, x)</math></b><br>$\text{read}(Y, y)$<br>$y \leftarrow 2y$<br><br>$\text{write}(Y, y)$ |

Initial:  $X = 10$  und  $Y = 10$

$\Rightarrow X = 20$  und  $Y = 20$

- $S_2$  ist nicht ergebnisäquivalent zu einer seriellen Ausführung von  $T_3$ ,  $T_4$
- Aber der finale Datenbankzustand ist konsistent – obwohl es einige lost updates gibt.

# Korrektheit von Schedules

Die bessere Wahl ist Definition D2:

Die Ausführungsreihenfolge ist **korrekt** wenn diese zu einer **seriellen Ausführung ergebnisäquivalent** ist.

Gegeben sei eine Menge an  $n$  nebenläufigen Transaktionen. Wie können wir effizient auf Korrektheit prüfen?

In weiterer Folge, gehen wir von vereinfachenden Annahmen aus

- **Nur reads und writes** werden verwendet um die Korrektheit festzustellen.
- Diese Annahme ist strenger als Definition D2, da noch weniger Schedules als korrekt gelten.

# Übersicht

## 2 Schedules und Serialisierbarkeit

- Schedules
- Conflict Serializability
- Conflict Graphs (Precedence Graphs)
- Recoverable Schedules und Cascadeless Schedules

# Eine vierte<sup>1</sup> Definition von Korrektheit: Conflict Serializability (Konfliktserialisierbarkeit)

## Definition (D4<sup>1</sup>)

Ein Schedule ist **conflict serializable** (konfliktserialisierbar) wenn er **konfliktäquivalent** zu einem seriellen Schedule ist.

---

<sup>1</sup>Die dritte Definition (D3) ist Sichtenserialisierbarkeit.

# Mögliche Konflikte zwischen Transaktionen

Konflikte zwischen Paaren von Transaktionen ( $T_1$  und  $T_2$ ) und deren Befehlen.

| schedule $S_A$ |            |
|----------------|------------|
| $T_1$          | $T_2$      |
| write(X, x)    | read(X, x) |

**Konflikt**

| schedule $S_B$ |             |
|----------------|-------------|
| $T_1$          | $T_2$       |
| write(X, x)    | write(X, x) |

**Konflikt**

| schedule $S_C$ |            |
|----------------|------------|
| $T_1$          | $T_2$      |
| write(X, x)    | read(X, x) |

**Konflikt**

| schedule $S_D$ |            |
|----------------|------------|
| $T_1$          | $T_2$      |
| read(X, x)     | read(X, x) |

**Kein Konflikt**

## Eine vierte Definition von Korrektheit: Conflict Serializability (Konfliktserialisierbarkeit)

### Definition (D4)

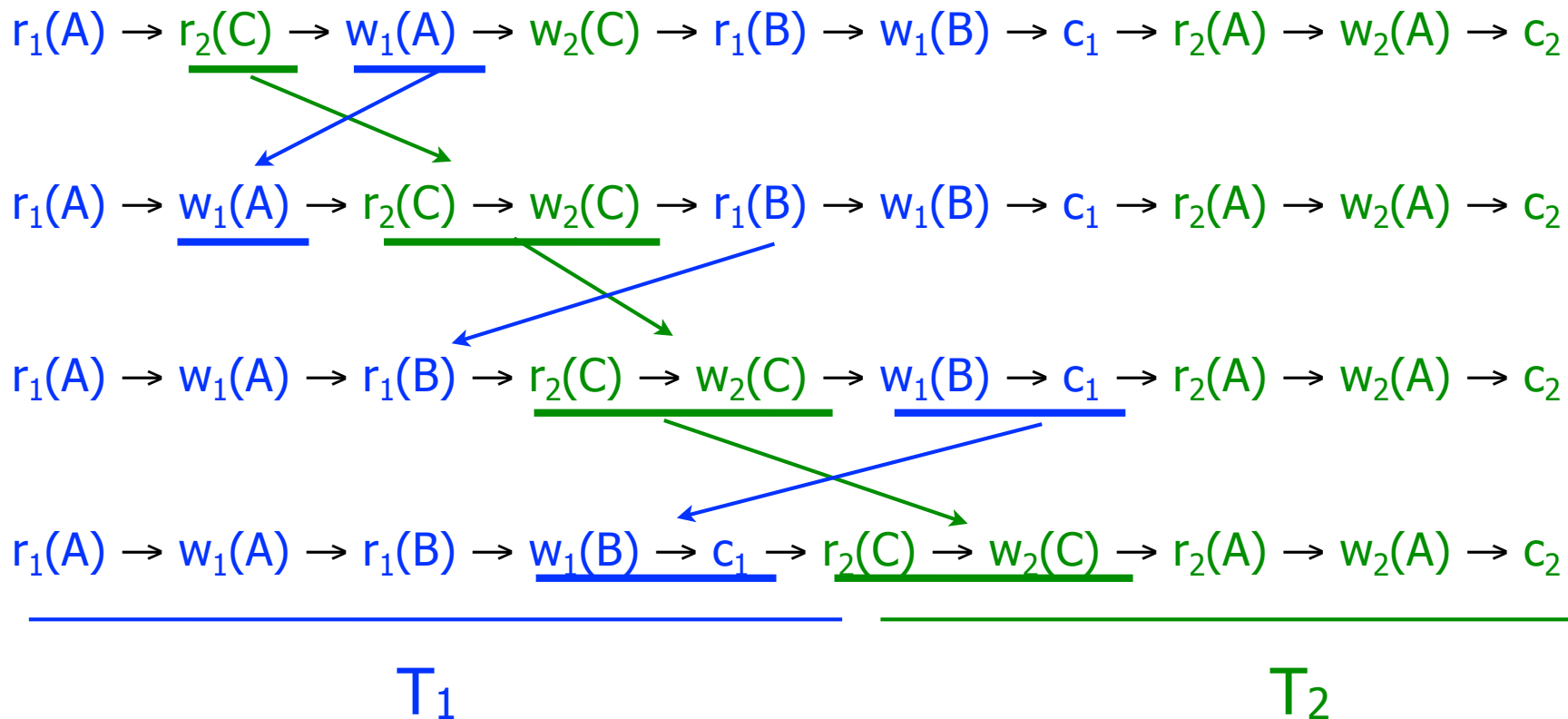
Ein Schedule ist **conflict serializable** wenn er **konfliktäquivalent** zu einem seriellen Schedule ist.

- Seien I und J aufeinanderfolgende Befehle eines Schedules S von mehreren Transaktionen.
- Wenn I und J keinen Konflikt haben, können wir deren Reihenfolge vertauschen, um einen neuen Schedule S' zu erzeugen.
- Die Befehlsreihenfolge ist in S und S' gleich, außer für I und J, deren Reihenfolge spielt keine Rolle.
- S und S' bezeichnen wir als **konfliktäquivalente Schedules**.



# Konfliktäquivalenz von zwei Schedules

Die Transformation zeigt, dass der initiale Schedule konfliktäquivalent zu einem seriellen Schedule ist. Daher ist dieser auch conflict serializable.



c ist die Abkürzung für commit, r (read), w (write)

# Menti

Fragen 12–15

# Conflict serializable oder nicht?

| schedule $S_A$ |                           |
|----------------|---------------------------|
| $T_1$          | $T_2$                     |
| read(Y, y)     | read(X, x)<br>write(X, x) |
| write(Y, y)    |                           |

Conflict serializable

| schedule $S_C$ |            |
|----------------|------------|
| $T_5$          | $T_6$      |
| read(X, x)     | read(X, x) |
| write(X, x)    |            |

Conflict serializable

| schedule $S_B$ |                           |
|----------------|---------------------------|
| $T_3$          | $T_4$                     |
| read(X, x)     | read(X, x)<br>write(X, x) |
| write(X, x)    |                           |

Nicht conflict serializable

| schedule $S_D$ |             |
|----------------|-------------|
| $T_7$          | $T_8$       |
| read(X, x)     | write(X, x) |
| write(X, x)    |             |

Nicht conflict serializable

# Übersicht

## 2 Schedules und Serialisierbarkeit

- Schedules
- Conflict Serializability
- Conflict Graphs (Precedence Graphs)
- Recoverable Schedules und Cascadeless Schedules

## Conflict Graph (Konfliktgraph)

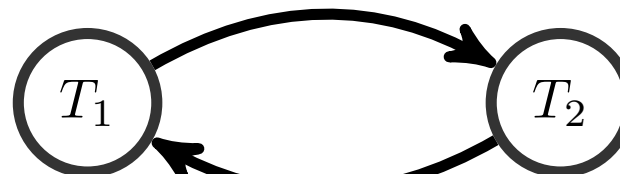
Wir erzeugen einen gerichteten Graphen (Conflict Graph oder Precedence Graph) für einen Schedule.

Annahme: Eine Transaktion führt auf ein Datenobjekt immer ein Read aus, bevor auf das gleiche Objekt eine Write ausgeführt wird.

Gegeben sei ein Schedule für die Transaktionen  $T_1, T_2, \dots, T_n$

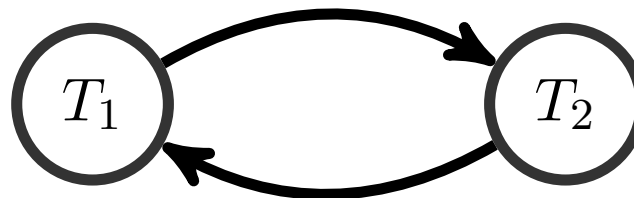
- Die Knoten des Conflict Graph sind die Transaktions-Ids.
- Eine Kante von  $T_i$  nach  $T_j$  zeigt einen Konflikt zwischen  $T_i$  und  $T_j$  an, wobei  $T_i$  den relevanten Zugriff früher durchführt.
- Manchmal werden Kantenlabels mit dem Namen des involvierten Datenobjekts verwendet.

Beispiel für einen Conflict Graph des Schedules  $S_1$



# Serialisierbarkeit feststellen

Gegeben sei ein Schedule  $S$  und ein Conflict Graph, we können wir feststellen ob der Schedule conflict serializable ist?



## Serialisierbarkeit feststellen

Gegeben sei ein Schedule  $S$  und ein Conflict Graph, we können wir feststellen ob der Schedule conflict serializable ist?

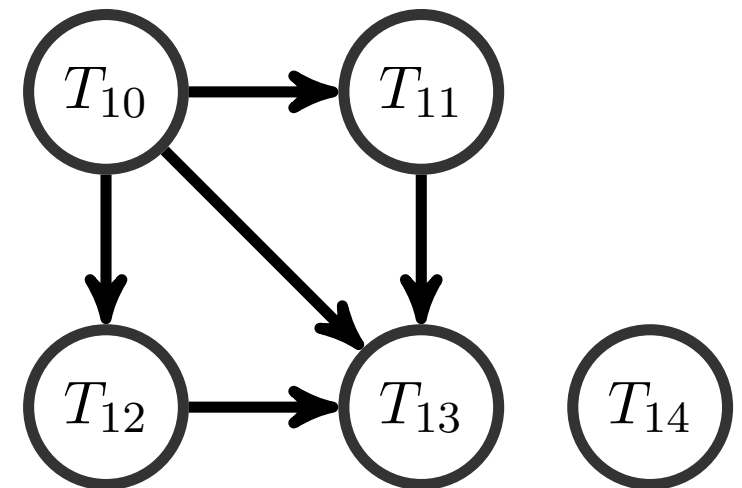
- Ein Schedule ist **conflict serializable**, wenn der Conflict Graph **acyklisch** ist.
- Intuitiv: Ein Konflikt zwischen zwei Transaktionen erzwingt eine bestimmte Ausführungsreihenfolge (topologische Sortierung)

Wir verwenden conflict serializability (und nicht eine andere Definition der Serialisierbarkeit) weil es eine praktikable Implementierungsmöglichkeit gibt.

