

5. Transaktionen

Einführung

Beispiel einer typischen Bankanwendung:

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

Jetzt ist natürlich die Frage, was hier schiefgehen kann.

Die Antwort dafür ist:

- Es könnte irgendwie zwischen Lesen und Schreiben was passieren.
- Alle Schritte müssen deshalb als Einheit betrachtet werden nach dem "Alles oder nichts"-Prinzip
- Sobald abgeschlossen, müssen die Änderungen *permanent gespeichert* werden.

Transaktion

Was ist eine Transaktion?

- **Definition:** Eine Transaktion fasst mehrere Datenbankoperationen zu einer einzigen logischen Einheit zusammen.
- **Aktionen innerhalb einer Transaktion:**
 - Zugriff auf verschiedene Datenbankeinträge.
 - Möglicherweise Veränderung einiger Einträge.
- **Definition der Transaktionsgrenzen:** Erfolgt durch den Benutzer oder die Softwareanwendung.

Eigenschaften von Transaktionen: ACID

Die ACID-Eigenschaften sind grundlegend für zuverlässige Transaktionsverarbeitung in Datenbanken.

Atomicity (Atomarität)

- **Prinzip:** Eine Transaktion wird als unteilbare Einheit behandelt.
- **Auswirkung:** Entweder werden *alle* Operationen innerhalb der Transaktion erfolgreich in der Datenbank durchgeführt (Commit), oder *keine* von ihnen wird übernommen (Rollback).
- **Implementierung:** Typischerweise durch die Verwendung von Log-Einträgen (Protokolldateien), die Änderungen vor der eigentlichen Durchführung festhalten und im Fehlerfall ein Zurücksetzen ermöglichen.

Consistency (Konsistenz)

- **Prinzip:** Die Ausführung einer Transaktion in Isolation (ohne gleichzeitige andere Transaktionen) bewahrt den konsistenten Zustand der Datenbank.
- **Gewährleistung:**
 - Einhaltung definierter Constraints (z.B. Datentypen, Wertebereiche).
 - Durchführung von Checks (Überprüfungen von Bedingungen).
 - Einhaltung von Assertions (Zusicherungen über den Datenbankzustand).
- **Anwendungsdefinition:** Die Semantik der Konsistenz wird auch durch die Anwendung selbst definiert.
 - **Beispiel:** Bei einer Banküberweisung muss die Summe des abgebenden und empfangenden Kontos gleich bleiben – es darf kein Geld "entstehen" oder "verschwinden". Die Gesamtsumme aller beteiligten Konten muss vor und nach der Transaktion identisch sein.

Isolation

- **Prinzip:** Jede Transaktion operiert auf der Datenbank so, als ob sie die einzige aktive Transaktion wäre.
- **Auswirkung:** Zwischenergebnisse einer Transaktion dürfen für andere, gleichzeitig laufende Transaktionen nicht sichtbar sein. Dies verhindert unerwünschte Beeinflussungen und Inkonsistenzen.
- **Implementierung:** Typischerweise durch den Einsatz von Locks (Sperren), die den Zugriff auf betroffene Datenobjekte für andere Transaktionen einschränken, bis die aktuelle Transaktion abgeschlossen ist.

Durability (Dauerhaftigkeit)

- **Prinzip:** Änderungen, die von einer erfolgreich abgeschlossenen Transaktion (Commit) an der Datenbank vorgenommen wurden, müssen dauerhaft gespeichert bleiben und dürfen auch bei Systemfehlern (z.B. Stromausfall, Festplattenausfall) nicht verloren gehen.

- **Gewährleistung:**

- Typischerweise durch die Verwendung von Log-Einträgen (Protokolldateien), in denen die durchgeführten Änderungen persistent protokolliert werden.
 - Im Fehlerfall können die Änderungen anhand des Logs wiederhergestellt werden (Recovery-Mechanismen).
-

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;

"autocommit" Modus

Jeder Befehl wird in einer eigenen Transaktion ausgeführt.

