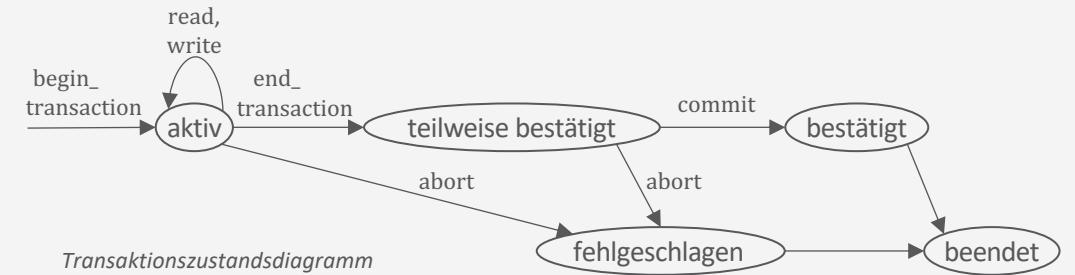


# Transaktionen

Datenbanken



# Inhalte: Datenbanken (DBs)

## 1. Einführung

- Anwendungen
- Datenbankmanagementsysteme

## 2. Datenbank-Modellierung

- Entity-Relationship-Modell (ER-Modell)
- Beziehung zwischen ER und UML

## 3. Das relationale Modell

- Relationales Datenmodell (RM)
- Vom ER-Modell zum RM
- Relationale Algebra als Anfragesprache

## 4. Relationale Entwurfstheorie

- Funktionale Abhängigkeiten
- Normalformen

## 5. Structured Query Language (SQL)

- Datendefinition
- Datenmanipulation

## 6. Anfrageverarbeitung

- Architektur
- Indexierung
- Anfragepläne, Optimierung

## 7. Transaktionen

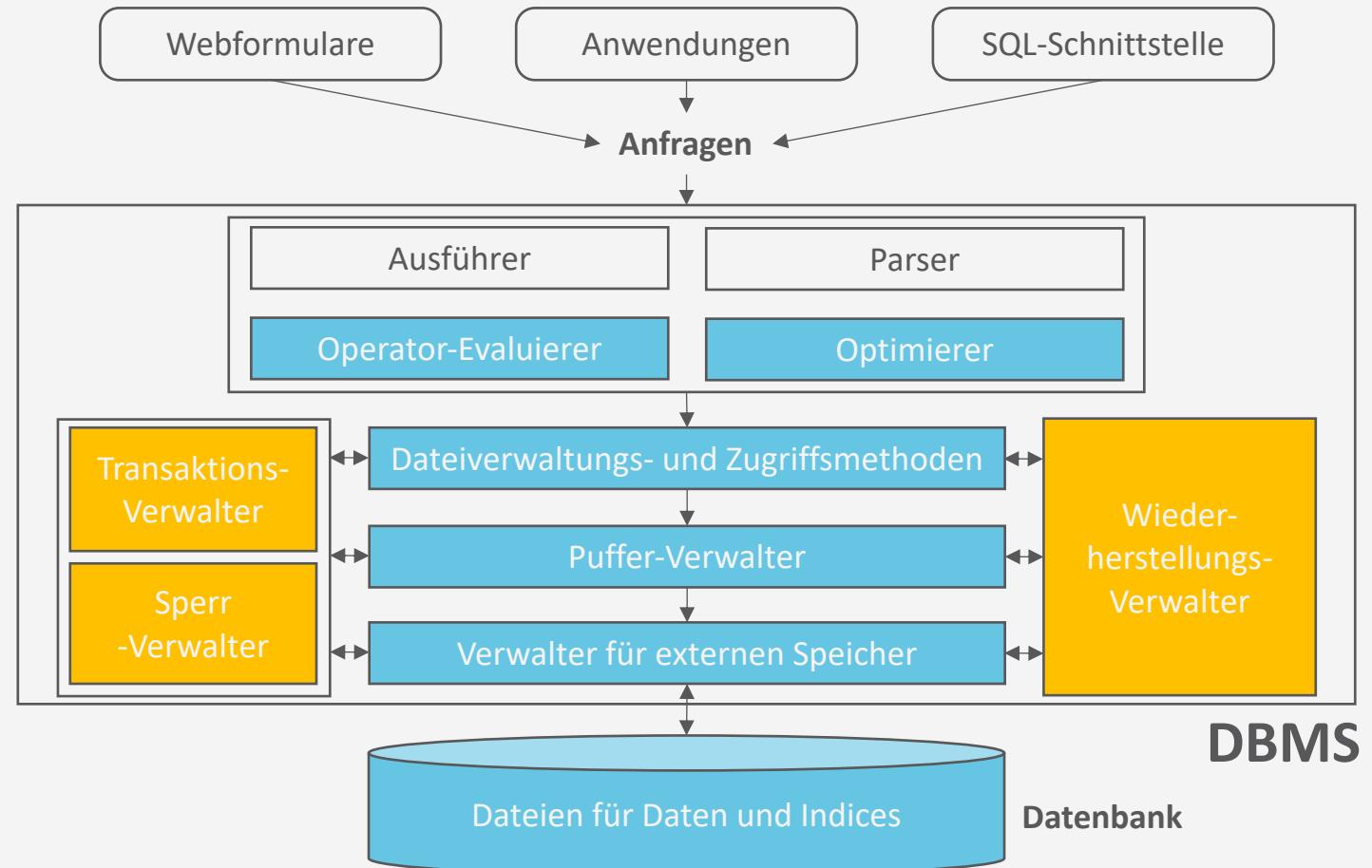
- Transaktionsverarbeitung, Schedules, Sperren
- Wiederherstellung

## 8. Verteilte Datenbanken

- Fragmentierung, Replikation, Allokation; CAP
- Anfragebeantwortung, föderierte Systeme

# Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



# Überblick: 7. Transaktionen

## A. *Transaktionsverarbeitung*

- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

## B. *Sperrverwaltung*

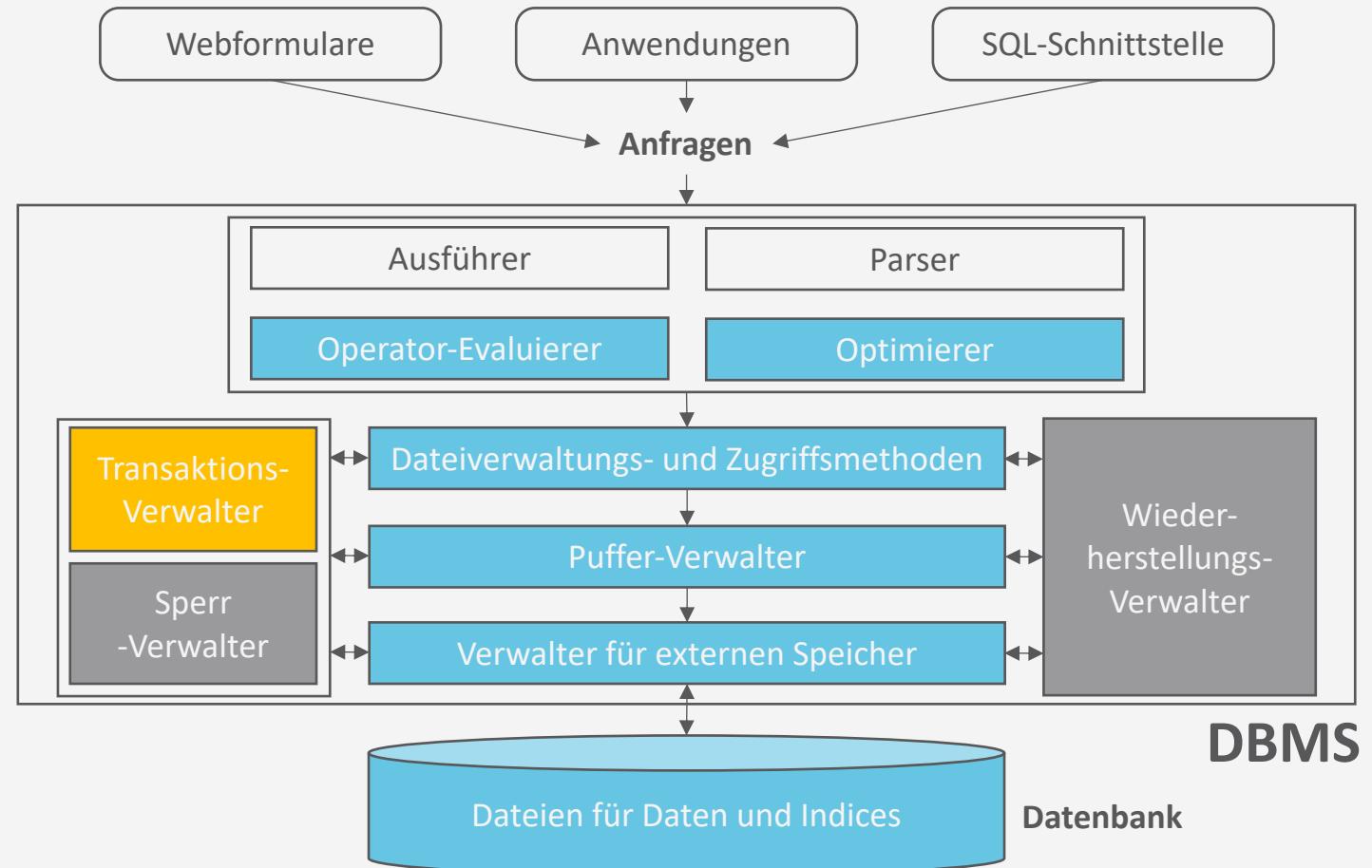
- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