# Beispiel Conflict Graph

schedule  $S_6$

| $T_{10}$                 | $T_{11}$                  | $T_{12}$                  | $T_{13}$   | $T_{14}$                                |
|--------------------------|---------------------------|---------------------------|--|---|
| read(Y, y)<br>read(Z, z) | read(X, x)                |                           |  | read(V, v)<br>read(W, w)<br>write(W, w) |
|                          | read(Y, y)<br>write(Y, y) |                           |  |   |
| read(T, t)               |                           | read(Z, z)<br>write(Z, z) |  |   |
|                          |                           |                           | read(Y, y)<br>write(Y, y)<br>read(Z, z)<br>write(Z, z) |   |
| read(U, u)               |                           |                           |  |   |



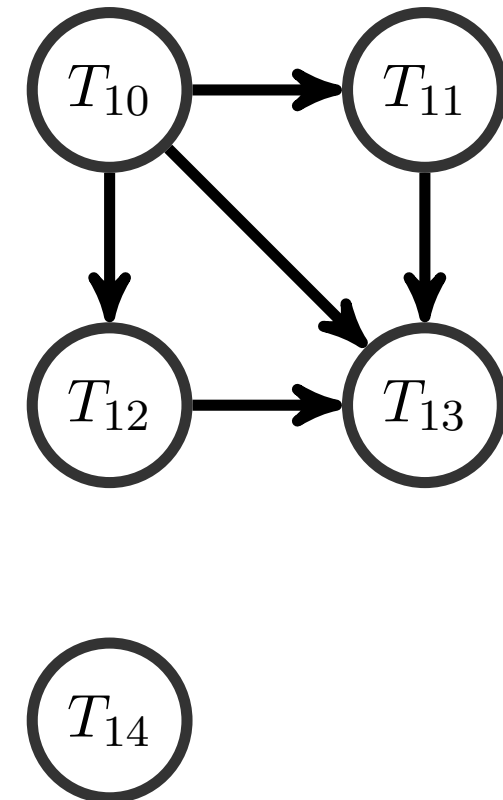


# Menti

Frage 16

# Beispiel Conflict Graph

| schedule $S_6$           |                           |                           |  |   |
|--------------------------|---------------------------|---------------------------|--|---|
| $T_{10}$                 | $T_{11}$                  | $T_{12}$                  | $T_{13}$   | $T_{14}$                                |
| read(Y, y)<br>read(Z, z) | read(X, x)                |                           |  | read(V, v)<br>read(W, w)<br>write(W, w) |
|                          | read(Y, y)<br>write(Y, y) | read(Z, z)<br>write(Z, z) |  |   |
| read(T, t)               |                           |                           | read(Y, y)<br>write(Y, y)<br>read(Z, z)<br>write(Z, z) |   |
| read(U, u)               |                           |                           |  |   |



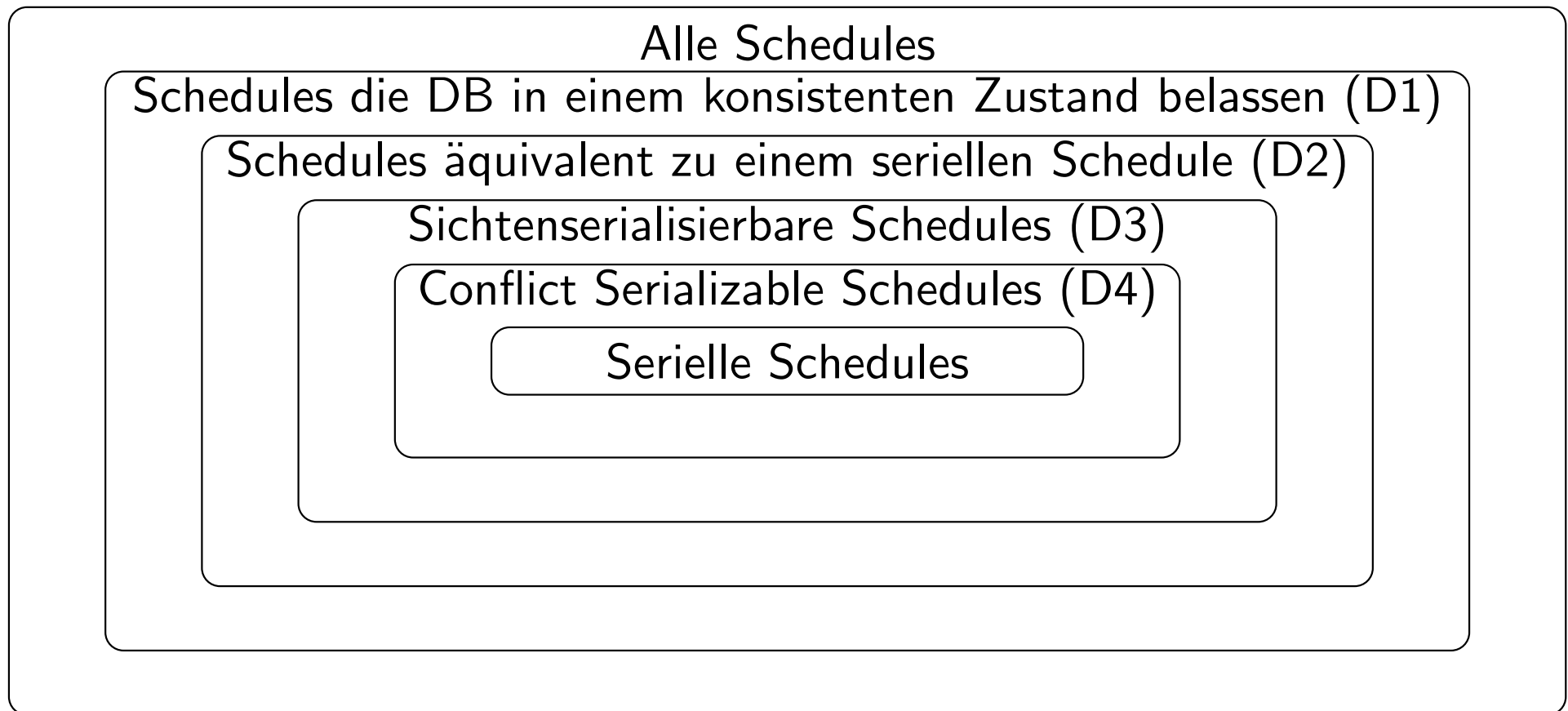
Welche der folgenden Schedules sind konfliktäquivalente serielle Schedules?

$T_{10}, T_{11}, T_{12}, T_{13},$  and  $T_{14}$  Yes

$T_{14}, T_{10}, T_{12}, T_{11},$  and  $T_{13}$  Yes

$T_{14}, T_{13}, T_{12}, T_{11},$  and  $T_{10}$  No

# Beziehungen zwischen Schedules



# Übersicht

- 2 Schedules und Serialisierbarkeit
  - Schedules
  - Conflict Serializability
  - Conflict Graphs (Precedence Graphs)
  - Recoverable Schedules und Cascadeless Schedules

## Transaktionen: Isolation und Atomicity

# Transaktionen können fehlschlagen

# Recoverable (Wiederherstellbare) Schedules

| schedule $S_A$ |             |
|----------------|-------------|
| $T_i$          | $T_j$       |
| read(X, x)     |             |
| write(X, x)    |             |
|                | read(X, x)  |
|                | write(X, x) |
|                | commit      |
| rollback       |             |

- Wenn  $T_i$  fehlschlägt, muss diese zurückgerollt werden, um die **Atomicity (Atomarität)** Eigenschaft zu erhalten (siehe Recovery).
- Wenn eine andere Transaktion  $T_j$  Daten gelesen hat welche von  $T_i$  beschrieben wurden, dann muss auch  $T_j$  zurückgerollt werden.  
 $\Rightarrow$  DBS muss sicherstellen das Schedules recoverable sind.
- Dieser Schedule ist nicht recoverable

## Recoverable (Wiederherstellbare) Schedules

Ein Schedule ist **recoverable**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt: Wenn  $T_j$  Daten liest die von  $T_i$  beschrieben wurden, dann muss  $T_i$  vor  $T_j$  committet werden.

| schedule $S_A$ |             |
|----------------|-------------|
| $T_i$          | $T_j$       |
| read(X, x)     |             |
| write(X, x)    |             |
| rollback       |             |
|                | read(X, x)  |
|                | write(X, x) |
|                | commit      |

Menti: Ist dieser Schedule recoverable?

| schedule $S_B$ |             |
|----------------|-------------|
| $T_i$          | $T_j$       |
| read(Y, y)     |             |
|                | read(X, x)  |
| write(Y, y)    |             |
|                | write(X, x) |
| rollback       |             |
|                | commit      |

Menti: Ist dieser Schedule recoverable?

# Menti

Fragen 17-18



# Recoverable (Wiederherstellbare) Schedules

Ein Schedule ist **recoverable**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt: Wenn  $T_j$  Daten liest die von  $T_i$  beschrieben wurden, dann muss  $T_i$  vor  $T_j$  committet werden.

| schedule $S_A$ |             |
|----------------|-------------|
| $T_i$          | $T_j$       |
| read(X, x)     |             |
| write(X, x)    |             |
| rollback       |             |
|                | read(X, x)  |
|                | write(X, x) |
|                | commit      |

recoverable

| schedule $S_B$ |             |
|----------------|-------------|
| $T_i$          | $T_j$       |
| read(Y, y)     |             |
|                | read(X, x)  |
| write(Y, y)    |             |
|                | write(X, x) |
| rollback       |             |
|                | commit      |

recoverable

# Cascading Rollbacks (Kaskadierendes Rücksetzen)

| schedule $S_{11}$                                      |            |                          |
|--|------------|--------------------------|
| $T_{22}$   | $T_{23}$   | $T_{24}$                 |
| read(A, a)<br>read(B, b)<br>write(A, a)<br>write(B, b) | read(A, a) | read(A, a)<br>read(B, b) |
| rollback   |            |                          |

Was passiert im Fall eines Rollbacks von  $T_{22}$ ?  
Ist dieser Schedule recoverable?

# Cascading Rollbacks (Kaskadierendes Rücksetzen)

| schedule $S_{11}$                                      |            |                          |
|--|------------|--------------------------|
| $T_{22}$   | $T_{23}$   | $T_{24}$                 |
| read(A, a)<br>read(B, b)<br>write(A, a)<br>write(B, b) | read(A, a) | read(A, a)<br>read(B, b) |
| rollback   |            |                          |

- $T_{22}$  Rollback  $\Rightarrow$  wir müssen auch  $T_{23}$  und  $T_{24}$  zurückrollen, weil diese "dirty data" lesen. (Cascading Rollback)
- Der Schedule ist nicht cascadeless (kommt nicht ohne kaskadierendes Rücksetzen aus).
- Aber der Schedule ist recoverable.