Einfache Konsistenzprüfung (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);
```

Da sind einfache Konsistenzchecks die man bei Erstellung einer Tabelle definieren kann

- Viele der Fehler können **von DBMS erkannt** werden
- Verwende diese Checks!

Savepoints

Transaktionen von langer Dauer können zusätzlich Savepoints definieren

SAVEPOINT savepoint_name;

Definiert einen Punkt/Zustand innerhalb einer Transaktion.

Eine Transaktion kann bis zum Savepoint **teilweise rückgängig gemacht werden.**

ROLLBACK TO savepoint_name;

Setzt die aktive Transaktion zurück bis zum Savepoints savepoint_name

Man kann einfach Zwischenspeicherungen machen.

Ein Beispiel dazu wäre:

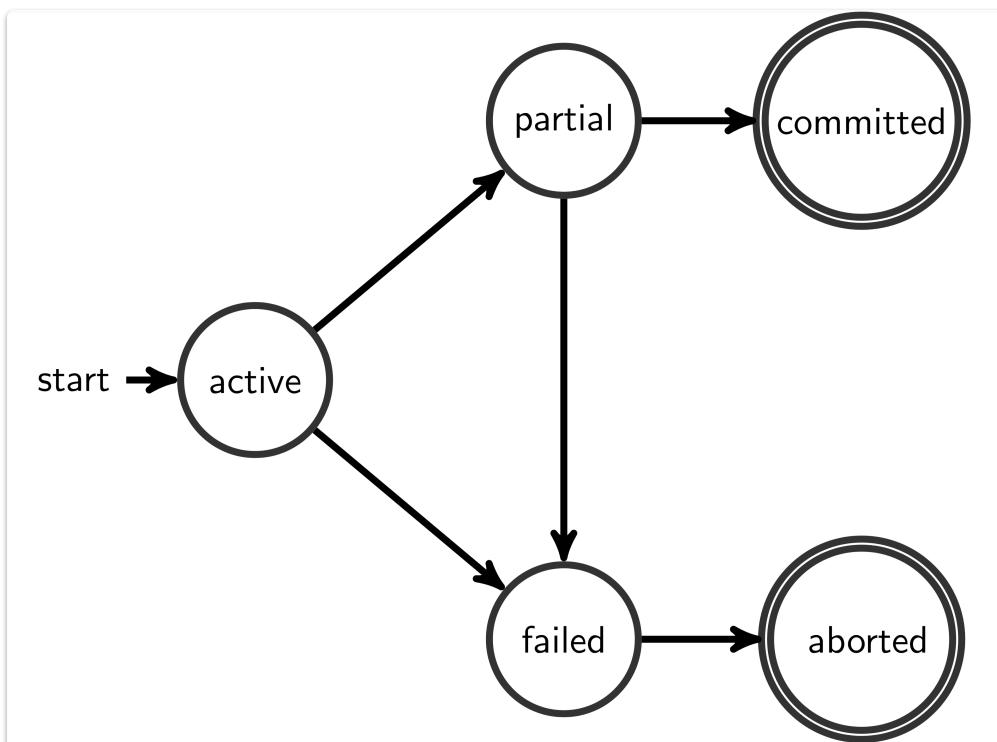
```
BEGIN;
INSERT INTO tab VALUES ...
SAVEPOINT A;
INSERT INTO tab VALUES...
SAVEPOINT B;
SELECT * FROM tab;
ROLLBACK TO A;
SELECT * FROM tab;
```

Beenden 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



Es kann sein, dass wenn man fertig mit seiner Operation ist und committen will, dass man das dann nicht machen kann, **weil eine andere Person in der Zwischenzeit etwas verändert hat**. Dadurch kann allerdings nichts schiefgehen und es wird sicher aborted.

Realisierung vom DBMS

Die beiden wichtigsten Komponenten der Transaktionsverwaltung sind:

Mehrbenutzersynchronisation (Isolation)

- **Ziel:** Semantische Korrektheit bei Nebenläufigkeit.
- **Vorteil:** Hoher Durchsatz.
- **Serialisierbarkeit:** Ergebnis nebenläufiger Ausführung soll serieller Ausführung entsprechen.

- **Isolation Levels:** Schwächere Stufen für mehr Nebenläufigkeit möglich.

Recovery (Atomicity und Durability)

- **Ziel:** Atomarität und Dauerhaftigkeit sicherstellen.
- **Rollback:** Zurücksetzen teilweise ausgeführter Transaktionen.
- **Wiederausführung:** Nach Ausfällen.
- **Persistenz:** Sicherstellen der Dauerhaftigkeit von Änderungen.

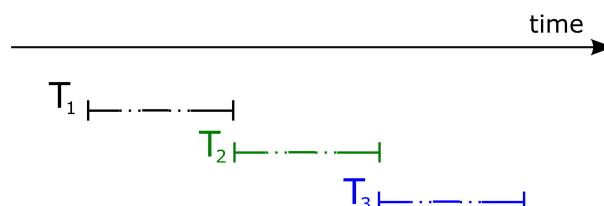
Schedules und Serialisierbarkeit

Nebenläufigkeit (Parallelität)

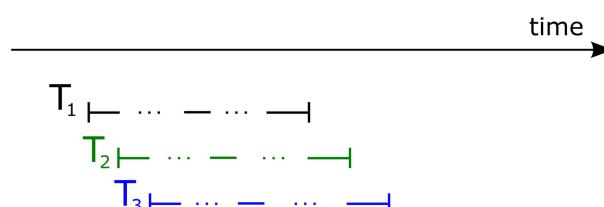
Das „I“ in ACID.

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

1. **Lost Updates** (verlorengegangene Änderung durch Überschreiben)

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

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

Steps	T_1	T_2
1.	<code>read(A, a₁)</code>	
2.	$a_1 := a_1 - 300$	
3.	write(A, a₁)	
4.		<code>read(A, a₂)</code>
5.		$a_2 := a_2 * 1.03$
6.		<code>write(A, a₂)</code>
7.	<code>read(B, b₁)</code>	
8.	...	
9.	abort	

3. Non-Repeatable Read (Abhängigkeit von anderen Updates)

T_1	T_2
<code>update account</code> <code>set balance=42000</code> <code>where accountID=12345</code>	<code>select sum(balance)</code> <code>from account</code> <code>select sum(balance)</code> <code>from account</code>

4. Phantomproblem (Abhängigkeit von neuen/gelöschten Tupeln)

T_1	T_2
<code>insert into account</code> <code>values (C,1000,...)</code>	<code>select sum(balance)</code> <code>from account</code> <code>select sum(balance)</code> <code>from account</code>

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 $Y = 21$

Beispiel von [hier](#) bis [hier](#).

Formale Definition eines Schedules

[DBS-7_transaktionen, p.19](#), [DBS-7_transaktionen, p.19](#)

- **Schedule (Historie):** Sequenz von Operationen mehrerer Transaktionen.
- **Nebenläufige Transaktionen:** Operationen können verzahnt sein.
- **Operationen:**
 - `read(Q, q)` : Liest Datenobjekt Q in lokale Variable q.
 - `write(Q, q)` : Schreibt lokale Variable q in Datenobjekt Q.
 - Arithmetische Operationen
 - `commit` : Beendet Transaktion erfolgreich.
 - `abort` : Bricht Transaktion ab.

Arten von Schedules

- **Serieller Schedule:** Operationen von Transaktionen werden nacheinander ohne Überlappung ausgeführt.
- **Nebenläufiger Schedule:** Operationen von Transaktionen werden zeitlich überlappend ausgeführt.

Gültiger Schedule

- Ein Schedule ist **gültig (valid)**, wenn das Ergebnis der Ausführung "korrekt" ist (Definition der Korrektheit hängt vom Kontext ab).

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)	read(X, x)		read(X, x)	read(X, x)	
$x \leftarrow 2x$			$x \leftarrow 2x$	$x \leftarrow x+1$	
write(X, x)	write(X, x)		write(X, x)	write(X, x)	
read(Y, y)	read(X, x)	read(Y, y)			read(X, x)
$y \leftarrow 2y$		$x \leftarrow x+1$			$x \leftarrow 2x$
write(Y, y)	write(X, x)	write(Y, y)			write(X, x)
read(X, x)			read(Y, y)	read(Y, y)	read(Y, y)
$x \leftarrow x+1$			$y \leftarrow 2y$	$y \leftarrow y+1$	$y \leftarrow 2y$
write(X, x)			write(Y, y)	write(Y, y)	write(Y, y)
read(Y, y)					
$y \leftarrow y+1$					
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 **konsistenten Zustand** hinterlassen.
- Voraussetzung: Jede einzelne Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand, wenn sie isoliert ausgeführt wird.
- Also muss auch in einem konsistenten Zustand gestartet werden

Definition D2 (Ergebnisäquivalenz):

- Eine nebenläufige Ausführung von Transaktionen muss **ergebnisäquivalent** zu einer seriellen Ausführung derselben Transaktionen sein.
- **Ergebnisäquivalent** bedeutet: Die finalen Zustände der Datenbank müssen identisch sein, egal in welcher seriellen Reihenfolge die Transaktionen ausgeführt worden wären.

Beispiel

schedule S_2	
T_3	T_4
read(X, x) write(X, x)	
	read(X, x) $x \leftarrow 2x$ write(X, x) read(Y, y) $y \leftarrow 2y$
read(Y, y) $y \leftarrow y+1$ write(Y, y)	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.

Die bessere Wahl ist Definition D2:

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

Gegeben ist eine Menge von n nebenläufigen Transaktionen. Wie können wir effizient auf Korrektheit prüfen?

Im Folgenden 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.

Konflikt Serialisierbarkeit

Mögliche Konflikte zwischen Transaktionen

Sobald ein write drinnen ist, haben wir das Potenzial zu einem Konflikt

„Konflikte“ zwischen Paaren von Transaktionen (T_1 und T_2) und deren Operationen.

schedule S_A	
T_1	T_2
write(X, x)	
	read(X, x)

Konflikt

schedule S_C	
T_1	T_2
	read(X, x)
write(X, x)	

Konflikt

schedule S_B	
T_1	T_2
write(X, x)	
	write(X, x)

Konflikt

schedule S_D	
T_1	T_2
read(X, x)	
	read(X, x)

Kein Konflikt

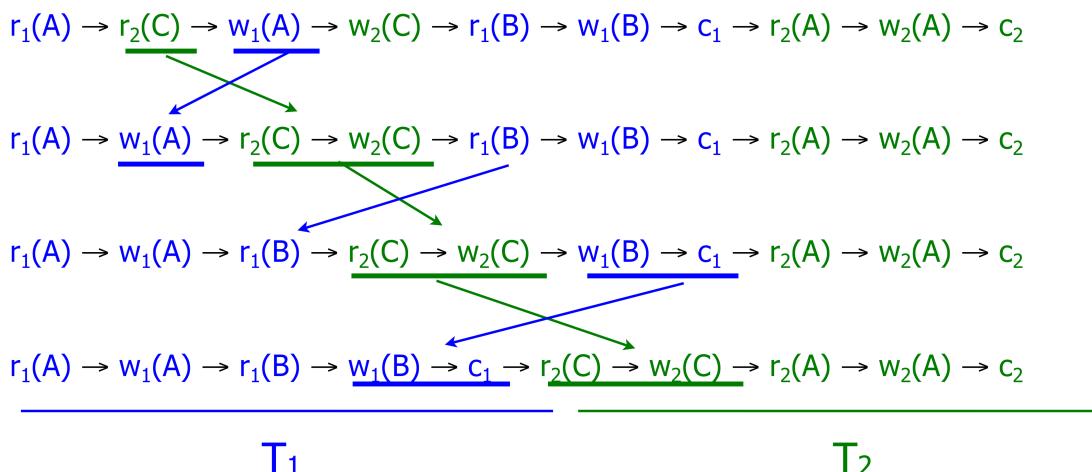
Definition (D4) Conflict Serializability

- Ein Schedule ist **conflict serializable**, wenn er **konfliktäquivalent** zu einem seriellen Schedule ist.
- **Konfliktäquivalente Schedules:**
 - Zwei Schedules S und S' sind konfliktäquivalent, wenn sie die gleichen Operationen in der gleichen Reihenfolge für jedes Datenobjekt haben, mit Ausnahme von Paaren aufeinanderfolgender Operationen, die **keinen Konflikt** aufweisen und deren Reihenfolge vertauscht wurde.
- **Kein Konflikt zwischen Operationen I und J:**
 - Sie gehören nicht zur gleichen Transaktion.
 - Mindestens eine der Operationen ist ein Lesevorgang.
 - Sie greifen nicht auf dasselbe Datenobjekt zu.
- **Konstruktion konfliktäquivalenter Schedules:** Durch wiederholtes Vertauschen nicht-konfliktierender, aufeinanderfolgender Operationen kann ein nebenläufiger Schedule in einen seriellen Schedule überführt werden, wenn er conflict serializable ist.

Wir tauschen die Reihenfolge so lange bis wir eine serialisierbare Schedule haben.

Beispiel Konfliktäquivalenz von 2 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)

Weitere Beispiele:

schedule S_A	
T_1	T_2
read(Y, y)	read(X, x) write(X, x)
write(Y, y)	

Conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	read(X, x) write(X, x)
write(X, x)	

Nicht conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	read(X, x)
write(X, x)	

Conflict serializable

schedule S_D	
T_7	T_8
read(X, x)	write(X, x)
write(X, x)	

Nicht conflict serializable

Conflict Graph (Precedence Graphs)

- Ziel:** Analyse der Konfliktserialisierbarkeit eines Schedules.
- Konstruktion:** Gerichteter Graph wird für einen gegebenen Schedule erstellt.
- Annahme:** Innerhalb einer Transaktion erfolgt ein `read` auf ein Datenobjekt immer vor einem `write` auf dasselbe Objekt.

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

- Die Knoten des Conflict Graphs 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 annotiert.

Beispiel für einen Conflict Graph des Schedules S_1



Serialisierbarkeit feststellen

Gegeben:

- Ein Schedule S
- Ein Conflict Graph

Wie feststellen, ob S conflict serializable ist?

Ein Schedule S ist **conflict serializable**, wenn der Conflict Graph azyklisch ist.

Intuition:

- Ein **Konflikt** zwischen zwei Transaktionen erzwingt eine bestimmte Ausführungsreihenfolge.
- Die Conflict Serializability entspricht der Existenz einer **topologischen Sortierung** des Conflict Graphen.

Warum Conflict Serializability?

Wir verwenden Conflict Serializability (und keine andere Definition der Serialisierbarkeit), **weil es eine praktikable Implementierung gibt.**

Beispiel für Conflict Graph

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