## C. *Wiederherstellungsverwaltung*

- Fehlersituationen
- Logging
- Recovery

# Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung

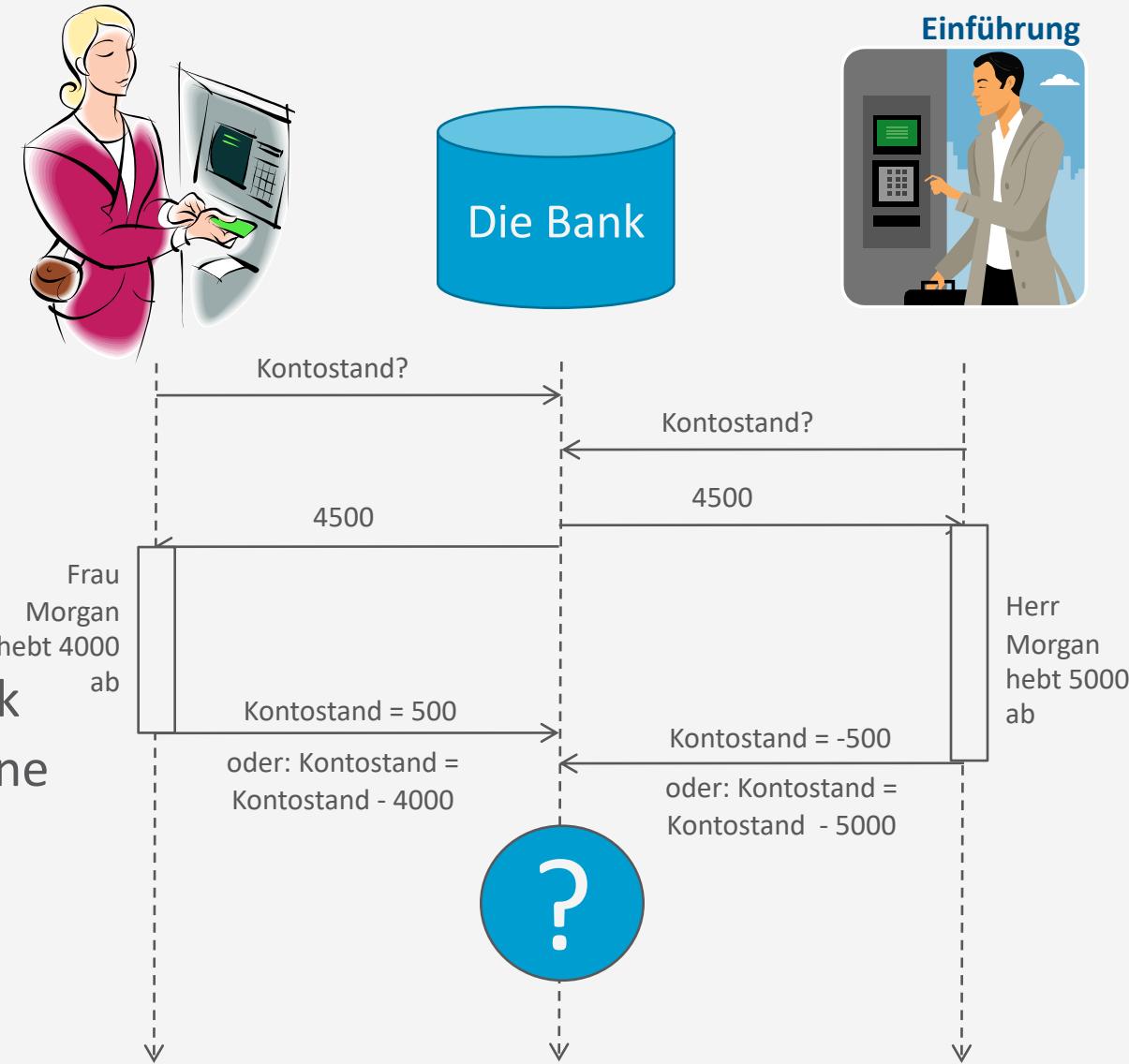


# Viele gleichzeitige Nutzer\*innen

- Jede\*r Nutzer\*in
  - Stellt Anfrage an den Kontostand
  - Verändert den Kontostand
 → Abfolge von DB-Befehlen = **Transaktion**
- Anforderungen:
  - **Atomicity** (Atomarität): Alles oder nichts
  - **Consistency** (Konsistenz): Vorher ok, hinterher ok
  - **Isolation** (Isolation): Jede\*r denkt, sie seien alleine auf der DB
  - **Durability** (Dauerhaftigkeit): Transaktionen bestätigt? Dann sind die Daten jetzt sicher

*ACID-Eigenschaften (später mehr)*

*Es ist später*

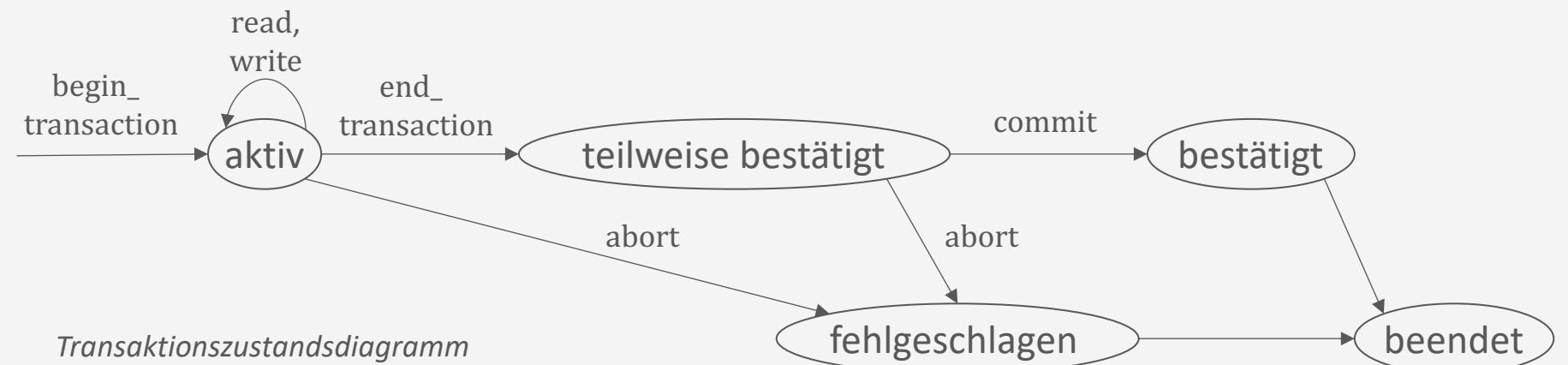


## DB-Transaktion - Definition

- **(DB-)Transaktion:** logische Verarbeitungseinheit auf einer DB
  - Enthält typischerweise mehrere DB-Operationen (Lesen, Einfügen, Aktualisieren, Löschen)
  - Können aus Anwendungsprogrammen heraus oder im Rahmen manueller Eingaben über eine SQL-Schnittstelle realisiert werden
  - Werden mit einem speziellen Schlüsselwort gestartet (**BEGIN\_TRANSACTION, BOT**) und beendet (**END\_TRANSACTION, EOT**)
    - Gelten nach Ausführung von END\_TRANSACTION als teilweise (bzw. vorläufig) bestätigt
      - Sind dann aber erst „logisch“ und noch nicht physisch persistent abgeschlossen
      - Werden erst nach **COMMIT** endgültig physisch durchgeführt (Vgl. Logging, siehe später)
      - Müssen zurückgesetzt werden (**ABORT**), wenn die Persistenz nicht durch COMMIT erreicht wird
    - Transaktion muss mit COMMIT oder ABORT enden
  - **Lesende Transaktion:** DB wird innerhalb einer Transaktion nicht verändert