# Cascadeless Schedules

Ein Schedule ist **cascadeless**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt: Wenn  $T_j$  Daten liest die von  $T_i$  beschrieben wurden, dann muss  $T_i$  vor dem Lesezugriff von  $T_j$  bereits committed sein.

| schedule $S_A$                      |  |  |
|-------------------------------------|--|--|
| $T_1$                               | $T_2$                                    | $T_3$  |
| read(A, a)<br>write(A, a)<br>commit | <br><br><br><br><br>read(A, a)<br>commit | <br><br><br><br>read(A, a)<br><br><br>commit |

# Menti

Frage 19

# Cascadeless Schedules

Ein Schedule ist **cascadeless**, wenn für jedes Transaktionspaar  $T_i$  und  $T_j$  gilt: Wenn  $T_j$  Daten liest die von  $T_i$  beschrieben wurden, dann muss  $T_i$  vor dem Lesezugriff von  $T_j$  bereits committed sein.

| schedule $S_{11}'$   |                      |                                    |
|--|----------------------|------------------------------------|
| $T_{22}$   | $T_{23}$             | $T_{24}$                           |
| read(A, a)<br>read(B, b)<br>write(A, a)<br>write(B, b)<br>rollback | read(A, a)<br>commit | read(A, a)<br>read(B, b)<br>commit |

Dieser Schedule ist auch recoverable

Cascading Rollbacks können verhindert werden, indem nur von Transaktionen gelesen wird, die bereits committed sind.

# Cascadeless Schedules

- Jeder cascadeless Schedule ist auch recoverable.
- Cascading Rollbacks können schnell zeitaufwändig werden.
- Es ist sinnvoll sich auf Schedules zu beschränken die cascadeless sind.

# Zusammenfassung: Transaktionen und Schedules

- Jede Transaktion erhält den konsistenten Zustand der Datenbank.
- Die serielle Ausführung einer Menge an Transaktionen erhält einen konsistenten Zustand.
- Bei einer nebenläufigen Ausführung, können die einzelnen Ausführungsschritte einer Menge an Transaktionen verzahnt sein.
- Ein nebenläufiger Schedule ist serialisierbar, wenn dieser äquivalent zu einem seriellen Schedule ist.
  - Conflict Serializability  
Bevorzugte Methode, da es praktikable Implementierungsmöglichkeiten gibt
  - Conflict Graphs
- Schedules müssen recoverable sein und sollten cascadeless sein.



# Lernziele

## Lernziele: Mehrbenutzersynchronisation

- Sperrbasierte Synchronisation verstehen und verwenden
- Zwei-Phasen-Sperrprotokoll verstehen und verwenden

## Motivation

- Exklusiver Zugriff auf die Datenbank durch mehrere Benutzer hat negativen Einfluss auf Durchsatz und Laufzeit

# Übersicht I

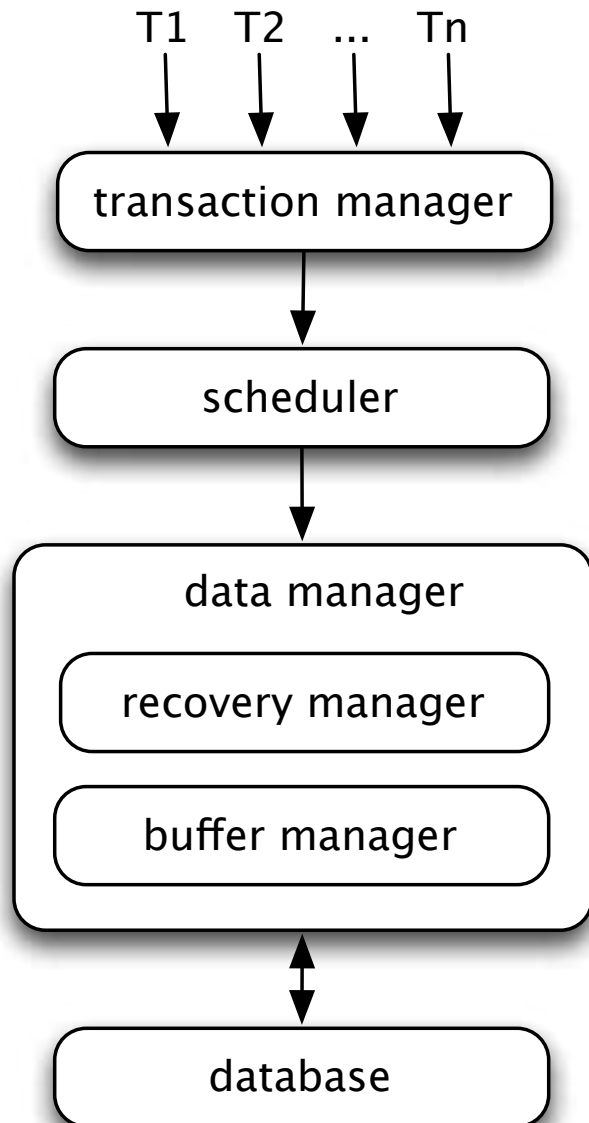
- 1 Transaktionen
  - Eigenschaften
  - Operationen auf Transaktionsebene
  - Umsetzung der ACID-Eigenschaften
- 2 Schedules und Serialisierbarkeit
  - Schedules
  - Conflict Serializability
  - Conflict Graphs (Precedence Graphs)
  - Recoverable Schedules und Cascadeless Schedules
- 3 Mehrbenutzersynchronisation
  - Sperrbasierte Synchronisation
  - Zwei-Phasen-Sperrprotokoll (2PL)
  - Lock Konvertierung
  - Erkennung von Deadlocks

# Übersicht II

- Vermeidung von Deadlocks

- 4 Recovery
  - Fehlerklassifikation
  - Datenspeicher
  - Log-Einträge
  - Log-Based Recovery

# Der Datenbank-Scheduler



Aufgabe des Schedulers:  
Operationen der Transaktionen  $T_1, \dots, T_n$   
in eine Reihenfolge bringen, die  
serialisierbar ist (und ohne  
kaskadierendes Rücksetzen auskommt).

## Synchronisationsverfahren

- Pessimistisch
  - **Sperrbasierte Synchronisation**
  - Zeitstempel-basierte Synchronisation
- Optimistisch

Basierend auf "Datenbanksysteme: Ein Einführung"  
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

# Sperrbasierte Synchronisation

**Sicherstellen von (conflict) serializable Schedules** durch **Verzögern** von Transaktionen, welche die Serialisierbarkeit verletzen würden.

Zwei Arten von Sperren können auf ein Datenobjekt Q gesetzt werden

- S (shared, read lock, Lesesperre)
- X (exclusive, write lock, Schreibsperre)

Sperroperationen

- **lock\_S(Q)** – setzt ein Shared Lock auf das Datenobjekt Q
- **lock\_X(Q)** – setzt ein Exclusive Lock auf das Datenobjekt Q
- **unlock(Q)** – Freigabe des Locks auf das Datenobjekt Q

# Sperrbasierte Synchronisation

## Privilegien bei Locks

Eine Transaktion die

- ein Exclusive Lock hält, darf einen schreibenden oder lesenden Zugriff durchführen
- ein Shared Lock hält, darf einen lesenden Zugriff durchführen

Verträglichkeitsmatrix (Kompatibilitätsmatrix)

|   | NL | S  | X |
|---|----|----|---|
| S | OK | OK | - |
| X | OK | -  | - |

NL – No Lock (keine Sperre)

- Nebenläufige Transaktionen dürfen nur kompatible Locks verwenden
- Eine Transaktion muss ggf. auf die Freigabe eines Locks warten

## Ein Beispiel

- $T_{15}$  überweist 50 EUR von Konto B auf Konto A.
- $T_{16}$  zeigt den gesamten Kontostand der Konten A und B.

| $T_{15}$              | $T_{16}$         |
|-----------------------|------------------|
| <b>lock_X(B)</b>      |                  |
| read(B, b)            |                  |
| $b \leftarrow b - 50$ | <b>lock_S(A)</b> |
| write(B, b)           | read(A, a)       |
| <b>unlock(B)</b>      | <b>unlock(A)</b> |
| <b>lock_X(A)</b>      | <b>lock_S(B)</b> |
| read(A, a)            | read(B, b)       |
| $a \leftarrow a + 50$ | <b>unlock(B)</b> |
| write(A, a)           | display(A+B)     |
| <b>unlock(A)</b>      |                  |

Gibt es ein Problem?  
→ Menti

Initial:  $A = 100$  und  $B = 200$

# Menti

## Frage 3



## Ein Beispiel

- $T_{15}$  überweist 50 EUR von Konto B auf Konto A.
- $T_{16}$  zeigt den gesamten Kontostand der Konten A und B.

| $T_{15}$   | $T_{16}$   |
|--|--|
| <b>lock_X(B)</b><br>read(B, b)<br>$b \leftarrow b - 50$<br>write(B, b)<br><b>unlock(B)</b><br><b>lock_X(A)</b><br>read(A, a)<br>$a \leftarrow a + 50$<br>write(A, a)<br><b>unlock(A)</b> | <b>lock_S(A)</b><br>read(A, a)<br><b>unlock(A)</b><br><b>lock_S(B)</b><br>read(B, b)<br><b>unlock(B)</b><br>display(A+B) |

Resultat bei serieller Ausführung:

$T_{16}$  zeigt 300

Bei diesem Schedule:

$T_{16}$  zeigt 250

Ursache des Problems:

Das Exclusive Lock auf B wurde zu früh freigegeben

Initial:  $A = 100$  und  $B = 200$

| schedule $S_7$   |  |
|--|--|
| $T_{15}$   | $T_{16}$   |
| <b>lock_X(B)</b><br>read(B, b)<br>$b \leftarrow b - 50$<br>write(B, b)<br><b>unlock(B)</b> | <b>lock_S(A)</b><br>read(A, a)<br><b>unlock(A)</b><br><b>lock_S(B)</b><br>read(B, b)<br><b>unlock(B)</b><br>display(A+B) |
| <b>lock_X(A)</b><br>read(A, a)<br>$a \leftarrow a + 50$<br>write(A, a)<br><b>unlock(A)</b> |  |

## Probleme bei zu früher Freigabe

- Initial:  $A = 100$  und  $B = 200$
- Serieller Schedule  $T_{15};T_{16}$  zeigt 300
- Serieller Schedule  $T_{16};T_{15}$  zeigt 300
- $S_7$  zeigt 250

**Frühe Sperrfreigaben** können zu **inkorrekten** Resultaten führen (Non-Serializable Schedules), aber sie erlauben einen höheren Grad an Nebenläufigkeit.

# Probleme bei später Sperrfreigabe

Lösung: Verzögern wir einfach die Sperrfreigaben bis zum Ende der Transaktion

| schedule $S_8$        |                  |
|-----------------------|------------------|
| $T_{17}$              | $T_{18}$         |
| <b>lock_X(B)</b>      |                  |
| read(B, b)            |                  |
| $b \leftarrow b - 50$ |                  |
| write(B, b)           |                  |
| ...                   |                  |
| <b>unlock(B)</b>      | <b>lock_S(A)</b> |
|                       | read(A, a)       |
|                       | ...              |
|                       | <b>unlock(A)</b> |

Ist das eine gute Lösung?  
→ Menti

# Menti

Frage 4

# Probleme bei später Sperrfreigabe

Lösung: Verzögern wir einfach die Sperrfreigaben bis zum Ende der Transaktion

| schedule $S_8$        |                  |
|-----------------------|------------------|
| $T_{17}$              | $T_{18}$         |
| <b>lock_X(B)</b>      |                  |
| read(B, b)            |                  |
| $b \leftarrow b - 50$ |                  |
| write(B, b)           |                  |
| ...                   |                  |
| <b>unlock(B)</b>      |                  |
|                       | <b>lock_S(A)</b> |
|                       | read(A, a)       |
|                       | ...              |
|                       | <b>unlock(A)</b> |

- Späte Sperrfreigaben verhindern Non-Serializable Schedules. Aber sie erhöhen die Chancen von **Deadlocks**.
- Lernen Sie damit zu leben!