```

graph TD
    T10((T10)) --> T11((T11))
    T10((T10)) --> T12((T12))
    T10((T10)) --> T13((T13))
    T11((T11)) --> T13((T13))
    T13((T13)) --> T14((T14))
  
```

Wir haben 5 Knoten weil wir 5 einzelne Transaktionen haben.

Kanten werden erstellt, je nach dem was von was Abhängig ist. Also dadurch dass Y in T10 gelesen und in T11 und T13 beschrieben wird, zeigt T10 auf T11 und T12. T10 zeigt außerdem noch auf T13 wegen dem Z.

So macht man das dan für alle anderen Kanten auch.

Einfach die Operationen der Transaktionen durchgehen, schauen wo die Konflikte sind und dann die Kanten einzeichnen.

Was sagt uns der Graph jetzt?

- er hat keinen Zyklus
- --> Er ist Konflikt serialisierbar

verschiedene Ergebnisse

es gibt jetzt verschiedene Reihenfolgen die funktionieren:

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

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

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

Beziehungen zwischen Schedules

Alle Schedules

Schedules die DB in einem konsistenten Zustand belassen (D1)

Schedules äquivalent zu einem seriellen Schedule (D2)

Sichtenserialisierbare Schedules (D3)

Conflict Serializable Schedules (D4)

Serielle Schedules

Recoverable Schedules und Cascadeless Schedules

Transaktionen können fehlschlagen

- Abort vom User aus
- Rollback
- Irgendwelche Fehler

Recoverable Schedules

- Wenn T_i fehlschlägt, muss die Transaktion zurückgesetzt werden, um die **Atomicity (Atomarität)** Eigenschaft zu erhalten (siehe Recovery).
- Wenn eine andere Transaktion T_j Daten gelesen hat, die von T_i geschrieben wurden, dann muss auch T_j zurückgesetzt werden.
⇒ DBS muss sicherstellen, dass Schedules recoverable sind.
- Dieser Schedule ist nicht recoverable.

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

Ein Schedule ist **recoverable**, wenn für jedes Transaktionspaar T_i und T_j gilt:

Wenn T_j Daten liest, die von T_i geschrieben wurden, dann muss T_i vor T_j committet werden.

schedule S_A		schedule S_B	
T_i	T_j	T_i	T_j
read(X, x)		read(Y, y)	
write(X, x)		write(Y, y)	
rollback		rollback	
	read(X, x)	read(X, x)	
	write(X, x)	write(X, x)	
	commit	commit	

recoverable
recoverable

Cascading Rollbacks (Kaskadierendes Rücksetzen)

schedule S_{11}		
T_{22}	T_{23}	T_{24}
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
	read(A, a)	
		read(A, a)
		read(B, b)
rollback		

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

Ist recoverable weil nicht committed wurde

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 geschrieben 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)		
read(B, b)		
write(A, a)		
write(B, b)		
rollback		
	read(A, a) commit	
		read(A, a)
		read(B, b)
		commit

Dieser Schedule ist auch recoverable

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

Ein Schedule ist **cascadeless**, wenn für jedes Transaktionspaar T_i und T_j gilt:

Wenn T_j Daten liest, die von T_i geschrieben wurden, dann muss T_i vor dem Lesezugriff von T_j bereits committed sein.

Vorteil:

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

Jeder cascadeless Schedule ist auch recoverable.

Cascading Rollbacks:

- Können schnell zeitaufwändig werden.

Schlussfolgerung:

- Es ist sinnvoll, sich auf Schedules zu beschränken, die cascadeless sind.
-

Zusammenfassung: Transaktionen und Schedules

- **Transaktionen:** Erhalten den konsistenten Zustand der Datenbank.
- **Serielle Ausführung:** Eine serielle Ausführung einer Menge von Transaktionen erhält einen konsistenten Zustand.
- **Nebenläufige Ausführung:** Die einzelnen Ausführungsschritte mehrerer Transaktionen können verzahnt sein.
- **Serialisierbarkeit:** Ein nebenläufiger Schedule ist serialisierbar, wenn er äquivalent zu einem seriellen Schedule ist.

Conflict Serializability

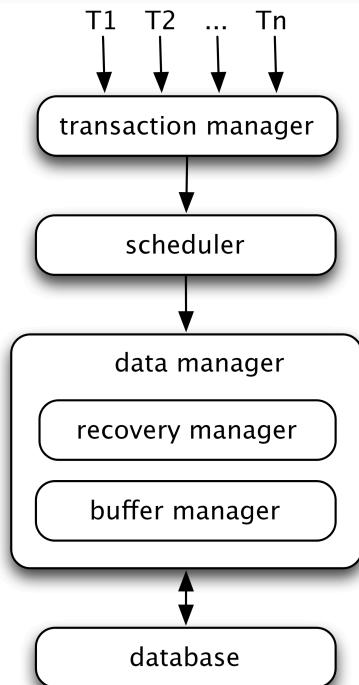
- **Bevorzugte Methode:** Aufgrund praktikabler Implementierungsmöglichkeiten.
- **Conflict Graphs:** Werden zur Bestimmung der Conflict Serializability verwendet. Ein Schedule ist conflict serializable, wenn sein Conflict Graph azyklisch ist.

Recoverability und Cascadeless Schedules

- Schedules müssen **recoverable** sein.
 - Ein Schedule ist recoverable, wenn für jedes Transaktionspaar T_i und T_j , bei dem T_j Daten liest, die von T_i geschrieben wurden, T_i vor T_j committet wird.
 - Schedules sollten **cascadeless** sein.
 - Jeder cascadeless Schedule ist auch recoverable.
 - Ein Schedule ist cascadeless, wenn für jedes Transaktionspaar T_i und T_j , bei dem T_j Daten liest, die von T_i geschrieben wurden, T_i vor dem Lesezugriff von T_j bereits committed ist.
 - Cascading Rollbacks können durch cascadeless Schedules verhindert werden.: Transaktionen und Schedules
-

Datenbank Scheduler

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

- **Ziel:** Sicherstellen von (Konflikt-)serialisierbaren Schedules durch Verzögern von Transaktionen, die die Serialisierbarkeit verletzen würden.
- **Zwei Arten von Sperren auf ein Datenobjekt Q:**
 - S (shared lock, read lock, Lesesperrre)
 - X (exclusive lock, write lock, Schreibsperrre)
- **Sperroperationen:**
 - `lock_S(Q)` - Setzt ein Shared Lock auf Datenobjekt Q. Mehrere Transaktionen können gleichzeitig ein Shared Lock auf demselben Objekt halten.
 - `lock_X(Q)` - Setzt ein Exclusive Lock auf Datenobjekt Q. Nur eine Transaktion kann ein Exclusive Lock auf einem Objekt halten.