## Schüsselwörter einer Transaktion

- **begin\_transaction**: Startet eine Transaktion
- **end\_transaction**: Beendet eine Transaktion logisch
- **commit** (Bestätigung): Fügt die Ergebnisse der Transaktion persistent in den Datenbestand ein
- **abort**: Abbruch einer Transaktion, DB-Änderungen werden vollständig verworfen bzw. zurückgenommen



## Lese- und Schreib-Operationen

- Für diese Lerneinheit: Datenbank = **Sammlung von Datenobjekten**
  - **Datenobjekt** kann Feld (Attribut) eines Datensatzes (Tupels), ein gesamter Datensatz oder ein Plattenblock (Zusammenfassung vieler Datensätze) sein  
→ Abstraktion von deren Größe (Granularität)
- Grundlegenden DB-Zugriffsoperationen in Transaktionen:
  - **read\_item( $X$ )**:
    - Liest ein Datenobjekt  $X$  aus einer DB in eine Programmvariable
    - Zur Vereinfachung der Notation wird angenommen, dass die Programmvariable ebenfalls  $X$  heißt
  - **write\_item( $X$ )** :
    - Schreibt den Wert einer Programmvariablen  $X$  in das entsprechende Datenobjekt  $X$  der DB
  - Beispiel: Transaktionen  $T_1, T_2$

$T_1$	$T_2$
read_item( $X$ )	read_item( $X$ )
$X := X - N$	$X := X + M$
write_item( $X$ )	write_item( $X$ )
read_item( $Y$ )	
$Y := Y + N$	
write_item( $Y$ )	

## Realisierung von Read und Write

- Aktionen zur Realisierung einer `read_item( $X$ )`-Anweisung:
  1. Suche Adresse des Plattenblocks, der Datenobjekt  $X$  enthält
  2. Kopiere den ermittelten Plattenblock in Arbeitsspeicher (sofern der Plattenblock noch extern ist, d.h. sich nicht im internen Arbeitsspeicher befindet)
  3. Kopiere den Wert des Datenobjekts  $X$  in die entsprechende Programm-variable  $X$
- Aktionen zur Realisierung einer `write_item( $X$ )`-Anweisung:
  1. Suche Adresse des Plattenblocks, der Datenobjekt  $X$  enthält
  2. Kopiere den ermittelten Block in Arbeitsspeicher (sofern der Plattenblock noch extern ist, d.h. sich nicht im internen Arbeitsspeicher befindet)
  3. Kopiere den Wert der Programmvariablen  $X$  an die entsprechende Stelle im Arbeitsspeicher
  4. Schreibe den modifizierten Block **sofort oder später** in den Plattenblock

# Fehlersituationen

Transaktionsverarbeitung



## Fehlermöglichkeiten

- Eine (verschachtelt) nebenläufige Ausführung der in  $T_1$  und  $T_2$  enthaltenen Anweisungen ohne jegliche Kontrollmechanismen, die die korrekte Ausführung sicherstellen, kann in verschiedene Fehlersituationen münden:
  - Lost Update
  - Dirty Read
  - Ghost Update
  - Unrepeatable Read
- Für die vier grundsätzlichen Fehlersituationen folgt je ein Beispiel zur Erläuterung. Die Fehler entstehen je nach konkreter zeitlicher Abfolge der einzelnen Operationen in nebenläufigen Transaktionen.
- Für alle Beispiele in diesem Foliensatz: Annahme, dass die Transaktionen mit ihrer letzten Operation logisch beendet werden (`end_transaction`)

## LOST UPDATE

- Operationen zweier Transaktionen greifen auf die gleichen Datenobjekte zu
- Dabei überschneiden sie sich zeitlich derart, dass einzelne durchgeführte Aktualisierungen der Datenobjekte verloren gehen
- Beispiel: Verschränkte Ausführung von  $T_1, T_2$ 
  - Aktualisierung von  $X$  in  $T_1$  in Zeile 5 geht durch das Schreiben in  $T_2$  in Zeile 7 verloren
    - $X$  hat einen falschen Wert in Zeile 3



	$T_1$	$T_2$
1	read_item( $X$ )	
2	$X := X - N$	
3		read_item( $X$ )
4		$X := X + M$
5	write_item( $X$ )	
6	read_item( $Y$ )	
7		write_item( $X$ )
8	$Y := Y + N$	
9	write_item( $Y$ )	

## DIRTY READ



- Durch eine Transaktion, die später abgebrochen wird, findet zunächst eine Aktualisierung eines Datenobjekts statt
- Eine andere Transaktion liest das modifizierte Datenobjekt, bevor die bereits durchgeföhrte Aktualisierung in  $T_1$  verworfen wird
- Beispiel: Verschränkte Ausführung von  $T_1, T_2$ 
  - $T_1$  schlägt fehl  $\rightarrow$  Wert von  $X$  muss auf den alten Wert zurückgesetzt werden
  - Inzwischen hat aber  $T_2$  den von  $T_1$  geschriebenen Wert gelesen und verrechnet

... Wie spielt  
ACID hier rein?

Zeit ↓

	$T_1$	$T_2$
1	read_item( $X$ )	
2	$X := X - N$	
3	write_item( $X$ )	
4		read_item( $X$ )
5		$X := X + M$
6		write_item( $X$ )
7		read_item( $Y$ )
8		abort
9		

## GHOST UPDATE



- Zwei nebenläufige Transaktionen
  - Eine Transaktion berechnet eine Aggregation auf einer Menge von Datenobjekten
  - Eine andere Transaktion aktualisiert einige dieser Datenobjekte
- In die Aggregation gehen u.U. Datenobjekte ein, die bereits aktualisiert wurden, und andere mit ihrem Wert vor einer Aktualisierung
- Beispiel: Verschränkte Ausführung von  $T_1, T_3$ 
  - $T_3$  liest  $X$ , nachdem  $N$  in  $T_1$  subtrahiert wurde, und  $Y$ , bevor  $N$  in  $T_1$  addiert wird
  - Ergibt ein um  $N$  falsches Resultat

...  Wie spielt  
ACID hier rein?

$T_1$	$T_3$
1	$sum := 0$
2	<code>read_item(<math>A</math>)</code>
3	$sum := sum + A$
	$\vdots$
4	<code>read_item(<math>X</math>)</code>
5	$X := X - N$
6	<code>write_item(<math>X</math>)</code>
7	<code>read_item(<math>X</math>)</code>
8	$sum := sum + X$
9	<code>read_item(<math>Y</math>)</code>
10	$sum := sum + Y$
11	<code>read_item(<math>Y</math>)</code>
12	$Y := Y + N$
13	<code>write_item(<math>Y</math>)</code>

## UNREPEATABLE READ



- Datenobjekt wird innerhalb einer Transaktion mehrfach gelesen, während eine andere Transaktion dieses Datenobjekt modifiziert
- Je nach zeitlicher Abfolge der Anweisungen in den beiden Transaktionen wird nicht der jeweils gleiche Wert wiederholt gelesen
- Ähnlicher Fehler: PHANTOM READ
  - Während eine Transaktion eine Tabelle (wiederholt) liest, fügt eine andere Transaktion neue Tupel ein oder löscht Tupel
- Beispiel: Verschränkte Ausführung von  $T'$ ,  $T''$ 
  - In Zeile 9 hat  $X$  einen um  $N$  niedrigeren Wert, als wenn der Lesebefehl vor Start von  $T''$  erfolgt wäre

$T'$	$T''$
1 $sum := 0$	
2 $read\_item(A)$	
3 $sum := sum + A$	
	⋮
4 $read\_item(X)$	
5 $sum := sum + X$	
6	$read\_item(X)$
7	⋮
8	$X := X - N$
9 $read\_item(X)$	$write\_item(X)$

## Zwischenzusammenfassung

- Verschiedene Operationen und Zustände während der Transaktionsverarbeitung
  - BEGIN\_TRANSACTION, END\_TRANSACTION, COMMIT, ABORT
  - READ\_ITEM, WRITE\_ITEM
- Probleme bei verschränkter Ausführung von Transaktionen
  - *Lost Update*: Überschreiben einer Aktualisierung bzw. Arbeiten mit einem veralteten Wert
  - *Dirty Read*: Lesen eines Wertes, der durch eine Transaktion geändert wurde, die danach abgebrochen wurde
  - *Ghost Update*: Zusammenhängende aktualisierte und nicht aktualisierte Werte werden verarbeitet
  - *Unrepeatable Read*: Werte ändern sich beim wiederholten Lesen (andere Reihenfolge führt zu anderem Ergebnis)