# Menti

Frage 5

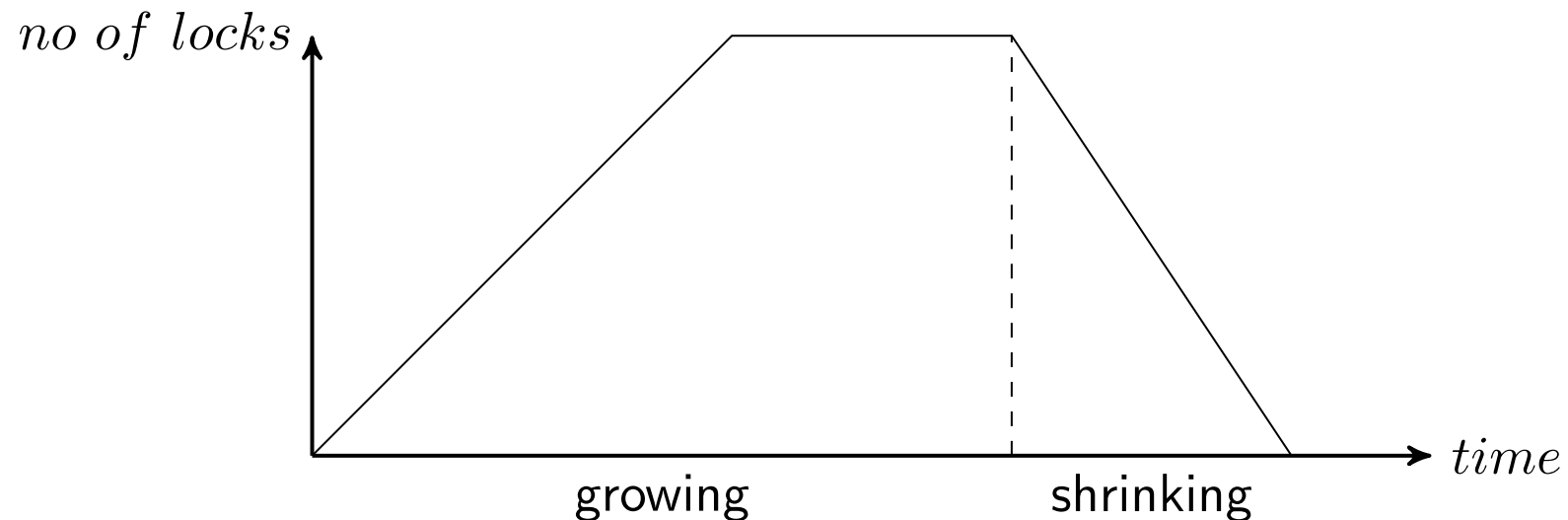
# Outline

- 3 Mehrbenutzersynchronisation
  - Sperrbasierte Synchronisation
  - Zwei-Phasen-Sperrprotokoll (2PL)
  - Lock Konvertierung
  - Erkennung von Deadlocks
  - Vermeidung von Deadlocks

# Das Zwei-Phasen-Sperrprotokoll (2PL)

- 1. Phase (Anforderungsphase, growing phase):
  - Transaktionen dürfen Locks anfordern.
  - Transaktionen dürfen keine Locks freigeben.
- 2. Phase (Freigabephase, shrinking phase):
  - Transaktionen dürfen keine Locks anfordern.
  - Transaktionen dürfen bisher erworbene Locks freigeben.

Sobald das erste Lock freigegeben wird, wechselt die Transaktion von der 1. in die 2. Phase.





# Menti

Fragen 6–9

## 2PL: Ja oder Nein?

| schedule $S_A$ | schedule $S_B$ |           | schedule $S_C$ |           | schedule $S_D$ |
|----------------|----------------|-----------|----------------|-----------|----------------|
| $T_1$          | $T_2$          | $T_3$     | $T_4$          | $T_5$     | $T_6$          |
| lock_X(A)      | lock_X(A)      |           | lock_X(A)      |           | lock_X(A)      |
| lock_X(B)      | lock_X(B)      |           |                | lock_X(B) | lock_X(B)      |
| lock_X(C)      | lock_X(C)      |           |                | lock_X(C) | unlock(B)      |
| unlock(A)      | unlock(B)      |           |                | unlock(C) | lock_X(C)      |
| unlock(C)      |                | lock_X(B) |                | unlock(B) | unlock(A)      |
| unlock(B)      | unlock(C)      |           | unlock(A)      |           | unlock(C)      |
| Ja             | unlock(A)      | unlock(B) | Ja             |           | Nein           |
|                | Ja             |           |                |           |                |

# Eigenschaften des Zwei-Phasen-Sperrprotokolls

- 2PL erzeugt nur serializable Schedules
  - Garantiert conflict serializability
  - 2PL erzeugt eine Menge aller möglichen serializable Schedules
- 2PL schützt nicht vor Deadlocks
- 2PL schützt nicht for Cascading Rollbacks
  - “Dirty” Reads sind möglich (Lesen von Transaktionen die noch nicht committed wurden)

## Cascading Rollbacks

Ein Abort einer Transaktion kann auch zu einem Abort bei anderen Transaktionen führen.

| schedule $S_{11}$   |  |                              | schedule $S_{11}'$   |  |                              |
|---|--|------------------------------|--|--|------------------------------|
| $T_{22}$  | $T_{23}$   | $T_{24}$                     | $T_{22'}$  | $T_{23'}$  | $T_{24'}$                    |
| <b>lock_X(A)</b><br><b>lock_X(B)</b><br><b>unlock(A)</b><br><br>abort | <br><br><br><b>lock_X(A)</b><br><b>unlock(A)</b> | <br><br><br><b>lock_X(A)</b> | <b>lock_X(A)</b><br><b>lock_X(B)</b><br><b>unlock(A)</b><br>commit | <br><br><br><b>lock_X(A)</b><br><b>unlock(A)</b><br>commit | <br><br><br><b>lock_X(A)</b> |

- Diese Schedules verwenden 2PL
- Abort in  $T_{22} \Rightarrow T_{23}$  und  $T_{24}$  müssen ebenfalls ein Abort auslösen

Wie könne wir diese Cascading Rollbacks verhindern?

Transaktionen dürfen keine uncommitted Daten lesen (siehe  $S_{11}'$ )

# Striktes und rigoroses Zwei-Phasen-Sperrprotokoll

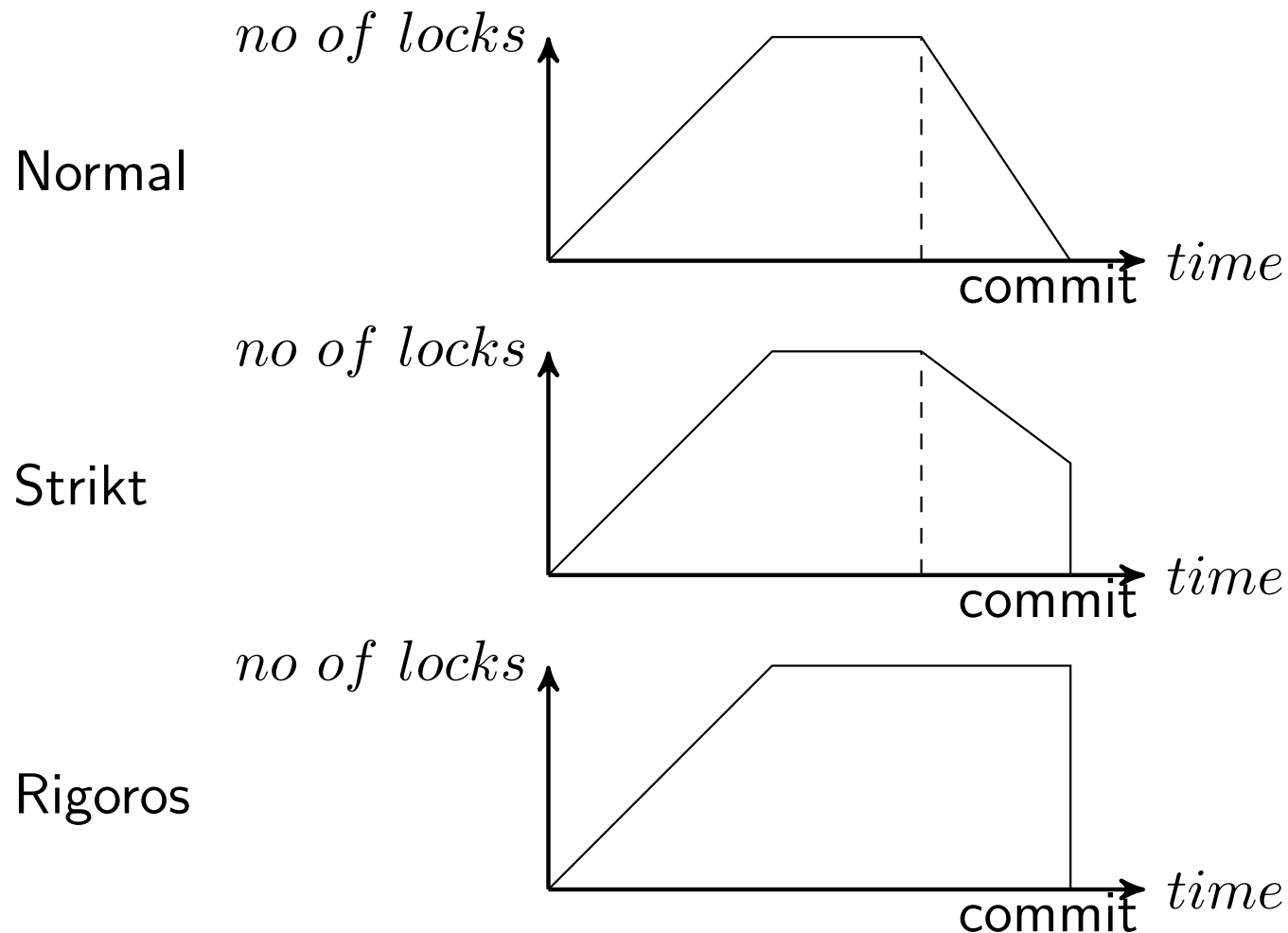
## Striktes 2PL

- **Exclusive** Locks werden nicht vor dem commit freigegeben
- Verhindert “Dirty Reads”