 - `unlock(Q)` - Freigabe des Locks auf Datenobjekt Q.

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)
- **OK** bedeutet, dass die Locks kompatibel sind und von verschiedenen Transaktionen gleichzeitig gehalten werden können.
- **-** bedeutet, dass die Locks inkompatibel sind und nicht gleichzeitig von verschiedenen Transaktionen gehalten werden können.
- **Wichtige Regeln:**
 - Nebenläufige Transaktionen dürfen nur kompatible Locks verwenden.
 - Eine Transaktion muss ggf. auf Freigabe eines Locks warten.

Beispiel

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}
lock_X(B) read(B, b) $b \leftarrow b - 50$ write(B, b) unlock(B) lock_X(A) read(A, a) $a \leftarrow a + 50$ write(A, a) unlock(A)	lock_S(A) read(A, a) unlock(A) lock_S(B) read(B, b) unlock(B) display(A+B)

Resultat bei serieller Ausführung:

T_{16} zeigt 300

Bei diesem Schedule:

T_{16} zeigt 250

Ursache des Problems:

Der Exclusive Lock auf B wurde zu früh freigegeben.

Initial: A = 100 und B = 200

Probleme bei zu früher Eingabe

schedule S_7		Probleme bei zu früher Freigabe
T_{15}	T_{16}	
lock_X(B) read(B, b) $b \leftarrow b - 50$ write(B, b) unlock(B)		<ul style="list-style-type: none"> 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
lock_S(A) read(A, a) unlock(A) lock_S(B) read(B, b) unlock(B) display(A+B)		<p>Frühe Sperrfreigaben können zu inkorrekten Resultaten führen (Non-Serializable Schedules), aber sie erlauben einen höheren Grad an Nebenläufigkeit.</p>
lock_X(A) read(A, A) $a \leftarrow a + 50$ write(A, a) unlock(A)		

Probleme bei später Sperrfreigabe

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

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

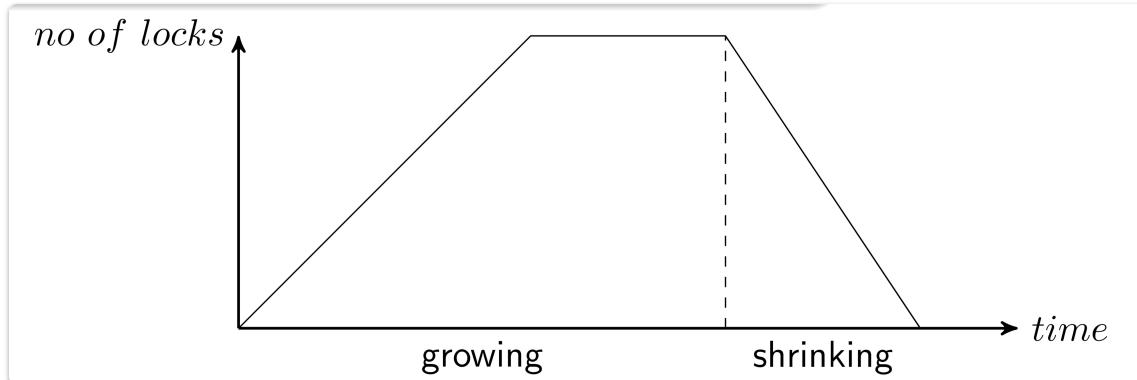
- Späte Sperrfreigaben verhindern Non-Serializable Schedules, aber sie erhöhen die Chancen von **Deadlocks**.
- Lernen Sie damit zu leben!

Das Zwei-Phasen-Sperrprotokoll (2PL)

- Ziel:** Sicherstellen der Konfliktserialisierbarkeit von Schedules.
- Besteht aus zwei Phasen:**
 - 1. Phase (Anforderungsphase, growing phase):**
 - Transaktionen dürfen Locks anfordern.
 - Transaktionen dürfen **keine** Locks freigeben.

2. 2. Phase (Freigabephase, shrinking phase):

- Transaktionen dürfen **keine** neuen Locks anfordern.
- Transaktionen dürfen bisher erworbene Locks freigeben.
- **Übergang:** Sobald der erste Lock freigegeben wird, wechselt die Transaktion von der ersten in die zweite Phase.



Die Abbildung zeigt den typischen Verlauf der Anzahl der gehaltenen Locks über die Zeit während der Ausführung einer Transaktion unter dem 2PL-Protokoll. In der "growing" Phase steigt die Anzahl der Locks an oder bleibt konstant. Sobald die erste Freigabe erfolgt, beginnt die "shrinking" Phase, in der die Anzahl der gehaltenen Locks monoton fällt.

Beispiele: 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				

- **Analyse der Schedules hinsichtlich des 2PL-Protokolls:**

- S_A : Hält sich an 2PL. Die Transaktion T_1 erwirbt alle Locks (A, B, C) und gibt sie dann frei. Es gibt keine Freigabe eines Locks vor der Anforderung eines neuen Locks.
- S_B : Hält sich an 2PL. Die Transaktion T_2 erwirbt alle Locks (A, B, C) und gibt sie dann frei.
- S_C : Hält sich an 2PL. Die Transaktion T_3 erwirbt Lock B und gibt es dann frei. Die Transaktion T_4 erwirbt Lock A und gibt es dann frei.
- S_D : Verstößt gegen 2PL. Die Transaktion T_6 gibt Lock B frei, bevor sie Lock C anfordert. Dies verletzt die Regel, dass nach der Freigabe eines Locks keine neuen Locks mehr angefordert werden dürfen.

Eigenschaften des Zwei-Phasen-Sperrprotokolls

- **2PL erzeugt nur serialisierbare Schedules:**
 - Garantiert Konfliktserialisierbarkeit.
 - 2PL erzeugt eine Untermenge aller möglichen serialisierbaren Schedules. Es gibt serialisierbare Schedules, die nicht durch 2PL erzeugt werden können.
- **2PL schützt nicht vor Deadlocks:** Auch wenn sich alle Transaktionen an das 2PL halten, können Deadlocks entstehen, wenn Transaktionen auf Locks warten, die von anderen Transaktionen gehalten werden, welche wiederum auf Locks warten, die die ersten Transaktionen halten.
- **2PL schützt nicht vor Cascading Rollbacks:** Wenn eine Transaktion Daten liest, die von einer anderen Transaktion geschrieben wurden, welche später abbricht (Rollback), muss die lesende Transaktion möglicherweise ebenfalls zurückgesetzt werden. 2PL alleine verhindert solche kaskadierenden Rollbacks nicht.
- **"Dirty" Reads sind möglich (Lesen von Transaktionen die noch nicht committed wurden):** Im Standard-2PL können Transaktionen Daten lesen, die von anderen Transaktionen geschrieben, aber noch nicht festgeschrieben (committed) wurden. Wenn die schreibende Transaktion später abbricht, hat die lesende Transaktion inkonsistente Daten gelesen.

Cascading Rollbacks

- Ein Abort einer Transaktion kann zu einem Abort anderer Transaktionen führen.

schedule S_{11}			schedule $S_{11'}$		
T_{22}	T_{23}	T_{24}	$T_{22'}$	$T_{23'}$	$T_{24'}$
lock_X(A) lock_X(B) unlock(A) abort	lock_X(A) unlock(A)	lock_X(A)	lock_X(A) lock_X(B) unlock(A) commit	lock_X(A) unlock(A) commit	lock_X(A)

• Diese Schedules verwenden 2PI

- **Beobachtung:** Diese Schedules verwenden 2PL.
- **Folge eines Aborts:** Abort in $T_{22} \Rightarrow T_{23}$ und T_{24} müssen ebenfalls einen Abort auslösen (Cascading Rollback).
- **Wie können wir Cascading Rollbacks verhindern?**