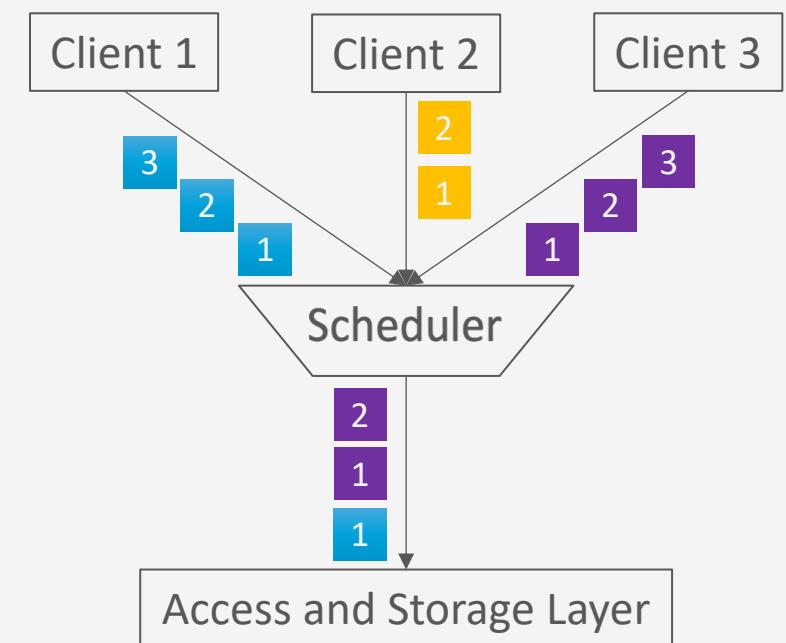
soON

# Schedules

Transaktionsverarbeitung

## Nebenläufige Ausführung

- Große Menge von Transaktionen
  - Sequentielle Ausführung vermeidet Probleme, aber kostet viel Zeit
  - Nebenläufige Ausführung bei entsprechender Hardware-Unterstützung ist schneller, kann aber zu Anomalien führen
- Anforderung an DBMS:  
Nebenläufige Ausführung und keine Anomalien
- Steuerprogramm (**Scheduler**) entscheidet über die Ausführungsreihenfolge der nebenläufigen Datenbankzugriffe
  - Ziel: Korrekte verschachtelte *Schedules* (Definition folgt)



## Schedules von Transaktionen

- Ausführungsplan / Schedule  $S$ 
  - Integrierte Abfolge der Operationen überlappend ausgeführter Transaktionen
  - Schedule  $S$  von  $n$  Transaktionen  $T_1, \dots, T_n$  ist eine Sequenz von Transaktionsoperationen mit der Eigenschaft, dass die Operationen einer Transaktion  $T_i$  in  $S$  in der gleichen Reihenfolge wie in  $T_i$  erscheinen,
    - I.e.,  $T_i = \langle p_1, \dots, p_{N'} \rangle$  und  $S = \langle o_1, \dots, o_N \rangle$  mit  $T_i = \pi_i(S)$  (Reihenfolge erhaltende Abb. von  $S$  auf  $T_i$ )
- Abkürzungen für die folgenden Folien:
  - Operation (read oder write) auf Objekt  $X$  einer Transaktion  $T_i$ :  $p_i(X)$
  - READ\_ITEM auf Objekt  $X$  einer Transaktion  $T_i$ :  $r_i(X)$
  - WRITE\_ITEM von Objekt  $X$  einer Transaktion  $T_i$ :  $w_i(X)$
  - COMMIT einer Transaktion  $T_i$ :  $c_i$
  - ABORT einer Transaktion  $T_i$ :  $a_i$

## Beispiele für Schedules

- Transaktionen
  - Reduziert auf  $w_i, r_i$
  - $T_1 = \langle r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
  - Bzw.  $T_1 = \langle r_1(X), w_1(X), r_1(Y), \textcolor{blue}{a}_1 \rangle$
  - $T_2 = \langle r_2(X), w_2(X) \rangle$
- Schedules
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$
  - $S_b = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), \textcolor{blue}{a}_1 \rangle$

$S_a$	$T_1$	$T_2$	$S_b$	$T_1$	$T_2$
	1 read_item( $X$ )			1 read_item( $X$ )	
	2 $X := X - N$			2 $X := X - N$	
	3	read_item( $X$ )		3 write_item( $X$ )	
	4	$X := X + M$		4	read_item( $X$ )
	5 write_item( $X$ )			5	$X := X + M$
	6 read_item( $Y$ )			6	write_item( $X$ )
	7	write_item( $X$ )		7 read_item( $Y$ )	
	8 $Y := Y + N$			8 abort	
	9 write_item( $Y$ )			9	

## Serielle Schedules

- Schedule  $S$  ist **seriell**, wenn die Operationen jeder in  $S$  enthaltenen Transaktion  $T$  vollständig hintereinander (streng sequenziell) ausgeführt werden
  - Für jede Transaktion  $T$  gilt, dass ihr Schritte direkt aufeinander folgen
  - Andernfalls ist ein Schedule nicht-seriell

## Serielle Schedules: Beispiele

- Schedule  $S_d$ 
  - $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$
  - $T_1|T_2$
- Schedule  $S_e$ 
  - $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
  - $T_2|T_1$

$S_d$	$T_1$	$T_2$	$S_e$	$T_1$	$T_2$
	1 read_item( $X$ )			1	read_item( $X$ )
	2 $X := X - N$			2	$X := X + M$
	3 write_item( $X$ )			3	write_item( $X$ )
	4 read_item( $Y$ )			4	read_item( $X$ )
	5 $Y := Y + N$			5	$X := X - N$
	6 write_item( $Y$ )			6	write_item( $X$ )
	7	read_item( $X$ )		7	read_item( $Y$ )
	8	$X := X + M$		8	$Y := Y + N$
	9	write_item( $X$ )		9	write_item( $Y$ )

## Korrekte Schedules

- Anomalien können nur auftreten, wenn die Schritte mehrerer Transaktionen verschränkt ausgeführt werden
  - Wenn alle Transaktionen bis zum Ende ausgeführt werden (keine Nebenläufigkeit), treten keine Anomalien auf
- Jede serielle Ausführung ist **korrekt**
- Verzicht auf nebenläufige Ausführung nicht praktikabel
  - Wartezeiten auf Platten
- Jede verschränkte Ausführung, die einen gleichen Zustand, wie eine serielle Ausführung erzeugt, ist **korrekt**

## Korrekte Schedules

- Manchmal kann man einfach Teilschritte aus verschiedenen Transaktionen in einem Plan umordnen
  - Nicht jedoch die Teilschritte innerhalb einer einzelnen Transaktion (da sonst eventuell anderes Ergebnis)
- Jeder Plan  $S'$ , der durch legale Umordnung von  $S$  generiert werden kann, heißt äquivalent zu  $S$
- Falls Umordnung nicht möglich (weil Ergebnis möglicherweise verfälscht) → Konflikt

## Konflikte

- Zwei Operationen  $p_i(X), p_j(X)$  in einem Schedule stehen in **Konflikt** (sind **konfliktär**), wenn alle folgenden Bedingungen erfüllt sind:
    - Sie gehören zu unterschiedlichen Transaktionen ( $i \neq j$ )
    - Sie greifen auf das gleiche Datenobjekt zu ( $X$ )
    - Mindestens eine Operation schreibt  $X$  ( $p_i(X) = w_i(X) \vee p_j(X) = w_j(X)$ )
      - Konfliktmatrix siehe rechts unten
- Reihenfolge von  $p_i(X), p_j(X)$  ist relevant
- Nicht in Konflikt stehende Operationen können auch parallel ausgeführt werden
    - Solche Schedules heißen **partiell geordnet**

Konfliktmatrix

	$r_j(X)$	$w_j(X)$
$r_i(X)$		X
$w_i(X)$	X	X

## Serialisierbarkeit

- Schedule  $S$  mit  $n$  Transaktionen ist **serialisierbar**, wenn er zu einem seriellen Schedule  $S'$  äquivalent ist, d.h., den gleichen DB-Zustand erreicht
  - Reihenfolge der konfliktären Operationen ist gleich
  - Ausführung eines serialisierbaren Schedules  $S$  ist **korrekt**
  - $S$  muss nicht seriell sein