## Rigoroses 2PL:

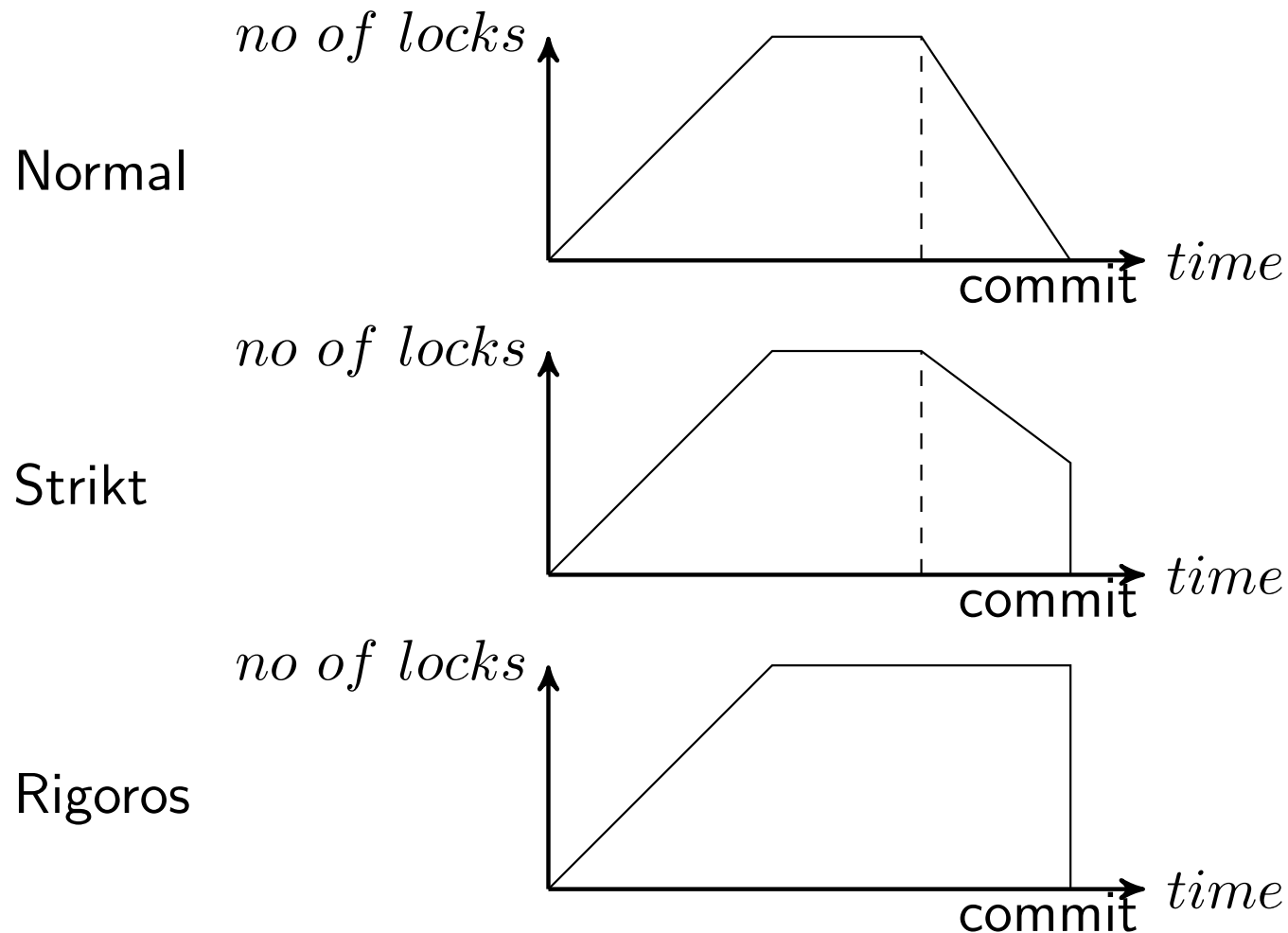
- **Alle** Locks werden erst nach dem commit freigegeben
  - Transaktionen können in der commit-Reihenfolge serialisiert werden
- 
- Vorteil:  
Keine Cascading Rollbacks
  - Nachteil:  
Weniger Nebenläufigkeit

# Übersicht: 2PL Protokolle



Was ist der Vorteil von rigorem 2PL gegenüber striktem 2PL?

# Übersicht: 2PL Protokolle



Rigorous 2PL: Transaktionen können in Commit-Reihenfolge serialisiert werden

# Lock Konvertierung

Ziel: 2PL anwenden, aber einen höheren Grad an Nebenläufigkeit erlauben

- Erste Phase
  - S-Lock auf ein Datenobjekt erhalten
  - X-Lock auf ein Datenobjekt erhalten
  - Konvertieren (upgrade) eines S-Locks zu einem X-Lock
- Zweite Phase
  - Freigabe eines S-Locks
  - Freigabe eines X-Locks
  - Konvertieren (downgrade) eines X-Locks zu einem S-Lock
- Dieses Protokoll garantiert weiterhin Serialisierbarkeit
- Ist abhängig vom Anwendungsprogrammierer (Passende Locks müssen eingefügt werden)



# Menti

Fragen 10–12

# Normales, striktes oder rigoroses 2PL?

schedule  $S_1$   
 $T_1$

lock\_S(A)  
 lock\_S(B)  
 lock\_X(B)  
 lock\_S(C)  
 unlock(A)  
 unlock(C)  
 commit

Strikt

schedule  $S_2$   
 $T_2$

lock\_S(A)  
 lock\_S(B)  
 lock\_X(B)  
 commit

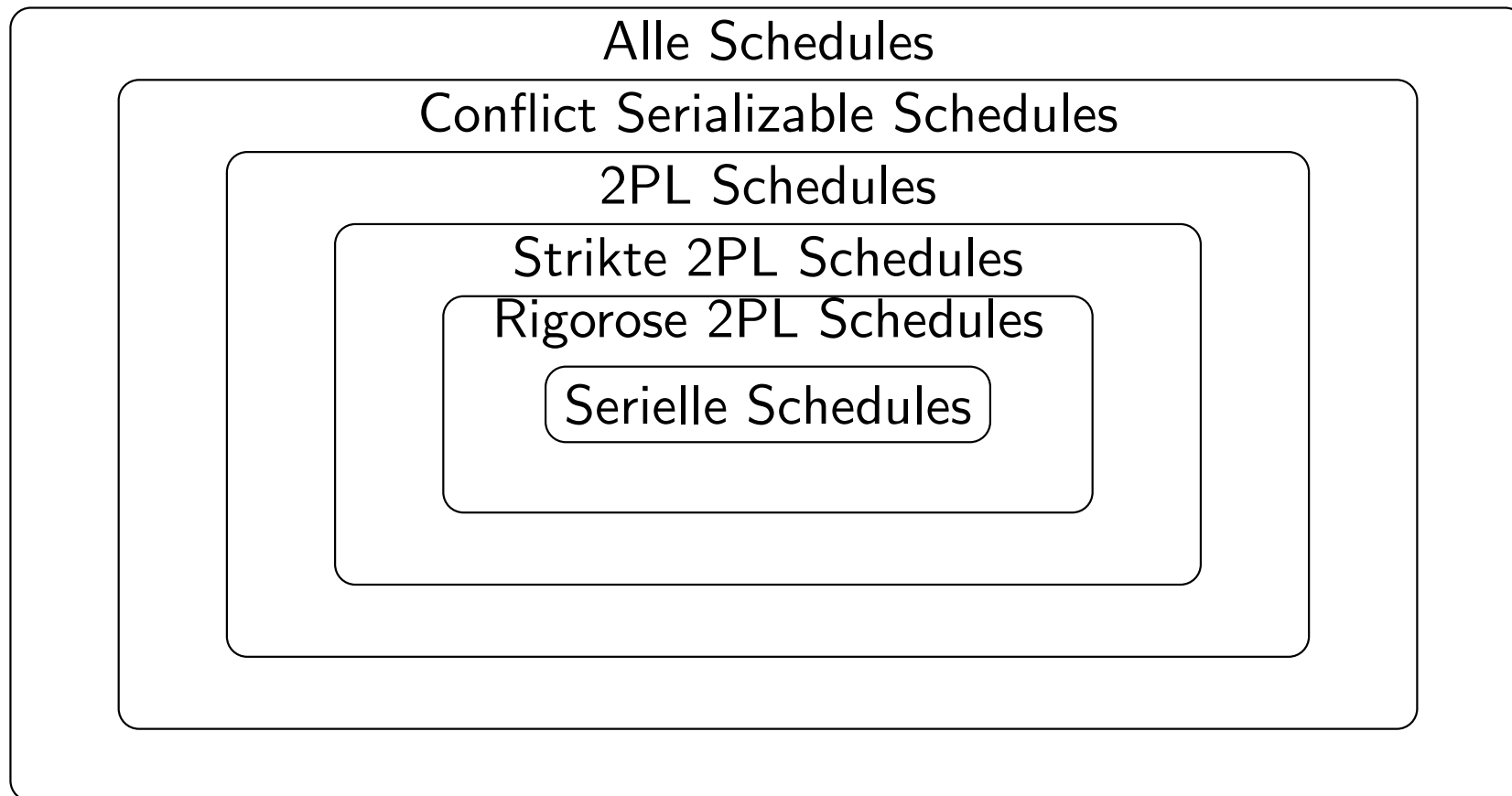
Rigoros

schedule  $S_3$   
 $T_3$

lock\_S(A)  
 lock\_S(B)  
 lock\_X(B)  
 unlock(B)  
 lock\_S(C)  
 unlock(A)  
 commit

keine 2 Phasen

# Übersicht über 2PL Schedules



# Übersicht

- 3 Mehrbenutzersynchronisation
  - Sperrbasierte Synchronisation
  - Zwei-Phasen-Sperrprotokoll (2PL)
  - Lock Konvertierung
  - Erkennung von Deadlocks
  - Vermeidung von Deadlocks

# Deadlocks

2PL kann Deadlocks nicht verhindern

| $T_1$               | $T_2$                       |  |
|---------------------|-----------------------------|--|
| <b>lock_X(A)</b>    | <b>lock_S(B)</b><br>read(B) |  |
| read(A)<br>write(A) |                             |  |
| <b>lock_X(B)</b>    | <b>lock_S(A)</b>            | $T_1$ muss warten auf $T_2$<br>$T_2$ muss warten auf $T_1$<br>$\Rightarrow$ Deadlock |
| ...                 | ...                         |  |

## Lösungen

- **Erkennung von Deadlocks and Recovery**
- **Prävention**
- Timeout

# Erkennung von Deadlocks

Erstelle eines Wartegraphs ( “Wait-for graph” ) und Prüfung auf Zyklen

- Ein Knoten für jede aktive Transaktion  $T_i$
- Kante  $T_i \rightarrow T_j$  wenn  $T_i$  auf die Lock-Freigabe von  $T_j$  wartet

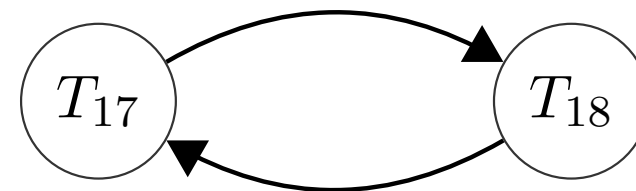
Ein Deadlock existiert, wenn der Wartegraph einen Zyklus hat

# Erkennung von Deadlocks

Wenn ein Deadlock erkannt wird

- Einen passenden Victim auswählen
- Abbruch des Victim und Freigabe aller zugehörigen Locks

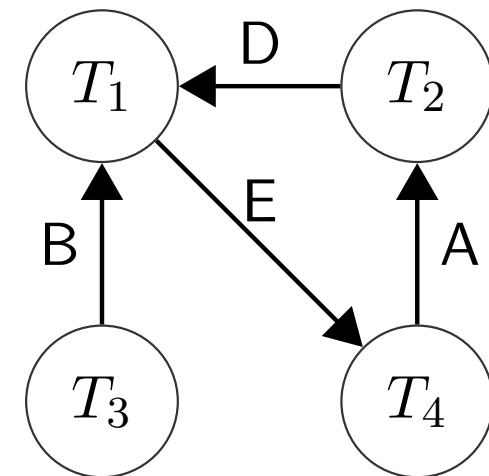
| schedule $S_8$   |  |
|--|--|
| $T_{17}$   | $T_{18}$   |
| <b>lock_X(B)</b><br>read(B, b)<br>$b \leftarrow b - 50$<br>write(B, b) | <b>lock_S(A)</b><br>read(A, a)<br><b>lock_S(B)</b> |
| <b>lock_X(A)</b>   |  |



# Erkennung von Deadlocks - Ein Beispiel

schedule  $S_A$

| $T_1$                                | $T_2$            | $T_3$            | $T_4$                                |
|--------------------------------------|------------------|------------------|--------------------------------------|
|                                      | <b>lock_X(A)</b> |                  |                                      |
| <b>lock_X(B)</b><br><b>lock_X(C)</b> |                  | <b>lock_X(B)</b> |                                      |
| <b>lock_X(D)</b>                     |                  |                  | <b>lock_X(E)</b><br><b>lock_X(A)</b> |
| <b>lock_X(E)</b>                     | <b>lock_X(D)</b> |                  |                                      |