 - Transaktionen dürfen keine **uncommitted Daten lesen** (siehe $S_{11'}$). In $S_{11'}$ liest $T_{23'}$ keine uncommitteten Daten von $T_{22'}$, da $T_{22'}$ die Daten (Lock auf A und B) erst freigibt, nachdem sie committed hat.

Striktes und rigoroses Zwei-Phasen-Sperrprotokoll

Striktes 2PL

- **Regel:** Exclusive Locks werden nicht vor dem Commit der Transaktion freigegeben.
- **Vorteil:** Verhindert "Dirty Reads", da keine uncommitted Daten gelesen werden können.

Rigoroses 2PL

- **Regel:** Alle Locks (Shared und Exclusive) werden erst nach dem Commit der Transaktion freigegeben.
- **Eigenschaft:** Transaktionen können in der Commit-Reihenfolge serialisiert werden.

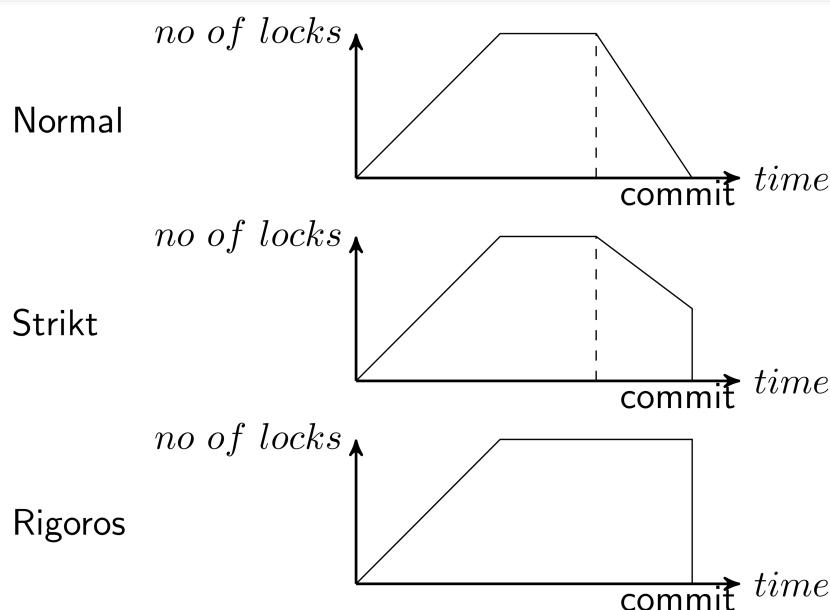
Gemeinsamer Vorteil

- Keine Cascading Rollbacks, da keine uncommitteten Daten von anderen Transaktionen gelesen werden.

Nachteil

- Weniger Nebenläufigkeit, da Locks länger gehalten werden und somit andere Transaktionen länger warten müssen.

Übersicht 2PL Protokolle



Vorteil von rigorosem 2PL gegenüber striktem 2PL

Rigoroses 2PL: Transaktionen können in Commit-Reihenfolge serialisiert werden.

Konvertierung von Sperren

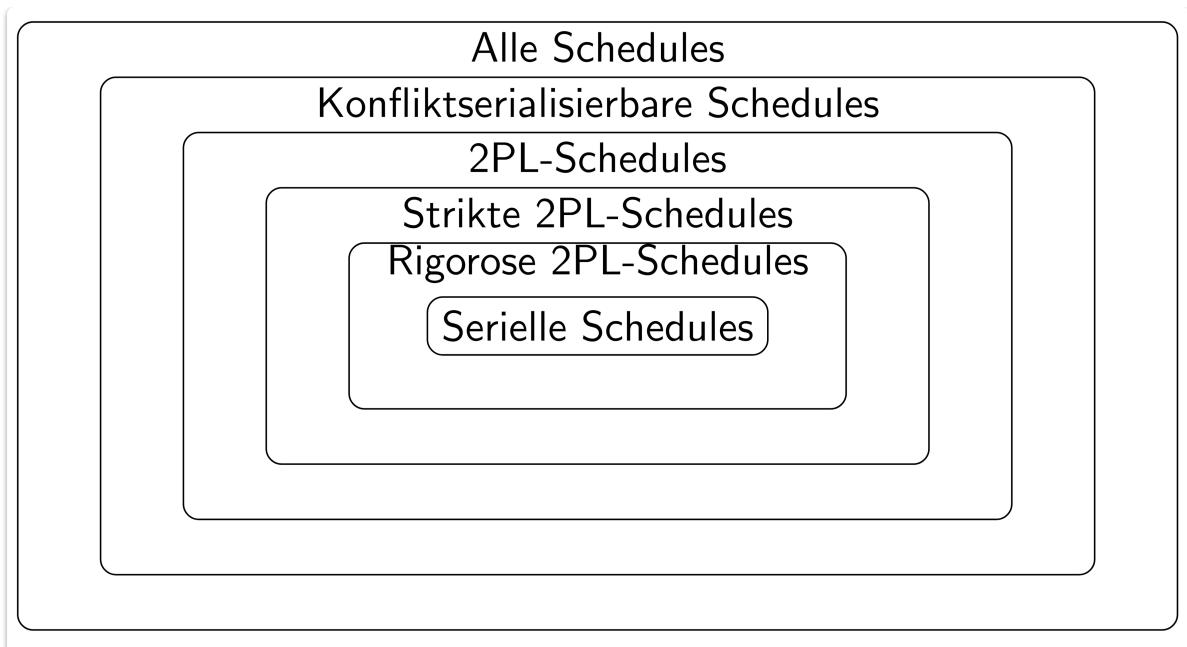
- **Ziel:** 2PL anwenden, aber einen höheren Grad an Nebenläufigkeit erlauben.
- **Erste Phase (Growing Phase):**
 - S-Lock erhalten (lock_S)

- X-Lock erhalten (`lock_X`)
- Konvertieren (Upgrade) eines S-Locks zu einem X-Lock. Eine Transaktion, die bereits einen Shared Lock auf einem Datenobjekt hält, kann diesen in einen Exclusive Lock umwandeln, falls keine andere Transaktion ebenfalls einen Shared Lock auf demselben Objekt hält.
- **Zweite Phase (Shrinking Phase):**
 - Freigabe eines S-Locks (`unlock`)
 - Freigabe eines X-Locks (`unlock`)
 - Konvertieren (Downgrade) eines X-Locks zu einem S-Lock. Eine Transaktion, die einen Exclusive Lock hält, kann diesen in einen Shared Lock umwandeln.
- **Wichtiger Hinweis:**
 - Dieses Protokoll garantiert weiterhin Serialisierbarkeit,
 - ist aber abhängig vom Anwendungsprogrammierer (passende Lock-Operationen müssen explizit im Code eingefügt werden).

Beispiele zu den Arten

schedule S_1	schedule S_2	schedule S_3
T_1	T_2	T_3
<code>lock_S(A)</code>	<code>lock_S(A)</code>	<code>lock_S(A)</code>
<code>lock_S(B)</code>	<code>lock_S(B)</code>	<code>lock_S(B)</code>
<code>lock_X(B)</code>	<code>lock_X(B)</code>	<code>lock_X(B)</code>
<code>lock_S(C)</code>	commit	<code>unlock(B)</code>
<code>unlock(A)</code>	Rigoros	<code>lock_S(C)</code>
<code>unlock(C)</code>		<code>unlock(A)</code>
commit		commit
Strikt		keine 2 Phasen

Übersicht über 2PL-Schedules



Erkennung von Deadlocks

Deadlocks

- **Problem:** 2PL allein kann Deadlocks nicht verhindern.

T_1	T_2	
lock_X(A) read(A) write(A) lock_X(B) ...	lock_S(B) read(B) lock_S(A) ...	T_1 muss warten auf T_2 T_2 muss warten auf T_1 \Rightarrow Deadlock

- **Erläuterung des Deadlocks im Beispiel:**

- T_1 hält ein exklusives Lock auf A und versucht, ein exklusives Lock auf B zu erhalten.
- T_2 hält ein shared Lock auf B und versucht, ein shared Lock auf A zu erhalten.
- Da T_1 ein exklusives Lock auf A hält, kann T_2 kein shared Lock auf A erhalten.
- Da T_2 ein shared Lock auf B hält, kann T_1 kein exklusives Lock auf B erhalten (ein exklusives Lock ist inkompatibel mit einem shared Lock).
- Beide Transaktionen warten aufeinander und können nicht fortfahren, was zu einem Deadlock führt.

- **Lösungen für Deadlocks:**

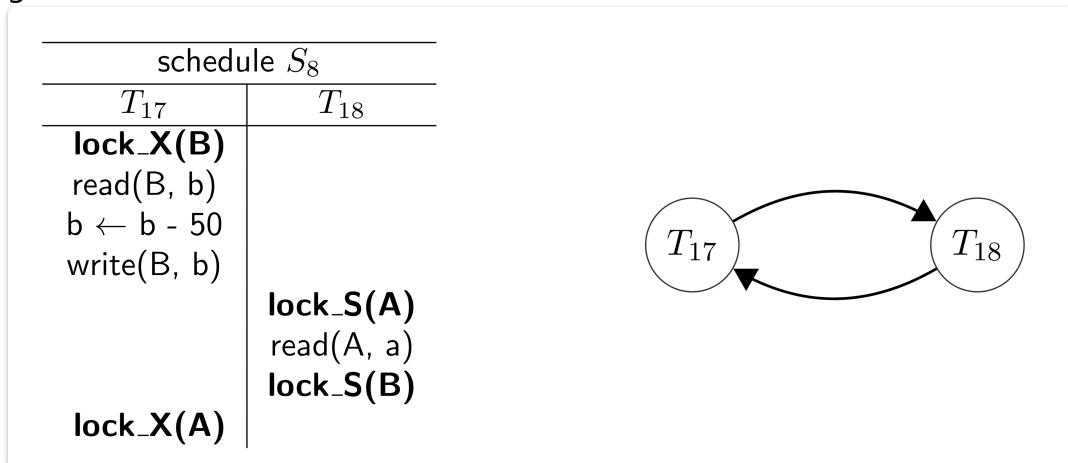
- **Erkennung von Deadlocks und anschließend Recovery:** Das System erkennt, wenn ein Deadlock aufgetreten ist, und ergreift Maßnahmen, um ihn aufzulösen (z.B.

Abbruch einer der beteiligten Transaktionen).

- **Prävention:** Strategien, die verhindern, dass Deadlocks überhaupt entstehen können (z.B. durch die Art und Weise, wie Locks angefordert werden).
- **Timeout:** Eine Transaktion wartet nur eine bestimmte Zeit auf einen Lock. Wenn die Zeit abläuft, wird angenommen, dass ein Deadlock vorliegt, und die Transaktion wird abgebrochen.

Erkennung von Deadlocks

- **Methode:** Erstellen eines Wartegraphen ("Wait-for graph") und Prüfung auf Zyklen.
 - Ein Knoten für jede aktive Transaktion T_i .
 - Eine gerichtete Kante $T_i \rightarrow T_j$ existiert, wenn Transaktion T_i auf die Lock-Freigabe von Transaktion T_j wartet.
 - Ein Deadlock existiert, wenn der Wartegraph einen Zyklus enthält.
- **Vorgehen bei Erkennung eines Deadlocks:**
 - Ein passendes Opfer auswählen (eine der Transaktionen im Zyklus).
 - Abbruch des Opfers und Freigabe aller zugehörigen Sperren, um den Zyklus im Wartegraph zu unterbrechen. Die abgebrochene Transaktion muss später neu gestartet werden.



- **Wartegraph für den Schedule S_8 im Deadlock-Zustand:**
 - $T_{17} \rightarrow T_{18}$ (da T_{17} auf ein Lock von T_{18} wartet - genauer gesagt, auf die Freigabe des Shared Locks auf B, um ein Exclusive Lock zu erhalten).
 - $T_{18} \rightarrow T_{17}$ (da T_{18} auf ein Lock von T_{17} wartet - genauer gesagt, auf die Freigabe des Exclusive Locks auf A, um ein Shared Lock zu erhalten).
 - Der Zyklus $T_{17} \rightarrow T_{18} \rightarrow T_{17}$ zeigt einen Deadlock an.

Beispiel

schedule S_A				
T_1	T_2	T_3	T_4	
lock_X(B) lock_X(C)	lock_X(A)			
lock_X(D)		lock_X(B)		
	lock_X(D)		lock_X(E) lock_X(A)	
lock_X(E)				

Zyklus zwischen T_1 , T_4 und T_2
 ⇒ Deadlock erkannt

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