# Konflikte, Äquivalenz, Serialisierbarkeit, Korrektheit: Beispiele

- Schedule  $S_d$ 
  - $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$
  - Konflikte:  $r_1(X), w_2(X)$ ,  
 $w_1(X), r_2(X)$ ,  $w_1(X), w_2(X)$
- Schedule  $S_c$ 
  - $S_c = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), w_1(Y) \rangle$
  - Konflikte: Wie oben
    - Äquivalent
    - Serialisierbar
  - Damit:  $S_c$  korrekt

$S_c$	$T_1$	$T_2$
	1 read_item( $X$ )	
	2 $X := X - N$	
	3 write_item( $X$ )	
	4	read_item( $X$ )
	5	$X := X + M$
	6	write_item( $X$ )
	7 read_item( $Y$ )	
	8 $Y := Y + N$	
	9 write_item( $Y$ )	

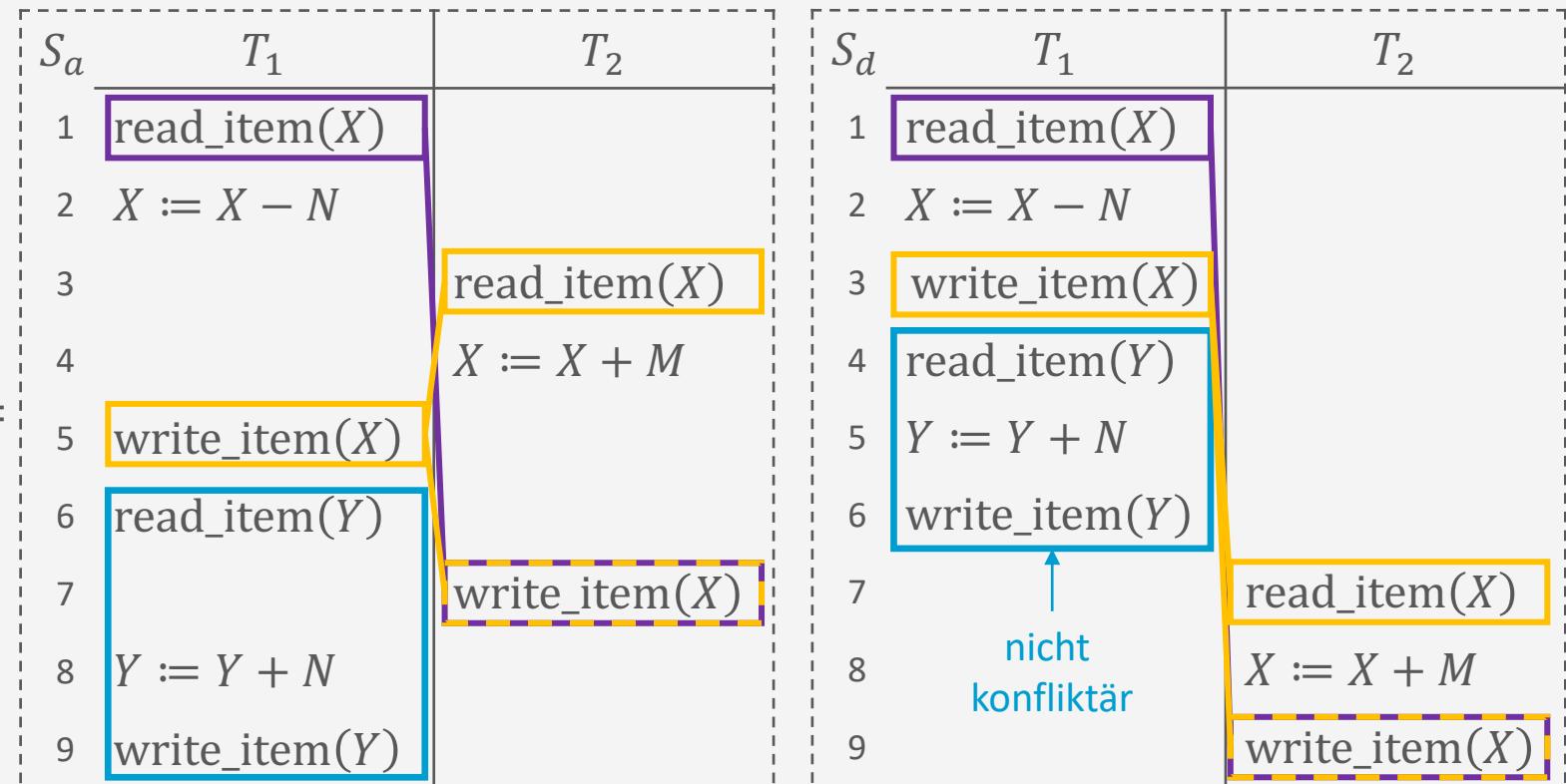
  

$S_d$	$T_1$	$T_2$
	1 read_item( $X$ )	
	2 $X := X - N$	
	3 write_item( $X$ )	
	4 read_item( $Y$ )	
	5 $Y := Y + N$	
	6 write_item( $Y$ )	
	7	read_item( $X$ )
	8	$X := X + M$
	9	write_item( $X$ )

↑ nicht konfliktär

# Konflikte, Äquivalenz, Serialisierbarkeit, Korrektheit: Beispiele

- Schedule  $S_a$ 
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$
  - Nicht äquivalent mit  $S_d$ 
    - Konfliktäre Operationen  $r_2(X), w_1(X)$  in  $S_a$  in anderer Reihenfolge in  $S_d$
  - Auch nicht äquivalent mit  $S_e = T_2 | T_1$ 
    - $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
  - Damit  $S_a$  nicht serialisierbar
  - Damit  $S_a$  nicht korrekt



## Korrektheitsprüfung mittels Serialisierungsgraphen

- **Serialisierungsgraph** (oder Konfliktgraph)  $SG$  für ein Schedule  $S$  ist ein gerichteter Graph  $G = (N, E)$ , wobei
  - $N = \{T_1, \dots, T_n\}$  Menge von Knoten, ein Knoten für jede Transaktion in  $S$
  - $E = \{e_1, \dots, e_m\}$  Menge gerichteter Kanten
    - Jede Kante  $e$  im Graphen hat die Form  $(T_i \rightarrow T_j)$ ,  $1 \leq i \leq n, 1 \leq j \leq n$ , wobei  $T_i$  der Start- und  $T_j$  der Endknoten von  $e$  ist
- Kante  $T_i \rightarrow T_j$  wird erzeugt, wenn in  $S$  eine Operation von  $T_i$  vor einer mit ihr in Konflikt stehenden Operation in  $T_j$  vorkommt
  - I.e.,  $p_i(X)$  und  $p_j(X)$  stehen in Konflikt und  $p_i(X)$  erscheint vor  $p_j(X)$  in  $S$

### Serialisierbarkeitstheorem

Schedule  $S$  ist serialisierbar, wenn der zugehörige Serialisierungsgraph keine Zyklen aufweist.

## Korrektheitsprüfung

- Eingabe: Schedule  $S$
- Ausgabe:  $S$  serialisierbar ja/nein
- Vorgehen
  1. Baue einen Serialisierungsgraph  $SG$  für  $S$ 
    - a. Erzeuge für jede Transaktion  $T_i$ , die in  $S$  auftritt, einen Knoten  $T_i$  in  $SG$
    - b. Für jeden Fall in  $S$ , bei dem erst  $T_i$  eine Operation  $p_i(X)$  ausführt und dann  $T_j$  eine Operation  $p_j(X)$  ausführt, wobei mindestens ein  $p(X)$  eine Schreiboperation ist, erzeuge eine Kante  $(T_i \rightarrow T_j)$  in  $SG$ 
      - Drei Fälle: (i)  $w_i(X), r_j(X)$ , (ii)  $r_i(X), w_j(X)$ , (iii)  $w_i(X), w_j(X)$
  2. Prüfe, ob  $SG$  keinen Zyklus enthält
    - a. Wenn kein Zyklus vorhanden:  $S$  ist serialisierbar
    - b. Sonst:  $S$  ist nicht serialisierbar

## Serialisierungsgraphen: Beispiele

- Serieller Schedule  $S_d$
- $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$



- Serieller Schedule  $S_e$
- $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$



- Beide Schedules serialisierbar

$S_d$	$T_1$	$T_2$
1	read_item( $X$ )	
2	$X := X - N$	
3	write_item( $X$ )	
4	read_item( $Y$ )	
5	$Y := Y + N$	
6	write_item( $Y$ )	
7		read_item( $X$ )
8		$X := X + M$
9		write_item( $X$ )