Zyklus zwischen  $T_1$ ,  $T_4$  und  $T_2$   
 $\Rightarrow$  Deadlock erkannt

Rollback von einer oder mehreren involvierten Transaktionen um den Deadlock zu lösen



# Rollback Kandidaten

Wahl einer guten Victim-Transaktion

Rollback von einer oder mehreren Transaktionen die am Zyklus beteiligt sind

- Die letzte (Minimierung des Rollback Aufwands)
- Diejenige, welche die meisten Locks hält (Maximierung der freigegebenen Ressourcen)

Aufpassen, dass nicht immer derselbe Victim gewählt wird (Starvation)

- “Rollback Counter”  
→ Ab einem gewissen Grenzwert: Keine Rollbacks mehr um Deadlocks aufzulösen

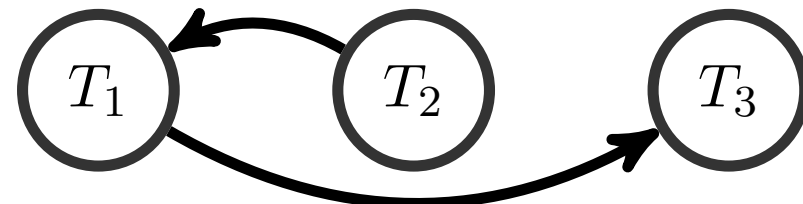
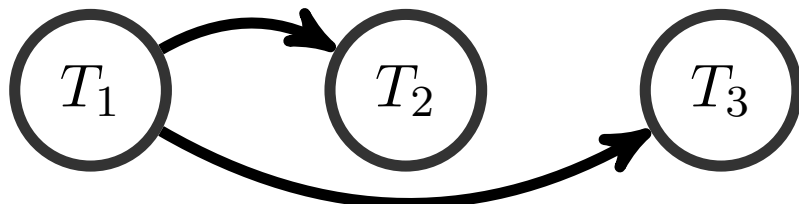
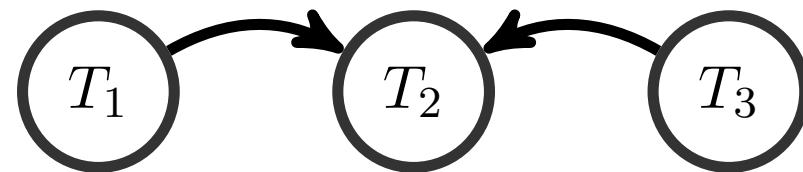
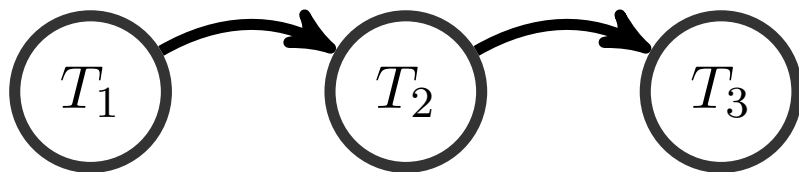
# Menti

## Frage 13

# Wartegraph

Welcher Wartegraph passt zu diesem Schedule?

| schedule $S_A$                             |                     |   |
|--|---------------------|---|
| $T_1$                                      | $T_2$               | $T_3$   |
| $\text{lock\_X(A)}$<br>$\text{lock\_X(B)}$ | $\text{lock\_X(A)}$ | $\text{lock\_X(B)}$<br>$\text{lock\_X(C)}$<br><br>$\text{lock\_X(D)}$ |



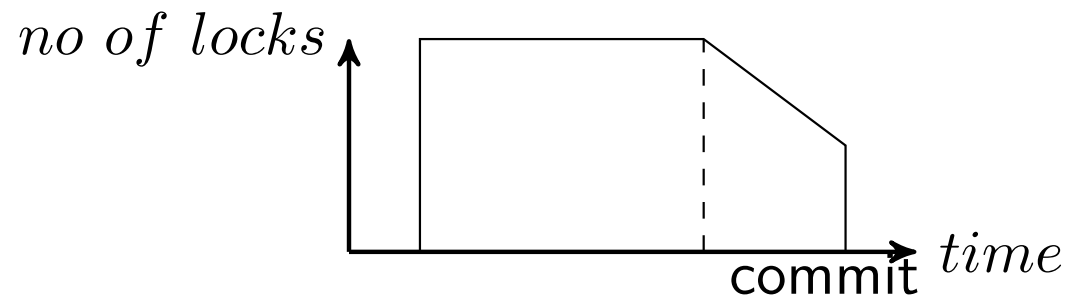
# Übersicht

- 3 Mehrbenutzersynchronisation
  - Sperrbasierte Synchronisation
  - Zwei-Phasen-Sperrprotokoll (2PL)
  - Lock Konvertierung
  - Erkennung von Deadlocks
  - Vermeidung von Deadlocks

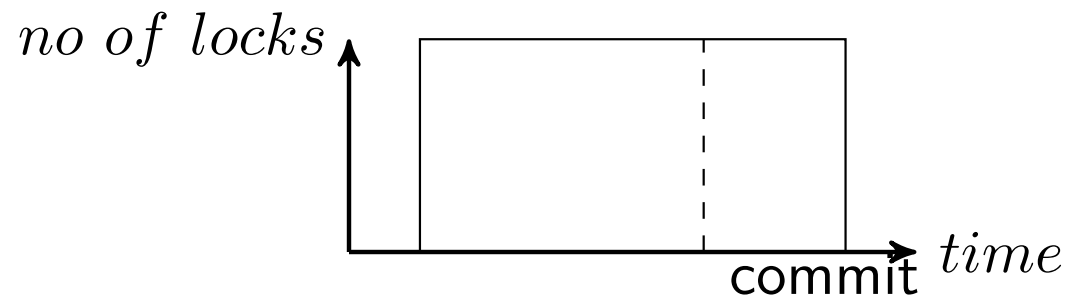
# Konservatives 2PL Protokoll

- Normales/Striktes/Rigoroses 2PL können Deadlocks nicht verhindern
- Zusätzliche Anforderung:  
Alle Locks (Shared und Exclusive) werden gleich zu Beginn einer Transaktion gesetzt

konservatives striktes 2PL



konservatives rigoroses 2PL



Nur in wenigen Applikationen verwendbar

# Zusammenfassung: Mehrbenutzersynchronisation

- Mehrere Protokolle zur Mehrbenutzersynchronisation wurden entwickelt
  - Hauptziel: nur serialisierbare, recoverable und cascadeless Schedules zulassen
  - Zwei-Phasen-Sperrprotokoll
    - Die meisten relationalen DBMS benutzen rigoroses 2PL
- Erkennung von Deadlocks (Wartegraph) und Verhindern von Deadlocks (konservatives 2PL)
- Serializability vs. Nebenläufigkeit

# Lernziele

## Lernziele: Recovery

- Grundlegende Logging Algorithmen verstehen
- Wichtigkeit von Atomicity und Durability verstehen

## Motivation

- Dem Benutzer zu vermitteln eine Transaktion wäre erfolgreich, ohne zu garantieren, dass der Effekt permanent ist, kann schnell teuer werden
- Wir möchten Konsistenz und Verfügbarkeit erhalten auch bei Abstürzen

# Übersicht

- 4 Recovery
  - Fehlerklassifikation
  - Datenspeicher
  - Log-Einträge
  - Log-Based Recovery



# Recovery

“Probleme” bei Transaktionen

- Atomicity
  - Transaktionen können ein abort ausführen (rollback)
- Durability
  - Was wenn das DBMS abstürzt?

Das DBMS garantiert, dass eine Transaktion

- entweder fertig wird und ein permanentes Resultat liefert (committed)
- oder keinen Effekt auf die Datenbank hat (aborted)

Die Rolle der **Recovery** Komponente ist, die Atomicity und Durability von Transaktionen bei Systemfehlern sicherzustellen.

## Wie kann Durability garantiert werden?

- Eine Transaktion verändert Daten im Arbeitsspeicher
- Die Daten wurden **noch nicht** auf die Festplatte geschrieben
- Commit wird durchgeführt

Der Benutzer nimmt an die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- Was passiert bei einem Blackout?
- Welche Daten sind in der Datenbank?

## Wie kann Durability garantiert werden?

- Eine Transaktion verändert Daten im Arbeitsspeicher
- Die Daten wurden **teilweise** auf die Festplatte geschrieben
- Commit wird durchgeführt

Der Benutzer nimmt an die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- Was passiert bei einem Blackout?
- Welche Daten sind in der Datenbank?

## Wie kann Durability garantiert werden?

- Eine Transaktion verändert Daten im Arbeitsspeicher
- Die Daten wurden **vollständig** auf die Festplatte geschrieben
- Commit wird durchgeführt

Der Benutzer nimmt an die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- Was passiert bei einem Hardware Fehler?  
⇒ Verlust einer Festplatte
- Welche Daten sind in der Datenbank?

## Wie kann Durability garantiert werden?

- Eine Transaktion verändert Daten im Arbeitsspeicher
- Die Daten wurden **vollständig** auf **mehrere** Festplatten geschrieben
- Commit wird durchgeführt

Der Benutzer nimmt an die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- Was passiert wenn es Feuer, Flut, Erdbeben, oder sonst etwas gibt?  
⇒ alle Festplatten sind verloren
- Welche Daten sind in der Datenbank?

## Wie kann Durability garantiert werden?

- Eine Transaktion verändert Daten im Arbeitsspeicher
- Die Daten wurden **vollständig** auf **mehrere** Festplatten geschrieben und die Festplatten wurden auf **mehreren geographische verschiedenen** Computerzentren verteilt
- Commit wird durchgeführt

Der Benutzer nimmt an die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- Was passiert wenn es Feuer, Flut, Erdbeben, oder sonst etwas bei allen Computerzentren gibt?  
⇒ alle Computerzentren sind verloren
- Welche Daten sind in der Datenbank?

# Durability

- Durability ist **relativ** und abhängig von der Anzahl an Kopien und den geographischen Orten.
- Garantien sind nur möglich wenn
  - wir zuerst die Kopien aktualisieren und
  - erst dann den Benutzer informieren, dass der Commit der Transaktion erfolgreich waren

Wir nehmen deshalb an, dass die WAL (Write Ahead Logging) Regel erfüllt ist.