```

graph TD
    T1((T1)) -- D --> T2((T2))
    T2((T2)) -- E --> T1((T1))
    T1((T1)) -- B --> T3((T3))
    T3((T3)) -- B --> T1((T1))
    T2((T2)) -- A --> T4((T4))
    T4((T4)) -- A --> T2((T2))
  
```

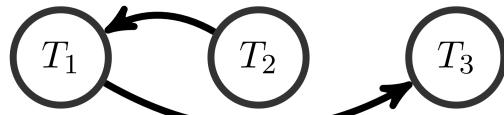
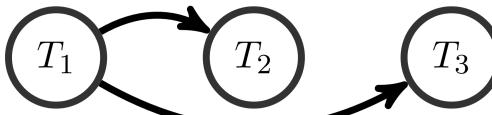
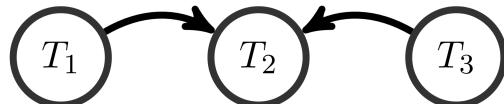
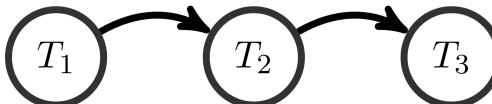
Rollback Kandidaten

- **Wahl einer guten Opfer-Transaktion (bei Deadlock-Erkennung):**
 - Rollback von einer oder mehreren Transaktionen, die am Zyklus beteiligt sind, um den Deadlock aufzulösen.
- **Kriterien für die Auswahl des Opfers:**
 - **Die letzte Transaktion im Zyklus (Minimierung des Rollback-Aufwands):** Oft wird die Transaktion gewählt, die zuletzt in den Wartezyklus eingetreten ist, da sie möglicherweise weniger Arbeit verrichtet hat.
 - **Diejenige, welche die meisten Locks hält (Maximierung der freigegebenen Ressourcen):** Durch den Abbruch einer Transaktion mit vielen Locks werden mehr Ressourcen für andere Transaktionen freigegeben, was potenziell die Wahrscheinlichkeit weiterer Deadlocks verringert.
- **Vermeidung von Starvation:**
 - Aufpassen, dass nicht immer dasselbe Opfer gewählt wird (Starvation). Eine Transaktion könnte wiederholt als Opfer ausgewählt und immer wieder zurückgesetzt werden, ohne jemals zum Abschluss zu kommen.
 - **"Rollback Counter":** Eine mögliche Lösung ist ein Zähler für jede Transaktion, der festhält, wie oft sie bereits zurückgesetzt wurde. Ab einem gewissen Grenzwert wird diese Transaktion nicht mehr als Opfer für einen Rollback ausgewählt, um Starvation zu verhindern.

Wertgraph

Welcher Wartegraph passt zu diesem Schedule?

schedule S_A		
T_1	T_2	T_3
	lock_X(A)	
lock_X(A) lock_X(B)		lock_X(B) lock_X(C) lock_X(D)



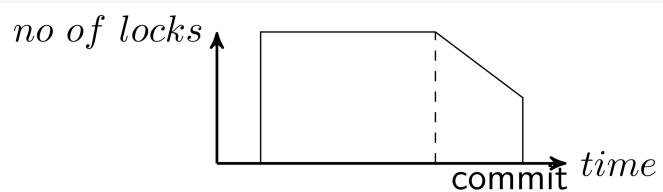
Heir passt nur Option 3 (links unten), weil T_3 wartet auf T_2 und muss auch auf T_3 warten. Da es keinen Zyklus gibt, sieht das gut aus und wir haben keine Probleme / kein Deadlock

Vermeidung von Deadlocks

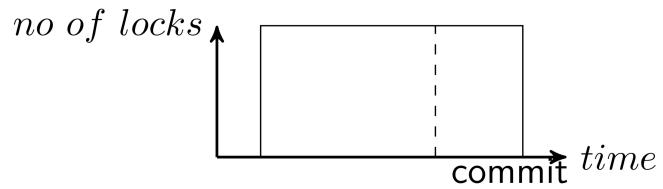
Konservatives 2PL-Protokoll

- Problem:** Normales, striktes und rigoroses 2PL können Deadlocks nicht verhindern.
- Zusätzliche Anforderung im konservativen 2PL:** Alle benötigten Locks (Shared und Exclusive) werden **gleich zu Beginn** einer Transaktion gesetzt. Wenn nicht alle Locks sofort verfügbar sind, wartet die Transaktion, bis alle angefordert werden können.

konservatives striktes 2PL



konservatives rigoroses 2PL



Die Abbildungen zeigen, dass im konservativen 2PL die Anzahl der gehaltenen Locks während der gesamten Lebensdauer der Transaktion bis zum Commit konstant bleibt. Es gibt keine "growing" oder "shrinking" Phase im herkömmlichen Sinne.

- **Verwendung:** Nur in wenigen Applikationen verwendbar, da es die Nebenläufigkeit stark einschränkt.

Zusammenfassung Mehrbenutzersynchronisation

- Mehrere Protokolle zur Mehrbenutzersynchronisation wurden entwickelt.
 - **Hauptziel:** Nur serialisierbare, recoverable und cascadeless Schedules zuzulassen.
 - **Zwei-Phasen-Sperrprotokoll (2PL):** Ein weit verbreitetes Protokoll zur Sicherstellung der Konfliktserialisierbarkeit.
 - Die meisten relationalen DBMS benutzen **rigoroses 2PL**, um Serialisierbarkeit und die Vermeidung von Cascading Rollbacks zu gewährleisten.
 - **Deadlock-Behandlung:**
 - **Erkennung von Deadlocks (Wartegraph)** und anschließendes Recovery (Rollback einer Transaktion).
 - **Verhindern von Deadlocks (konservatives 2PL):** Durch das sofortige Anfordern aller benötigten Locks wird die Entstehung von Zyklen in Wartegraphen vermieden.
 - **Trade-off:** Serialisierbarkeit vs. Nebenläufigkeit. Strengere Serialisierungsprotokolle (wie rigoroses oder konservatives 2PL) reduzieren oft die mögliche Nebenläufigkeit.
-

Recovery

Fehlerklassifikation

- **Aspekte:**
 - **Atomarität:**
 - Transaktionen können einen Abort ausführen (Rollback). Entweder werden alle Operationen einer Transaktion erfolgreich abgeschlossen (Commit), oder keine von ihnen hat einen dauerhaften Effekt auf die Datenbank (Abort).
 - **Dauerhaftigkeit:**
 - Was muss passieren, wenn das DBMS abstürzt? Nach einem Commit einer Transaktion müssen die Änderungen dauerhaft in der Datenbank gespeichert bleiben, auch bei Systemfehlern.
- **Garantie des DBMS:** Das DBMS garantiert, dass eine Transaktion
 - entweder fertig wird und ein permanentes Resultat liefert (committed)
 - oder keinen Effekt auf die Datenbank hat (aborted).
- **Rolle der Recovery-Komponente:** Die Rolle der Recovery-Komponente ist es, die Atomarität und Dauerhaftigkeit von Transaktionen trotz etwaiger Systemfehler zu garantieren.

Wie kann Dauerhaftigkeit garantiert werden?

Noch nicht auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **noch nicht** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem Blackout (Systemausfall) in diesem Moment?
- Welche Daten sind tatsächlich in der Datenbank auf der Festplatte gesichert? Die Änderungen der gerade committeten Transaktion **könnten verloren sein**, wenn sie noch nicht persistent gespeichert wurden.

Teilweise auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **teilweise** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem Blackout (Systemausfall) in diesem Moment?
- Welche Daten sind in der Datenbank? Es ist unklar, ob alle Änderungen der committeten Transaktion persistent gespeichert wurden, da der Schreibvorgang möglicherweise unterbrochen wurde. Dies kann zu einer **inkonsistenten Datenbank** führen, bei der einige, aber nicht alle Änderungen der Transaktion übernommen wurden.

Vollständig auf der Festplatte

- **Szenario:**

- Eine Transaktion verändert Daten im Arbeitsspeicher.
- Die Daten wurden **vollständig** auf die Festplatte geschrieben.
- Ein Commit wird durchgeführt.
- Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.

- **Problem:**

- Was passiert bei einem **Hardware-Fehler**, z.B. dem **Verlust einer Festplatte**?

- Welche Daten sind in der Datenbank? Obwohl die Daten der Transaktion auf *einer* Festplatte gespeichert wurden, sind sie bei einem Ausfall dieser spezifischen Festplatte verloren.
- **Schlussfolgerung:** Das alleinige Schreiben auf eine Festplatte garantiert keine vollständige Dauerhaftigkeit gegen Hardware-Fehler. Es sind zusätzliche Mechanismen zur Datensicherung und Redundanz erforderlich.

Vollständig auf mehreren Festplatten

- **Szenario:**
 - Eine Transaktion verändert Daten im Arbeitsspeicher.
 - Die Daten wurden **vollständig auf mehrere Festplatten** geschrieben (Redundanz).
 - Ein Commit wird durchgeführt.
 - Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.
- **Problem:**
 - Was passiert, wenn es ein schwerwiegendes Ereignis wie Feuer, Flut, Erdbeben oder ähnliches gibt, das den physischen Standort der Datenbank betrifft und **alle Festplatten zerstört oder unzugänglich macht?**
 - Welche Daten sind in der Datenbank? In diesem Fall wären die Daten, obwohl redundant auf mehreren lokalen Festplatten gespeichert, **verloren**, da alle Kopien gleichzeitig zerstört wurden.
- **Schlussfolgerung:** Um gegen solche katastrophalen Ereignisse gewappnet zu sein, sind **externe Backups** und **Disaster-Recovery-Pläne** unerlässlich, bei denen Daten an einem geografisch entfernten Ort gesichert werden.

Alle Festplatten kaputt

- **Szenario:**
 - Eine Transaktion verändert Daten im Arbeitsspeicher.