$S_e$	$T_1$	$T_2$
1		read_item( $X$ )
2		$X := X + M$
3		write_item( $X$ )
4	read_item( $X$ )	
5	$X := X - N$	
6	write_item( $X$ )	
7	read_item( $Y$ )	
8	$Y := Y + N$	
9	write_item( $Y$ )	

## Serialisierungsgraphen: Beispiele

- Schedule  $S_a$ 
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$



- Schedule  $S_c$ 
  - $S_c = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), w_1(Y) \rangle$



- $S_a$  nicht serialisierbar (Zyklus),  $S_c$  ja

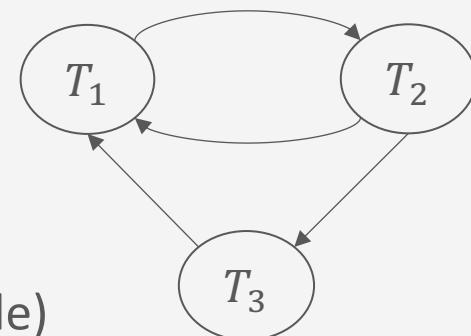
	$T_1$	$T_2$
$S_a$	1 read_item( $X$ ) 2 $X := X - N$ 3 4 5 write_item( $X$ ) 6 read_item( $Y$ ) 7 8 $Y := Y + N$ 9 write_item( $Y$ )	read_item( $X$ ) $X := X + M$ write_item( $X$ )
$S_c$	1 read_item( $X$ ) 2 $X := X - N$ 3 write_item( $X$ ) 4 5 6 7 read_item( $Y$ ) 8 $Y := Y + N$ 9 write_item( $Y$ )	read_item( $X$ ) $X := X + M$ write_item( $X$ )

## Serialisierungsgraphen: Beispiele

- Transaktionen  $T_1, T_2, T_3$

$T_1$	$T_2$	$T_3$
read_item( $X$ )	read_item( $Z$ )	read_item( $Y$ )
write_item( $X$ )	read_item( $Y$ )	read_item( $Z$ )
read_item( $Y$ )	write_item( $Y$ )	write_item( $Y$ )
write_item( $Y$ )	read_item( $X$ )	write_item( $Z$ )
		write_item( $X$ )

- Schedule  $S_g$  rechts
  - Serialisierungsgraph:
    - Zyklen im Graph
    - Nicht serialisierbar (kein äquivalenter serieller Schedule)



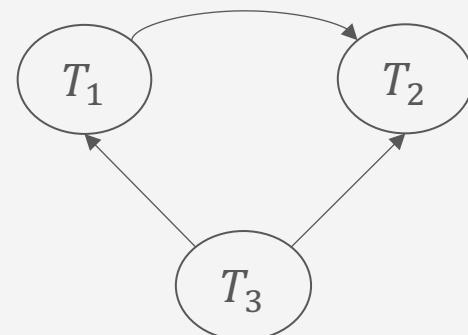
$S_g$	$T_1$	$T_2$	$T_3$	Transaktionen
1		read_item( $Z$ )		
2		read_item( $Y$ )		
3		write_item( $Y$ )		
4				
5				
6	read_item( $X$ )			
7		write_item( $X$ )		
8				
9				
10				
11		read_item( $Y$ )		
12		write_item( $Y$ )		
13			write_item( $X$ )	

## Serialisierungsgraphen: Beispiele

- Transaktionen  $T_1, T_2, T_3$

$T_1$	$T_2$	$T_3$
read_item( $X$ )	read_item( $Z$ )	read_item( $Y$ )
write_item( $X$ )	read_item( $Y$ )	read_item( $Z$ )
read_item( $Y$ )	write_item( $Y$ )	write_item( $Y$ )
write_item( $Y$ )	read_item( $X$ )	write_item( $Z$ )
		write_item( $X$ )

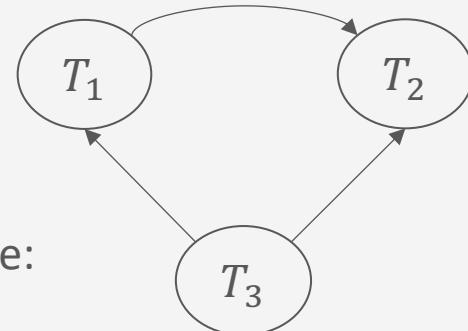
- Schedule  $S_h$  rechts
  - Serialisierungsgraph:
    - Graph zyklenfrei
    - Serialisierbar



$S_h$	$T_1$	$T_2$	$T_3$	Transaktionen
1				read_item( $Y$ )
2				read_item( $Z$ )
3		read_item( $X$ )		
4		write_item( $X$ )		
5				write_item( $Y$ )
6				write_item( $Z$ )
7				read_item( $Z$ )
8		read_item( $Y$ )		
9		write_item( $Y$ )		
10			read_item( $Y$ )	
11			write_item( $Y$ )	
12			read_item( $X$ )	
13			write_item( $X$ )	

## Äquivalente serielle Schedules

- Gegeben ein zyklenfreier Serialisierungsgraphen  $SG$  für ein Schedule  $S$
- Äquivalenter serieller Schedule durch topologische Sortierung von  $SG$ 
  - *Topologische Sortierung*: Sequenz aller Knoten im Graph, wobei Elternknoten vor Kindknoten in der Sequenz erscheinen müssen
  - Beispiel vorherige Folie
    - Serialisierungsgraph:
    - Graph zyklenfrei
    - Serialisierbar
    - Äquivalenter serieller Schedule:  
 $T_3|T_1|T_2$



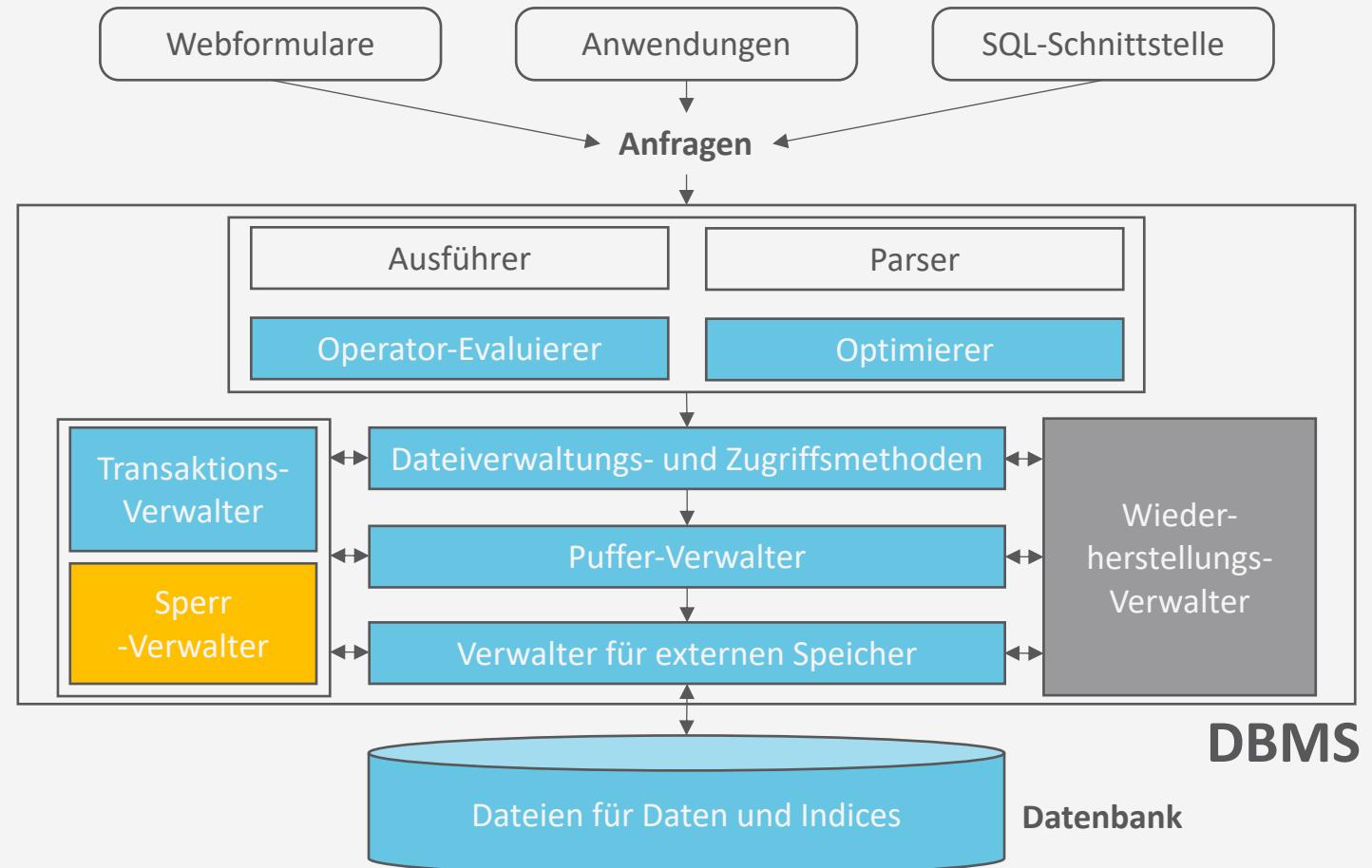
$S_h$	$T_1$	$T_2$	$T_3$
1			read_item(Y)
2			read_item(Z)
3	read_item(X)		
4	write_item(X)		
5			write_item(Y)
6			write_item(Z)
7			read_item(Z)
8		read_item(Y)	
9		write_item(Y)	
10			read_item(Y)
11			write_item(Y)
12			read_item(X)
13			write_item(X)