Variationen der WAL Regel:

- **Log-Based Recovery**
- Volle Redundanz: Spiegeln aller Daten auf mehreren Computern (Festplatten, Rechenzentren) die alle das gleiche machen

# Fehlerklassifikation

Transaktionsfehler (Fehler einer Transaktion, die noch nicht committed wurde)

- Rückgängigmachen der Änderungen

Systemabsturz (Fehler mit Hauptspeicherverlust)

- Änderungen von Transaktionen die committed wurden, müssen erhalten bleiben
- Änderungen von Transaktionen die nicht committed wurden, müssen rückgängig gemacht werden

Festplattenfehler

- Recovery basierend auf Archiven oder Datenbank Dumps



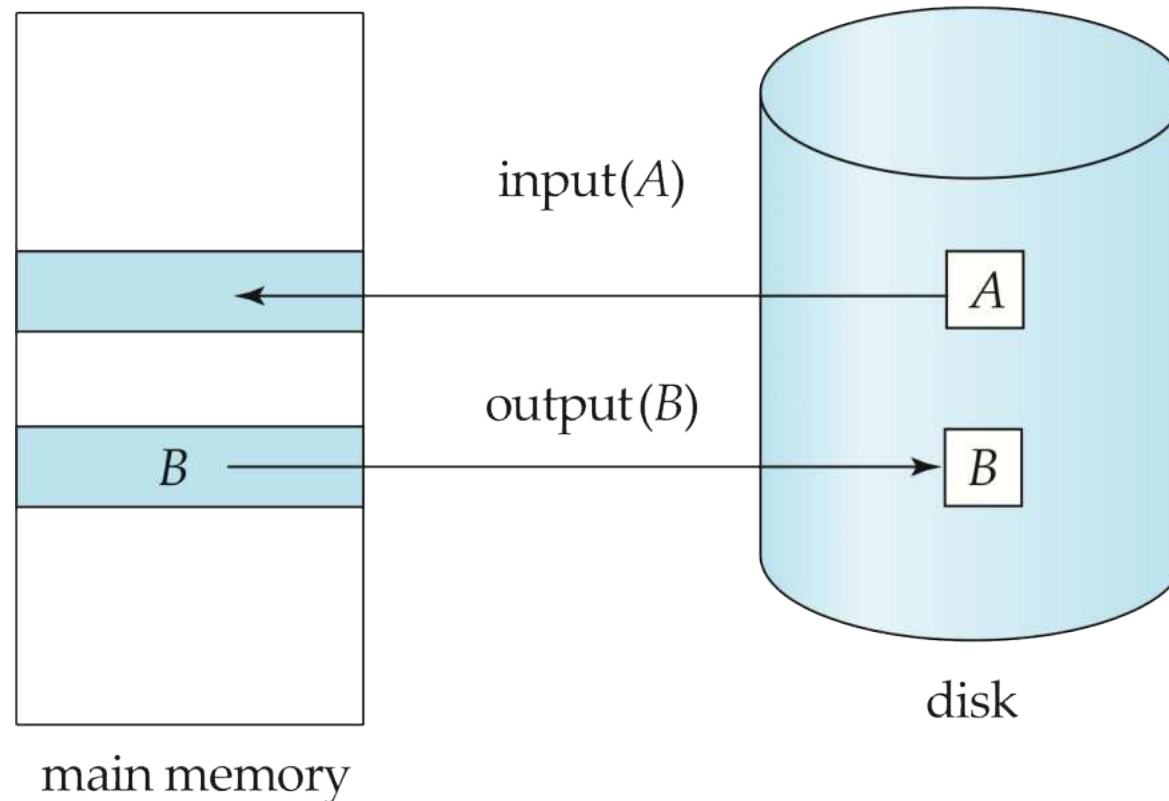
# Übersicht

## 4 Recovery

- Fehlerklassifikation
- **Datenspeicher**
- Log-Einträge
- Log-Based Recovery

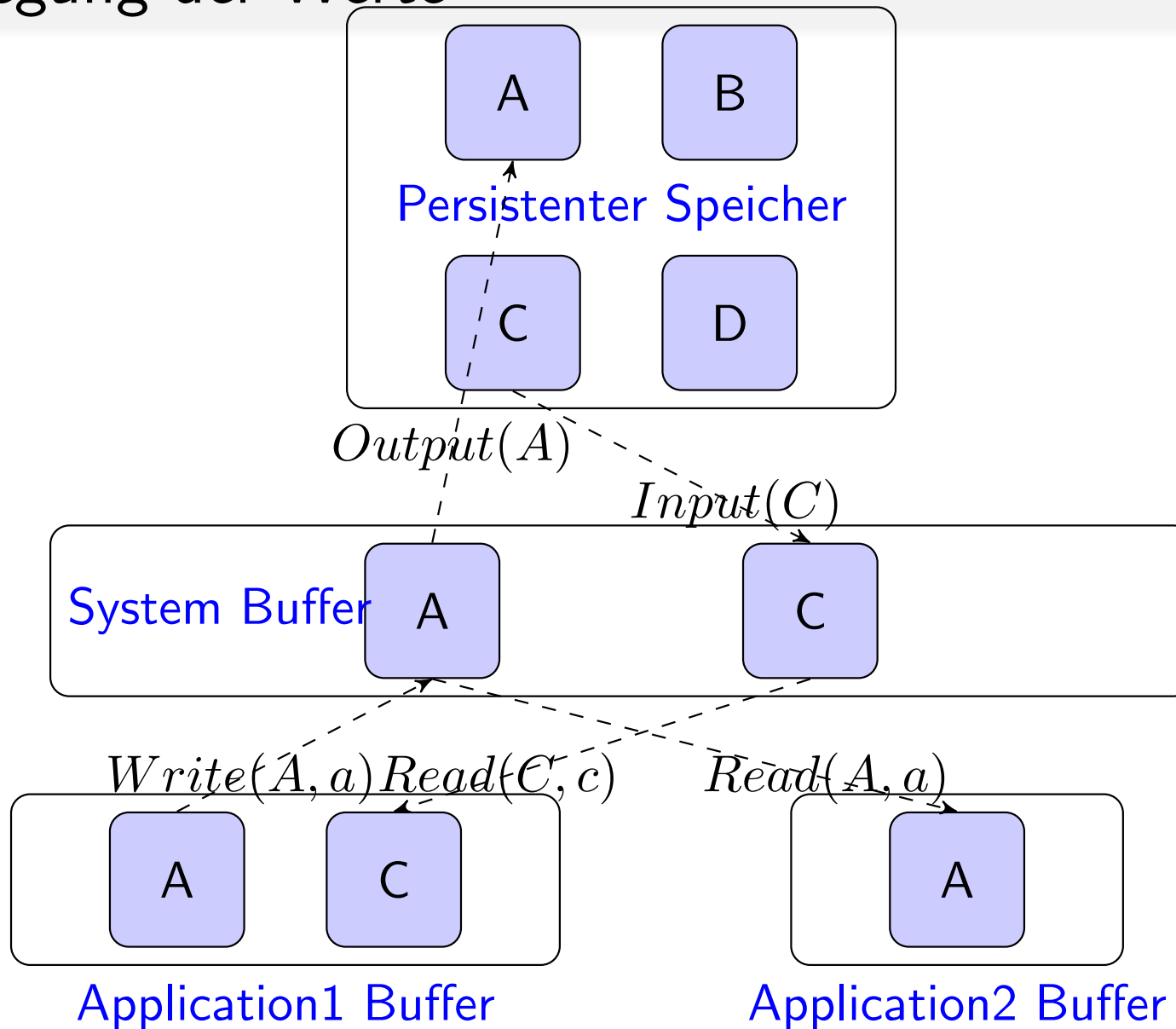
## Zweistufige Speicherhierarchie

Daten werden in Seiten (Pages) und Blöcken (Blocks) organisiert



- **Flüchtiger Speicher (Volatile Storage)** (Arbeitsspeicher)
- **Nichtflüchtiger Speicher (Non-Volatile Storage)** (Fesplatte)
- Stabiler Speicher (Stable storage) (RAIDS, Remote Backups,...)

# Bewegung der Werte



# Speicher Operationen

Transaktionen greifen auf die Datenbank zu und verändern Werte

- Operationen um Blöcke mit Datenobjekten zwischen der Festplatte und dem Arbeitsspeicher (System Buffer) zu bewegen
  - **Input(Q)**  
Transferieren des Blocks, welcher Datenobjekt Q beinhaltet, in den Arbeitsspeicher
  - **Output(Q)**  
Transferieren des Blocks, welcher Q beinhaltet, auf die Festplatte & ersetzen
- Operationen um Werte zwischen Datenobjekten und lokalen Variablen zu verschieben
  - **read(Q,q)**  
weist den Wert des Datenobjekts Q der Variable q zu
  - **write(Q,q)**  
weist den Werte der Variable q dem Datenobjekt Q zu

# Übersicht

## 4 Recovery

- Fehlerklassifikation
- Datenspeicher
- Log-Einträge
- Log-Based Recovery

# Die WAL Regel für Log-Based Recovery

## WAL (Write Ahead Logging)

- Bevor eine Transaktion in den **commit** Zustand wechselt, müssen “alle zugehörigen” Log-Einträge auf einen stabilen Speicher geschrieben worden sein, inkl. dem commit Log-Eintrag
- Bevor eine modifizierte Seite (oder Block) im Arbeitsspeicher in die Datenbank (Nichtflüchtiger Speicher) geschrieben werden kann, müssen “alle zugehörigen” Log-Einträge auf einen stabilen Speicher geschrieben worden sein

# Logging

Während einer normalen Operation

- Am Beginn registriert eine Transaction  $T$  sich selbst im **Log**:  $[T \text{ start}]$
- Wenn ein Datenobjekt  $X$  bearbeitet wird durch  $\text{write}(X, x)$

① Folgender Log-Eintrag wird hinzugefügt

- $[T, X, V\text{-alt}, V\text{-neu}]$
- Transaktions-ID (d.h.,  $T$ )
- Name des Datenobjekts (d.h.,  $X$ )
- Alter Wert des Objekts
- Neuer Wert des Objekts

② Schreibe den neuen Wert von  $X$

*Der Buffer Manager schreibt die neuen Werte später asynchron auf die Festplatte*

- Während der Fertigstellung, fügt Transaction  $T$  den Eintrag  $[T \text{ commit}]$  ins Log ein, danach führt  $T$  den commit aus

*Die Transaktion führt den commit genau dann aus, wenn der commit Eintrag (nach allen vorherigen Einträgen) ins Log geschrieben wird!*

## Struktur eines Log-Eintrags (Log Record)

[TID, DID, old, new]

**TID** ID der Transaktion welche die Änderung verursacht hat

**DID** ID des Datenobjekts

Ort auf der Festplatte (Page, Block, Offset)

**old** Wert des Datenobjekts vor der Änderung

**new** Wert des Datenobjekts nach der Änderung

### Zusätzliche Einträge

**start** Transaktion TID wurde gestartet [TID start]

**commit** Transaktion TID wurde committed [TID commit]

**abort** Transaktion TID wurde aborted [TID abort]



| schedule $S_1$   |  |   |
|--|--|---|
| $T_1$  | $T_2$  | $T_3$   |
| begin<br>read(B, b)<br>$b \leftarrow b+100$<br>write(B, b)<br>commit | begin<br>read(D, d)<br>$d \leftarrow d+470$<br>write(D, d)<br>commit | begin<br>read(D, d)<br>read(E, e)<br>$d \leftarrow d-10$<br>write(D, d)<br>$e \leftarrow e-20$<br>write(E, e)<br>commit |

## Log-Einträge Beispiel

[TID, DID, old, new]

[T1 start]

[T1, B, 300, 400]

[T1 commit]

[T2 start]

[T2, D, 60, 530]

[T2 commit]

[T3 start]

[T3, D, 530, 520]

[T3, E, 70, 50]

[T3 commit]

# Menti

Frage 14–17

# Was stimmt hier nicht?

[T1 commit]

[T1, B, 300, 400]

[T1 start]

Commit vor start

[T1 start]

[T1, B, 300, 400]

[T1 commit]

[T1, C, 40, 540]

Es darf keine Log-Einträge für T1  
nach dem commit geben

# Was stimmt hier nicht?

[T1 start]  
[T1, C, 40, 10]  
[T1 start]  
[T1, B, 300, 400]  
[T1 commit]

[T1 start]  
[T1, C, 40, 10]  
[T1, B, 300, 400]  
[T1 commit]

Korrekt!