 - Die Daten wurden vollständig auf mehrere Festplatten geschrieben, und die Festplatten wurden auf mehreren **geografisch verschiedenen Computerzentren** verteilt.
 - Ein Commit wird durchgeführt.
 - Der Benutzer nimmt an, die Transaktion wäre erfolgreich abgeschlossen und alle Änderungen sind in der Datenbank gespeichert.
- **Problem:**
 - Was passiert, wenn es ein extrem unwahrscheinliches, aber theoretisch mögliches Ereignis wie Feuer, Flut, Erdbeben oder ähnliches **gleichzeitig bei allen Computerzentren** gibt, sodass alle Standorte und damit alle Kopien der Daten verloren gehen?
 - Welche Daten sind in der Datenbank? In diesem extremen Szenario wären **alle Daten verloren**.

- **Schlussfolgerung:** Obwohl die Wahrscheinlichkeit eines solchen globalen Datenverlusts durch geografische Verteilung extrem gering ist, zeigt es die fundamentale Abhängigkeit von der physischen Existenz der Datenträger. Um absolute Sicherheit zu gewährleisten, bräuchte es hypothetisch widerstandsfähige Speicherorte außerhalb unseres Planeten oder ähnliche futuristische Lösungen. In der praktischen Realität stellt die geografische Verteilung über mehrere Rechenzentren jedoch einen extrem hohen Grad an Ausfallsicherheit dar.

Dauerhaftigkeit (Durability)

- **Relativität:** Dauerhaftigkeit ist **relativ** und hängt von der Anzahl der Datenkopien und deren geografischer Verteilung ab. Je mehr Kopien an unterschiedlichen Orten existieren, desto höher ist die Wahrscheinlichkeit, dass Daten auch bei Ausfällen einzelner Komponenten oder Standorte erhalten bleiben.
- **Garantien:** Garantien für Dauerhaftigkeit sind nur möglich, wenn:
 - wir **zuerst** die Kopien der Daten persistent aktualisieren (z.B. auf Festplatten schreiben)
 - und **erst dann** den Benutzer darüber informieren, dass der Commit der Transaktion erfolgreich war. Dieses Prinzip wird oft durch das **Write-Ahead Logging (WAL)** sichergestellt.
- **Annahme:** Wir nehmen deshalb an, dass die WAL-Regel erfüllt ist. Das bedeutet, dass Änderungen an der Datenbank zuerst in einem Logfile protokolliert werden, bevor sie in die eigentliche Datenbank geschrieben werden. Dieses Log ermöglicht es dem System, nach einem Absturz den Zustand der Datenbank wiederherzustellen.
- **Variationen der WAL-Regel (Strategien zur Erhöhung der Dauerhaftigkeit):**
 - **Log-Based Recovery:** Die Wiederherstellung der Datenbank nach einem Fehler basiert auf der Analyse des Transaktionslogs.
 - **Volle Redundanz:** Spiegeln aller Daten auf mehreren Computern (Festplatten, Rechenzentren), die alle die gleichen Operationen ausführen. Dies bietet hohe Fehlertoleranz und Datenverfügbarkeit.

Fehlerklassifikation

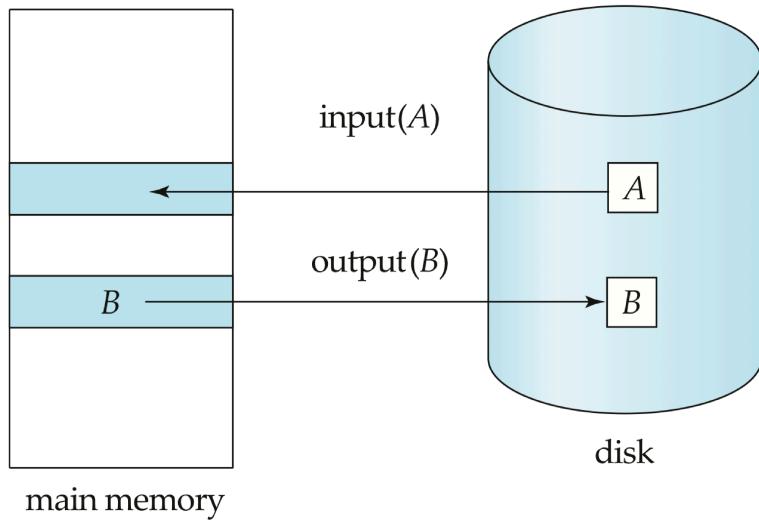
- **Transaktionsfehler (Fehler einer Transaktion, die noch nicht committed wurde):**
 - **Maßnahme:** Rückgängigmachen der Änderungen (Rollback) der betroffenen Transaktion, um die Atomarität zu gewährleisten.
- **Systemabsturz (Fehler mit Hauptspeicherverlust):**
 - **Anforderungen an die Recovery:**
 - Änderungen von Transaktionen, die **committed** wurden, müssen erhalten bleiben (Dauerhaftigkeit).
 - Änderungen von Transaktionen, die **nicht committed** wurden, müssen rückgängig gemacht werden (Atomarität).
- **Festplattenfehler:**

- **Maßnahme:** Recovery basiert in der Regel auf Archiven oder Datenbank-Dumps (regelmäßige Sicherungskopien der Datenbank), die auf anderen Speichermedien gespeichert sind. Zusätzlich können Transaktionslogs verwendet werden, um Änderungen seit dem letzten Backup wiederherzustellen (Log-Based Recovery).

Datenspeicher

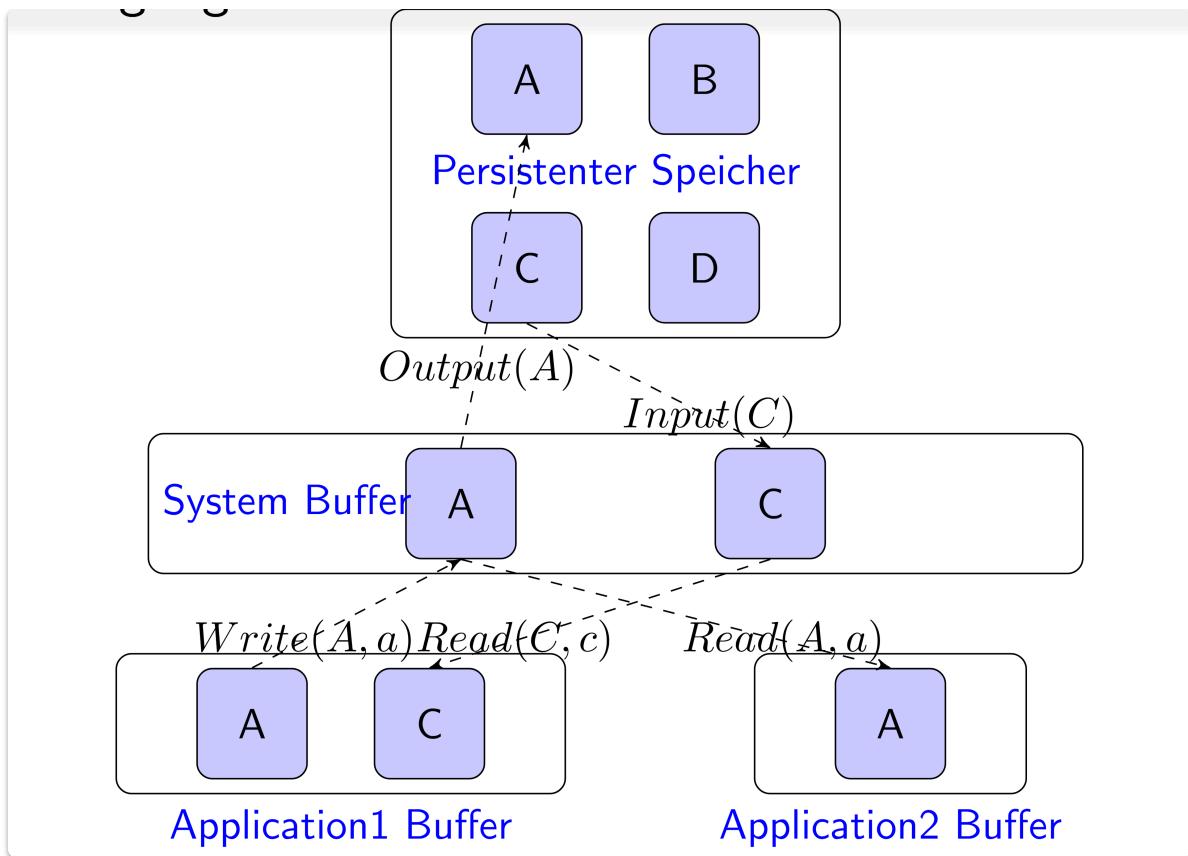
Zweistufige Speicherhierarchie

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



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

Bewegung der Daten



Speicheroperationen

- 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):** Transferiert den Block, der das Datenobjekt Q beinhaltet, in den Arbeitsspeicher. Dieser Vorgang ist notwendig, um auf die Daten zugreifen zu können.
 - **Output(Q):** Transferiert den Block, der das Datenobjekt Q beinhaltet, auf die Festplatte. Dieser Vorgang dient dazu, Änderungen persistent zu speichern.
- **Operationen, um Werte zwischen Datenobjekten und lokalen Variablen zu verschieben:**
 - **read(Q, q):** Weist den Wert des Datenobjekts Q der lokalen Variable q der Transaktion zu.
 - **write(Q, q):** Weist den Wert der lokalen Variable q dem Datenobjekt Q in der Datenbank (im Buffer) zu. Diese Änderung wird möglicherweise erst später mit Output(Q) auf die Festplatte geschrieben.

Log-Einträge

Die WAL-Regel für Log-Based Recovery

WAL (Write Ahead Logging)

- **Regel 1 (Commit-Regel):** Bevor eine Transaktion in den Commit-Zustand wechselt, müssen **alle zugehörigen Log-Einträge** auf einen stabilen Speicher (z.B. Festplatte) geschrieben worden sein, **inklusive dem Commit-Log-Eintrag** selbst. Dies stellt sicher, dass ein Commit auch nach einem Systemabsturz nachvollzogen werden kann.
- **Regel 2 (Write-Regel):** Bevor eine modifizierte Seite (oder ein Block) im Arbeitsspeicher in die Datenbank (nichtflüchtiger Speicher) geschrieben werden kann, müssen **alle zugehörigen Log-Einträge** für diese Änderung bereits auf einem stabilen Speicher geschrieben worden sein. Dies gewährleistet, dass die Log-Informationen zur Wiederherstellung verfügbar sind, falls ein Absturz während des Schreibens der Daten in die Datenbank erfolgt.

Zusammenfassend: Die WAL-Regel besagt, dass das Transaktionslog, das alle Änderungen und den Commit-Status festhält, immer persistent gespeichert sein muss, bevor die eigentlichen Datenbankänderungen auf die Festplatte geschrieben werden oder bevor eine Transaktion als committed gilt. Dies ist die Grundlage für eine zuverlässige Wiederherstellung nach Systemausfällen.

Logging

Im normalen Betrieb

- **Transaktionsstart:** Am Beginn registriert sich eine Transaktion T selbst im Log: $[T \text{ start}]$
- **Datenobjektänderung (write(X, x)):** Wenn ein Datenobjekt X durch die Transaktion T auf den neuen Wert x gesetzt wird:
 1. **Log-Eintrag:** Folgender Log-Eintrag wird dem Log hinzugefügt:
 - $[T, X, V\text{-alt}, V\text{-neu}]$
 - T : Transaktions-ID
 - X : Name des Datenobjekts
 - $V\text{-alt}$: Alter Wert des Objekts vor der Änderung
 - $V\text{-neu}$: Neuer Wert des Objekts nach der Änderung
 2. **Schreiben des neuen Werts:** Der neue Wert von X wird im Arbeitsspeicher (Buffer) aktualisiert. Der Buffer Manager schreibt diese geänderten Seiten später asynchron auf die Festplatte.
- **Transaktionsende (Commit):** Am Ende der Transaktion fügt Transaktion T den Eintrag $[T \text{ commit}]$ ins Log ein.
- **Commit-Definition:** Eine Transaktion gilt als **committed** genau dann, wenn der Commit-Eintrag (nach allen vorherigen Log-Einträgen der Transaktion) **im Log steht**. Dies ist ein zentraler Punkt für die WAL-Regel.

Struktur eines Log-Eintrags (Log Record)

- **Update-Eintrag:**
 - Format: $[TID, DID, old, new]$
 - **TID:** ID der Transaktion, die die Änderung verursacht hat.

- **DID:** ID des Datenobjekts. Dies kann die genaue Speicheradresse auf der Festplatte angeben (z.B. Seite, Block, Offset).
 - **old:** Wert des Datenobjekts vor der Änderung.
 - **new:** Wert des Datenobjekts nach der Änderung.
- **Zusätzliche Steuerungseinträge:**
- **Start-Eintrag:** [TID start] - Markiert den Beginn der Transaktion mit der ID TID.