## Zwischenzusammenfassung

- Schedules: Sequenz von Operationen aus einer Menge von Transaktionen
  - Ursprüngliche Reihenfolge der Operationen aus den Transaktionen beizubehalten
- Serielle Schedules: strikt sequentielle Anordnung der Transaktionen
  - Serielle Schedules sind korrekt
- Konflikte: Wenn Schreiboperationen auf demselben Datenobjekt involviert sind
- Serialisierbarkeit
  - Serialisierbarer Schedule äquivalent zu seriellem Schedule → Korrekter Schedule
- Serialisierungsgraphen
  - Knoten für Transaktionen; Kanten bei konfliktären Operationen
  - Test auf Serialisierbarkeit / Korrektheit: Zyklenfreier Serialisierungsgraph
  - Äquivalenter serieller Schedule mittels topologischer Sortierung

# Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
    - Serielle, serialisierbare Schedules
    - Konflikte, Serialisierungsgraph
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



# Überblick: 7. Transaktionen

## A. *Transaktionsverarbeitung*

- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

## B. *Sperrverwaltung*

- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

## C. *Wiederherstellungsverwaltung*

- Fehlersituationen
- Logging
- Recovery

## Sperr-Verwaltung

- Transaktionen müssen **Sperren** anfragen für Datenobjekte, auf die sie zugreifen wollen
- Falls eine Sperre nicht zugeteilt wird (z.B. weil eine andere Transaktion  $T'$  die Sperre schon hält), wird die anfragende Transaktion  $T$  **blockiert**
  - Sperrverwalter setzt die Ausführung von Aktionen einer blockierten Transaktion  $T$  aus
- Sobald  $T'$  die Sperre freigibt, kann sie an  $T$  vergeben werden (oder an eine andere Transaktion, die darauf wartet)
  - Transaktion, die eine Sperre erhält, wird fortgesetzt
- Sperren regeln die relative Ordnung der Einzeloperationen verschiedener Transaktionen
- Ziel: Automatisierte Generierung von serialisierbaren Schedules

## Implementierung eines Sperrverwalters

- Ein Sperrverwalter muss drei Aufgaben effektiv erledigen:
  1. Prüfen, welche **Sperren für eine Ressource** gehalten werden (um eine Sperranforderung zu behandeln)
  2. Bei Sperr-Rückgabe müssen die **Transaktionen**, die die Sperre haben wollen, schnell **identifizierbar** sein
  3. Wenn eine Transaktion beendet wird, müssen alle von der Transaktion angeforderten und gehaltenen **Sperren zurückgegeben** werden
- **Sperrtabelle:** Datenstruktur zur Speicherung der Lock-Information zu jedem Objekt
  - Meist als Hash-Tabelle organisiert, um effizient auf Lock-Informationen zugreifen zu können



# Sperren und Sperrprotokolle

Sperrverwaltung

## Protokolle und Sperren

- Protokolle mit Sperren:
  - Binäre Sperren (einfach aber restriktiv)
  - Mehrfachmodus- bzw. gemeinsame/exklusive Sperren (praxisrelevant)
  - Zwei-Phasen-Sperrprotokoll (praxisrelevant)
  - Multiversionsprotokolle (Verbesserung der Performanz)
  - Multiversionsprotokolle mit Zeitstempelung (Verbesserung der Performanz)
  - Zertifizierungssperren (Verbesserung der Performanz)
- Protokolle ohne Sperren:
  - Zeitstempelbasierte Transaktionsverarbeitung (praxisrelevant)

## Binäre Sperren

- Mit jedem Datenobjekt  $X$  ist eine Sperre assoziiert
  - Sperre kann zwei Zustände annehmen: „gesperrt“ oder „entsperrt“
    -   $\text{Lock}(X) = 1$ : Objekt ist gesperrt
    -   $\text{Lock}(X) = 0$ : Objekt ist entsperrt
  - Operationen:
    - $\text{lock\_item}(X)$ : sperrt das Objekt
    - $\text{unlock\_item}(X)$ : entsperrt das Objekt
- Wenn eine Transaktion  $T_1$  ein Objekt  $X$  gesperrt hat, kann eine Operation  $T_2$  auf  $X$  nicht zugreifen (bzw. selber sperren)
  - $T_2$  muss warten, bis  $X$  durch  $T_1$  wieder entsperrt wurde
- Implementierung
  - Eintrag in Sperrtabelle: ( $\text{DatenobjektID}, \text{LOCKZustand}, \text{TransaktionsID}$ )

## Binäre Sperren

- Protokoll mit binären Sperren nach folgenden Regeln für jede Transaktion  $T$ :
  1. Vor  $\text{read\_item}(X)$  und  $\text{write\_item}(X)$ :  $\text{lock\_item}(X)$
  2. Nach allen  $\text{read\_item}(X)$  und  $\text{write\_item}(X)$ :  $\text{unlock\_item}(X)$
  3. Kein  $\text{lock\_item}(X)$  durch  $T$ , wenn  $T$  schon eine Sperre auf  $X$  hat
  4. Nur dann  $\text{unlock\_item}(X)$ , wenn  $T$  auch eine Sperre auf  $X$  hat
- Problem:
  - Sehr restriktiv
    - Z.B. keine parallelen Leseoperationen möglich

## Sperren mit Mehrfachmodus

- Mit jedem Datenobjekt  $X$  ist eine Sperre assoziiert mit drei möglichen Zuständen:
  -   $Lock(X) = S$  (Lesesperre)
    - Auf  $X$  wird lesend zugegriffen (*shared*) → Als **Sperrmodus S** bezeichnet
    - Weiterer Lesezugriff problemlos möglich, Schreibzugriff potenziell problematisch
  -   $Lock(X) = X$  (Schreibsperre)
    - Auf  $X$  wird schreibend zugegriffen (*exklusive*) → Als **Sperrmodus X** bezeichnet
    - Auf  $X$  kann durch andere Transaktion nicht zugegriffen werden
  -   $Lock(X) = 0$  (entsperrt) : Auf  $X$  wird nicht zugegriffen
    - Kompatibilitätsmatrix (rechts)
    - Zustand: Gibt es eine Sperre?
    - Anforderung: Was für eine Sperre soll gesetzt werden?
- Operationen: `read_lock( $X$ )`, `write_lock( $X$ )` und `unlock( $X$ )`

Anforderung	Zustand	0	S	X
S		✓	✓	-
X		✓	-	-

## Sperren mit Mehrfachmodus

- Mögliche Implementierung:
  - Einträge in der Sperrtabelle:  
 $(\text{DatenobjektID}, \text{LOCKZustand}, \#\text{Zugriffsoperationen}, [\text{TransaktionsID}, \dots])$
  - Idee: Zählen der lesenden Transaktionen ( $\#\text{Zugriffsoperationen}$ )
  - Bei schreibenden Transaktionen ( $\text{lock}(X) = X$ ) ist  $\#\text{Zugriffsoperationen} = 1$
- Regeln für Transaktion  $T$ :
  1.  $\text{read\_lock}(X)$  oder  $\text{write\_lock}(X)$  anstoßen vor irgendwelchen  $\text{read\_item}(X)$
  2.  $\text{write\_lock}(X)$  vor irgendwelchen  $\text{write\_item}(X)$
  3.  $\text{unlock}(X)$  nach allen  $\text{read\_item}(X)$  und  $\text{write\_item}(X)$
  4. Kein  $\text{read\_lock}(X)$ , falls irgendeine Sperre auf  $X$  durch  $T$  besteht
  5. Kein  $\text{write\_lock}(X)$ , falls irgendeine Sperre auf  $X$  durch  $T$  besteht
  6. Nur dann  $\text{unlock}(X)$ , wenn eine Sperre auf  $X$  durch  $T$  besteht