Mehrere start Einträge für T1

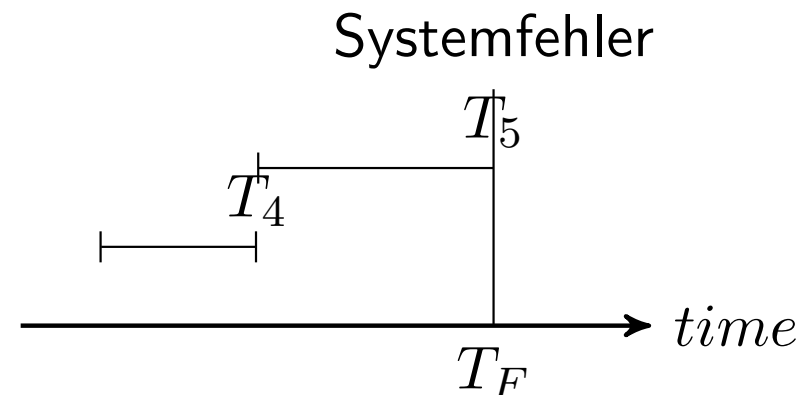
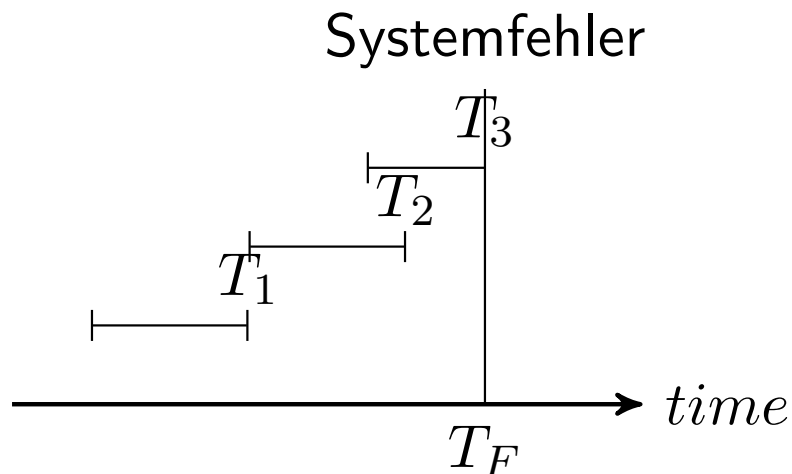
# Übersicht

- 4 Recovery
  - Fehlerklassifikation
  - Datenspeicher
  - Log-Einträge
  - Log-Based Recovery

# Log-Based Recovery

Operationen zur Wiederherstellung nach Fehlern

- **Redo**: Die Änderungen an der Datenbank wiederholen
- **Undo**: Den Datenbankzustand vor der Ausführung wiederherstellen



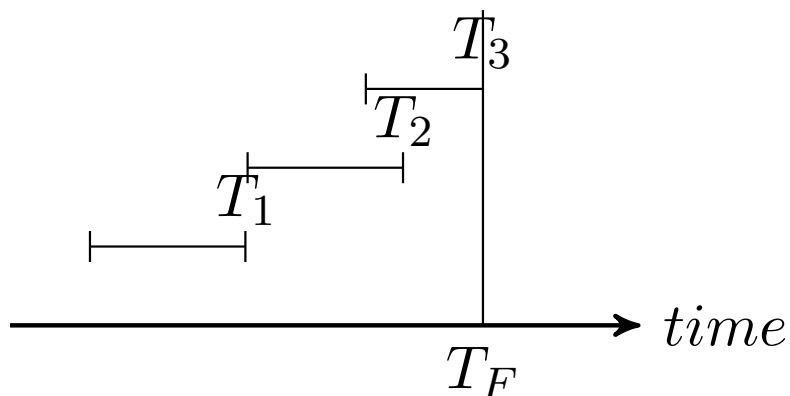
Was macht man bei diesen Transaktionen?

# Log-Based Recovery

Operationen zur Wiederherstellung nach Fehlern

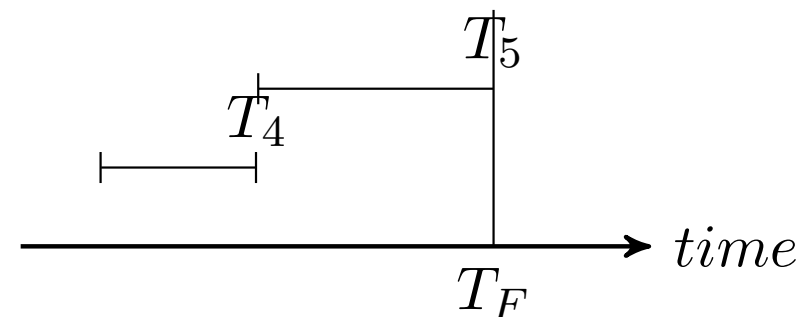
- **Redo**: Die Änderungen an der Datenbank wiederholen
- **Undo**: Den Datenbankzustand vor der Ausführung wiederherstellen

Systemfehler



- Redo  $T_1$  and  $T_2$
- Undo  $T_3$

Systemfehler



- Redo  $T_4$
- Undo  $T_5$

# Recovery Algorithmus

Um die Datenbank nach einem Fehler wiederherzustellen

- Reproduzieren (redo) der Resultate von Transaktionen die committed wurden
- Rückgängigmachen (undo) von Transaktionen die noch nicht committed wurden

Anmerkungen

- In einem Multitasking System, müssen eventuell mehr als eine Transaktion rückgängig gemacht werden.
- Wenn ein Systemabsturz während der Recovery Phase auftritt, muss auch die neue Recovery korrekte Resultate liefern (**Idempotenz**).



# Log-Based Recovery

Datenbank

|   |     |
|---|-----|
| A | 100 |
| B | 300 |
| C | 5   |
| D | 60  |
| E | 80  |

Log-Einträge

[T1 start]  
[T1, B, 300, 400]  
[T1, C, 5, 10]  
[T2 start]  
[T2, E, 80, 480]  
[T1, A, 100, 560]  
[T1 commit]  
[T2, A, 560, 570]  
[T2, D, 60, 530]

Wie würden Sie das Log verwenden (systematisch), um die Datenbank nach einem Absturz wiederherzustellen?

# Drei Recovery Phasen

- ① Redo (Vollständige Wiederholung der Historie)
  - Das ganze Log Schritt für Schritt durchgehen
  - **Alle** Änderungen in derselben Reihenfolge wie im Log anwenden
  - Bestimmen der “Undo” Transaktionen
    - $[T_i \text{ start}]$ :  $T_i$  zur “Undo List” hinzufügen
    - $[T_i \text{ abort}]$  oder  $[T_i \text{ commit}]$ :  $T_i$  von der “Undo List” entfernen
- ② Undo (Rollback) aller Transaktionen in der “Undo List”
  - Das ganze Log rückwärts durchgehen
  - Rückgängigmachen aller Änderungen von Transaktionen in der “Undo List” – Erstellung eines Compensation Log-Eintrags
  - Für jeden  $[T_i \text{ start}]$  Eintrag einer Transaktion  $T_i$  in der “Undo List”, füge einen  $[T_i \text{ abort}]$  Eintrag ins Log ein, entferne  $T_i$  von der “Undo List”
  - Stopp, sobald die “Undo List” leer ist

# Compensation Log-Einträge

[TID, DID, value]

- Erstellt zum Rückgängigmachen (ausgleichen/compensate) der Änderungen von [TID, DID, value, newValue]
- Redo-Only Log-Eintrag
- Kann auch für einen Rollback einer Transaktion während der normalen Ausführung verwendet werden

# Beispiel

## Phase 1 (Redo)

Datenbank

|   |     |
|---|-----|
| A | 100 |
| B | 300 |
| C | 5   |
| D | 60  |
| E | 80  |

Undo List

{ T1 }

Log-Einträge

[T1 start]

[T1, B, 300, 400]

[T1, C, 5, 10]

[T2 start]

[T2, E, 80, 480]

[T1, A, 100, 560]

[T1 commit]

[T2, A, 560, 570]

[T2, D, 60, 530]

# Beispiel

## Phase 1 (Redo)

Datenbank

|   |     |
|---|-----|
| A | 100 |
| B | 400 |
| C | 5   |
| D | 60  |
| E | 80  |

Undo List

{ T1 }

Log-Einträge

[T1 start]

[T1, B, 300, 400]

[T1, C, 5, 10]

[T2 start]

[T2, E, 80, 480]

[T1, A, 100, 560]

[T1 commit]

[T2, A, 560, 570]

[T2, D, 60, 530]

# Beispiel

## Phase 1 (Redo)

Datenbank

|   |     |
|---|-----|
| A | 100 |
| B | 400 |
| C | 10  |
| D | 60  |
| E | 80  |

Undo List

{ T1, T2 }

Log-Einträge

[T1 start]

[T1, B, 300, 400]

[T1, C, 5, 10]

[T2 start]

[T2, E, 80, 480]

[T1, A, 100, 560]

[T1 commit]

[T2, A, 560, 570]

[T2, D, 60, 530]

# Beispiel

## Phase 1 (Redo)

### Datenbank

|   |     |
|---|-----|
| A | 100 |
| B | 400 |
| C | 10  |
| D | 60  |
| E | 480 |

### Undo List

{ T1, T2 }

### Log-Einträge

[T1 start]  
[T1, B, 300, 400]  
[T1, C, 5, 10]  
[T2 start]  
[T2, E, 80, 480]  
[T1, A, 100, 560]  
[T1 commit]  
[T2, A, 560, 570]  
[T2, D, 60, 530]

# Beispiel

## Phase 1 (Redo)

Datenbank

|   |     |
|---|-----|
| A | 570 |
| B | 400 |
| C | 10  |
| D | 60  |
| E | 480 |

Undo List

{ T2 }

Log-Einträge

[T1 start]  
[T1, B, 300, 400]  
[T1, C, 5, 10]  
[T2 start]  
[T2, E, 80, 480]  
[T1, A, 100, 560]  
[T1 commit]  
[T2, A, 560, 570]  
[T2, D, 60, 530]



# Beispiel

## Phase 1 (Redo)

Datenbank

|   |     |
|---|-----|
| A | 570 |
| B | 400 |
| C | 10  |
| D | 530 |
| E | 480 |

Undo List  
{ T2 }

## Log-Einträge

[T1 start]  
[T1, B, 300, 400]  
[T1, C, 5, 10]  
[T2 start]  
[T2, E, 80, 480]  
[T1, A, 100, 560]  
[T1 commit]  
[T2, A, 560, 570]  
[T2, D, 60, 530]

# Beispiel

## Phase 2 (Undo)

Datenbank

|   |     |
|---|-----|
| A | 560 |
| B | 400 |
| C | 10  |
| D | 60  |
| E | 80  |

Undo List

{ }

Log-Einträge

[T1 start]  
[T1, B, 300, 400]  
[T1, C, 5, 10]  
[T2 start]  
[T2, E, 80, 480]  
[T1, A, 100, 560]  
[T1 commit]  
[T2, A, 560, 570]  
[T2, D, 60, 530]  
[T2, D, 60]  
[T2, A, 560]  
[T2, E, 80]  
[T2 abort]

# ARIES

- State-Of-The-Art Methode
- Erweitert den vorgestellten Algorithmus um einige Optimierungen und einer Vorab-Phase: Durchgehen des Logs um Dirty Pages zu identifizieren und um den "Startpunkt" des Logs sowie die "Undo" Transaktionen zu bestimmen
- Behandlung von Dirty Pages:  
DirtyPageTable, Erweiterung der Log-Einträge (pageID), Erweiterung der Pages (pageLSN, letzter Log-Eintrag der Änderungen vorgenommen hat),...

Log-Einträge in ARIES haben eine Log Sequence Number (LSN):  
[LSN, TransactionID, PageID, redoValue, undoValue, prevLSN]

Compensation Log-Eintrag in ARIES (CLR):  
[LSN, TransactionID, PageID, redoValue, prevLSN, undoNxtLSN]

# Zusammenfassung: Recovery

- Ziel: Sicherstellen von Atomicity und Durability trotz Systemfehlern und Abstürzen
- Durability ist relativ
- WAL Regel
- Log-Based Recovery
  - Alle Änderungen müssen in eine Log Datei geschrieben werden
  - Eine Transaktion führt den Commit genau dann durch, wenn der Commit Eintrag ins Log geschrieben wurde