 - **Commit-Eintrag:** [TID commit] - Markiert das erfolgreiche Ende (Commit) der Transaktion mit der ID TID.
 - **Abort-Eintrag:** [TID abort] - Markiert das abgebrochene Ende (Abort) der Transaktion mit der ID TID.

Beispiele

Generelles Log-Einträge-Beispiel

schedule S_1			Log-Einträge Beispiel
T_1	T_2	T_3	
begin read(B, b) $b \leftarrow b+100$ write(B, b) commit	begin read(D, d) $d \leftarrow d+470$ write(D, d) commit	begin read(D, d) read(E, e) $d \leftarrow d-10$ write(D, d) $e \leftarrow e-20$ write(E, e) commit	[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]

Finde die Fehler

[T1 commit]
[T1, B, 300, 400]
[T1 start]

Commit vor Start

[T1 start]
[T1, B, 300, 400]
[T1 commit]
[T1, C, 40, 540]

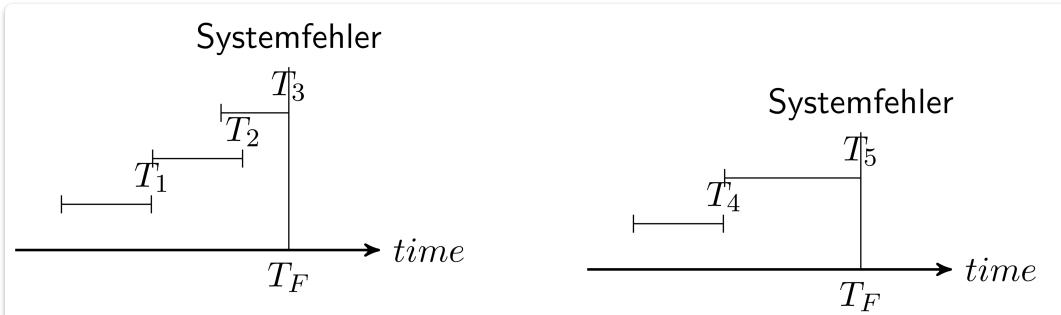
Es darf keine Log-Einträge für T1 nach dem Commit geben.

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

Mehrere Start-Einträge für T1

Log-Based Recovery

- **Operationen zur Wiederherstellung nach Fehlern:**
 - **Redo:** Die Änderungen an der Datenbank wiederholen, die von zum Zeitpunkt des Systemfehlers committed waren.
 - **Undo:** Den Zustand der Datenobjekte auf den Wert vor der Ausführung der Operationen von Transaktionen zurücksetzen, die zum Zeitpunkt des Systemfehlers nicht committed waren (aborted oder noch aktiv).



Die Abbildungen zeigen den Zeitpunkt des Systemfehlers (T_F) im Verhältnis zur Lebensdauer verschiedener Transaktionen (T_1, T_2, T_3, T_4, T_5).

- **Analyse der Transaktionen im Hinblick auf Redo/Undo:**
 - **Linke Abbildung (Systemfehler nach T_3):**
 - **Redo T_1 und T_2 :** Da T_1 und T_2 vor dem Systemfehler committed wurden, müssen ihre Änderungen in der Datenbank wiederhergestellt werden (Redo).
 - **Undo T_3 :** Da T_3 zum Zeitpunkt des Systemfehlers noch nicht committed war, müssen alle ihre bisherigen Änderungen rückgängig gemacht werden (Undo).
 - **Rechte Abbildung (Systemfehler nach T_5):**
 - **Redo T_4 :** Da T_4 vor dem Systemfehler committed wurde, müssen ihre Änderungen wiederholt werden (Redo).
 - **Undo T_5 :** Da T_5 zum Zeitpunkt des Systemfehlers nicht committed war, müssen ihre Änderungen rückgängig gemacht werden (Undo).

Recovery Algorithmus

- **Grundlegende Schritte:**

- **Reproduzieren (redo) der Resultate von Transaktionen, die committed wurden:**
Anhand des Logs werden die Änderungen der erfolgreich abgeschlossenen Transaktionen erneut angewendet, um sicherzustellen, dass ihre Effekte in der Datenbank erhalten bleiben.
- **Rückgängigmachen (undo) von Transaktionen, die noch nicht committed wurden:**
Anhand des Logs werden die Änderungen unvollständiger Transaktionen entfernt, um die Atomarität zu gewährleisten. Die Datenbank wird in einen Zustand zurückversetzt, als ob diese Transaktionen nie begonnen hätten.

- **Anmerkungen:**

- In einem Multitasking-System müssen eventuell mehr als eine Transaktion rückgängig gemacht werden, falls mehrere Transaktionen zum Zeitpunkt des Fehlers aktiv waren.
- **Idempotenz der Recovery:** Wenn ein Systemabsturz während der Recovery-Phase auftritt, muss auch ein erneuter Recovery-Anlauf korrekte und konsistente Resultate liefern. Das Recovery-Verfahren muss idempotent sein, d.h., es darf keine schädlichen oder inkonsistenten Nebeneffekte haben, wenn es mehrmals ausgeführt wird.

Log-Based Recovery

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

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

Drei Recovery-Phasen

Phase 1:

- **Redo (Vollständige Wiederholung der Historie)**
 - Alle Log-Einträge werden Schritt für Schritt in chronologischer Reihenfolge durchgegangen (vorwärts).
 - Alle im Log verzeichneten Änderungen werden in derselben Reihenfolge auf die Datenbank angewendet (Redo).

- **Bestimmen der "Undo"-Transaktionen:**

- Für jeden `[T_i start]` Eintrag im Log wird die Transaktion T_i zur "Undo List" hinzugefügt. Dies sind potenziell unvollständige Transaktionen.
- Wenn ein `[T_i commit]` oder `[T_i abort]` Eintrag für eine Transaktion T_i gefunden wird, wird T_i von der "Undo List" entfernt, da diese Transaktion entweder erfolgreich abgeschlossen oder explizit abgebrochen wurde. Die Transaktionen, die am Ende dieser Phase noch in der "Undo List" sind, waren zum Zeitpunkt des Systemfehlers aktiv und nicht committed.

Phase 2:

- **Undo (Rollback aller Transaktionen in der "Undo List")**

- Alle Log-Einträge werden **rückwärts** durchgegangen (vom Ende zum Anfang).
- Für jede Transaktion T_i in der "Undo List" werden alle ihre im Log verzeichneten Änderungen rückgängig gemacht (Undo).
- Für jede rückgängig gemachte Operation wird ein **Compensation-Log-Eintrag** erstellt, der die inverse Operation protokolliert. Dies ist wichtig für den Fall eines erneuten Fehlers während der Undo-Phase.
- Für jeden `[T_i start]` Eintrag einer Transaktion T_i in der "Undo List" wird ein `[T_i abort]` Eintrag ins Log geschrieben und T_i von der "Undo List" entfernt, da alle ihre Änderungen nun rückgängig gemacht wurden.
- Die Undo-Phase stoppt, sobald die "Undo List" leer ist, d.h., alle unvollständigen Transaktionen zurückgesetzt wurden.

Compensation-Log-Einträge

- **Format:** `[TID, DID, value]`
 - **TID:** ID der Transaktion, die den Ausgleich vornimmt (in der Recovery-Phase ist dies oft eine spezielle Systemtransaktion).
 - **DID:** ID des betroffenen Datenobjekts.
 - **value:** Der Wert, auf den das Datenobjekt zurückgesetzt wird (der ursprüngliche Wert vor der Änderung der unvollständigen Transaktion).
- **Zweck:** Erstellt zum Rückgängigmachen (Ausgleichen/Kompensieren) der Änderungen, die durch einen vorherigen Log-Eintrag der Form `[TID, DID, value, newValue]` einer unvollständigen Transaktion verursacht wurden. Der Compensation-Log-Eintrag stellt somit den ursprünglichen Zustand wieder her.
- **Redo-Only-Log-Eintrag:** Compensation-Log-Einträge sind in der Regel "Redo-Only". Das bedeutet, dass während der Redo-Phase eines späteren Recovery-Prozesses diese Einträge angewendet werden, um sicherzustellen, dass die Rückgängigmachung der unvollständigen Transaktion bestehen bleibt. Es ist nicht notwendig, Compensation-Log-Einträge selbst wieder rückgängig zu machen.
- **Verwendung im normalen Betrieb:** Compensation-Log-Einträge können auch für einen expliziten Rollback einer Transaktion während der normalen Ausführung verwendet werden.

werden. In diesem Fall werden sie erzeugt, um die bereits erfolgten Änderungen der Transaktion zu neutralisieren.

Beispiel

Ein langes Beispiel dafür gibt es von [DBS-7_transaktionen part 2, p.90](#) bis [hier](#) in den Slides.

ARIES

- **State-Of-The-Art Methode** für Log-Based Recovery.
 - **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:** Verwendet eine Dirty Page Table (pageID, recLSN), eine 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 Atomarität und Dauerhaftigkeit trotz Systemfehlern und Abstürzen.
- **Dauerhaftigkeit ist relativ** und hängt von der Redundanz und der geografischen Verteilung der Daten ab.
- **WAL-Regel (Write Ahead Logging):** Log-Einträge müssen persistent gespeichert sein, bevor die zugehörigen Datenbankänderungen geschrieben oder ein Commit durchgeführt wird.
- **Log-Based Recovery:** Ein Verfahren zur Wiederherstellung der Datenbank mithilfe eines Transaktionslogs.
 - Alle Änderungen müssen in eine Log-Datei geschrieben werden.
 - Eine Transaktion führt ein Commit genau dann durch, wenn der Commit-Eintrag im Log geschrieben wurde.