Anforderung	Zustand	0	S	X
S		✓	✓	-
X		✓	-	-

## Sperren mit Mehrfachmodus - Sperrenänderung

- Für weniger restriktive Lock-Mechanismen Änderung von Regeln 4 und 5 → **Sperrenänderung**
- Damit kann eine Schreibsperre zur Lesesperre gelockert oder eine Lesesperre zur Schreibsperre verschärft werden:
  1. `read_lock(X)` oder `write_lock(X)` anstoßen vor irgendwelchen `read_item(X)`
  2. `write_lock(X)` vor irgendwelchen `write_item(X)`
  3. `unlock(X)` nach allen `read_item(X)` oder `write_item(X)`
  4. Kein `read_lock(X)`, falls **eine Lesesperre** auf  $X$  durch  $T$  besteht
  5. Kein `write_lock(X)`, falls **eine Schreibsperre** auf  $X$  durch  $T$  besteht
  6. Nur dann `unlock(X)`, wenn eine Sperre auf  $X$  durch  $T$  besteht

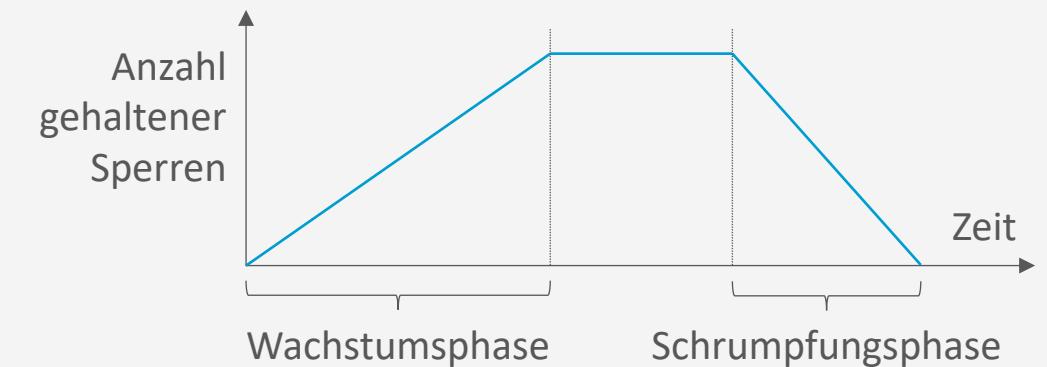
## Sperren: Bewertung

- Binäre und Mehrfach-Sperren garantieren selbst noch keine Serialisierbarkeit
  - Beispiel: Schedule  $S$ 
    - Anfangswerte:
      - $X = 20, Y = 30$
    - Resultat von  $S$ :
      - $X = 50, Y = 50$
    - Resultat des seriellen Plans  $T_1 | T_2$ :
      - $X = 50, Y = 80$
    - Resultat des seriellen Plans  $T_2 | T_1$ :
      - $X = 70, Y = 50$
- Besseres Protokoll gesucht!

$S$	$T_1$	$T_2$
1	read_lock( $Y$ )	
2	read_item( $Y$ )	
3	unlock( $Y$ )	
4		read_lock( $X$ )
5		read_item( $X$ )
6		unlock( $X$ )
7		write_lock( $Y$ )
8		read_item( $Y$ )
9		$Y := X + Y$
10		write_item( $Y$ )
11		unlock( $Y$ )
12		write_lock( $X$ )
13		read_item( $X$ )
14		$X := X + Y$
15		write_item( $X$ )
16		unlock( $X$ )

## Zwei-Phasen-Sperrprotokoll (*Two-Phase Locking, 2PL*)

- Idee: Sperroperationen aller Objekte werden vor der ersten Entsperroperation ausgeführt
- Damit ergeben sich zwei Phasen für eine Transaktion  $T$ :
  1. **Wachstumsphasen:**
    - $T$  sammelt immer mehr Sperren auf Objekte
  2. **Schrumpfungsphase:**
    - $T$  gibt immer mehr Sperren auf Objekte frei
    - Bei Sperrenänderungen:
      - Verschärfungen nur in Wachstumsphase (Lesesperrre zu Schreibsperrre)
      - Lockerungen nur in Schrumpfungsphase (Schreibsperrre zu Lesesperrre)
- Schedules, die dem Zwei-Phasen-Sperrprotokoll folgen, sind **serialisierbar**
- Schreiboperationen sind durch exklusive Sperren abgesichert



## Beispiel

- Nicht Zwei-Phasen-Sperrprotokoll
  - $T_1$ : `write_lock(X)` nach `unlock(Y)`
  - $T_2$ : `write_lock(Y)` nach `unlock(X)`

$T_1$	$T_2$
<code>read_lock(Y)</code>	<code>read_lock(X)</code>
<code>read_item(Y)</code>	<code>read_item(X)</code>
<code>unlock(Y)</code>	<code>unlock(X)</code>
<code>write_lock(X)</code>	<code>write_lock(Y)</code>
<code>read_item(X)</code>	<code>read_item(Y)</code>
$X := X + Y$	$Y := X + Y$
<code>write_item(X)</code>	<code>write_item(Y)</code>
<code>unlock(X)</code>	<code>unlock(Y)</code>

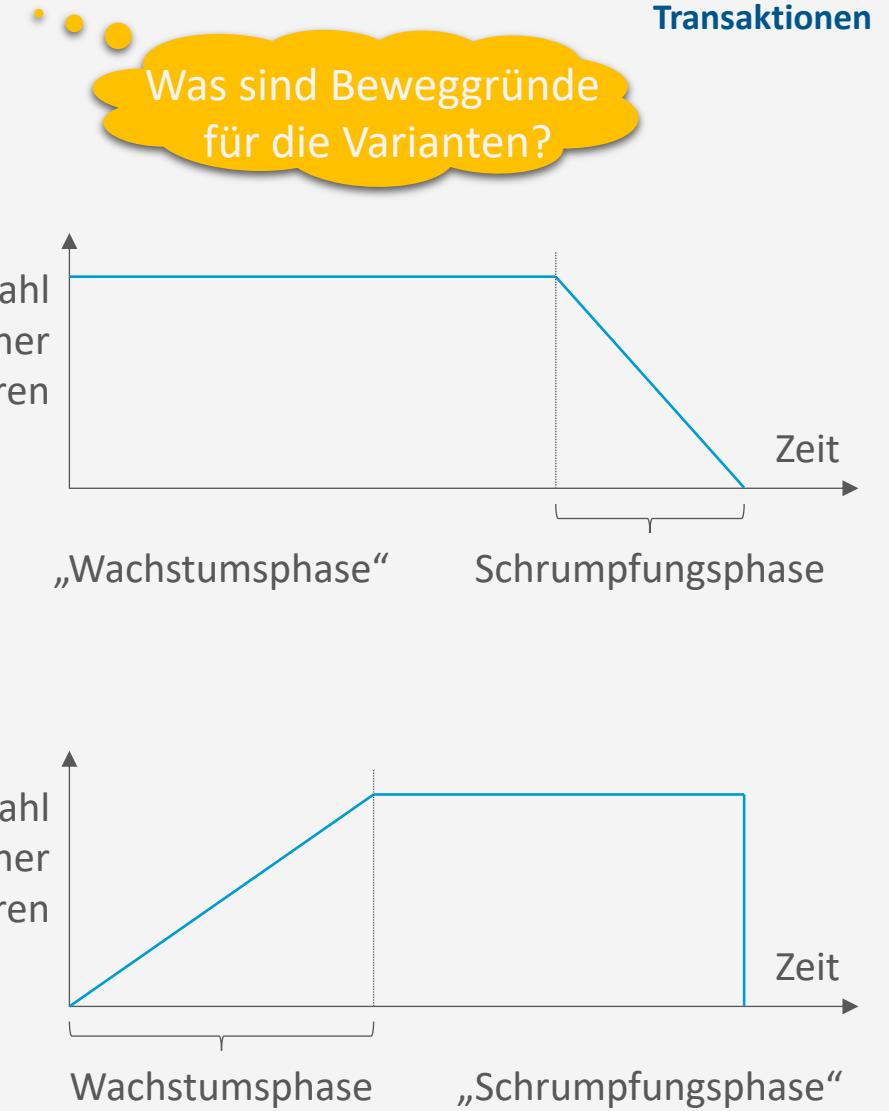
$T'_2$  wartet auf  
`unlock(Y)` von  $T'_1$   
und  
 $T'_1$  wartet auf  
`unlock(X)` von  $T'_2$

- Zwei-Phasen-Sperrprotokoll
  - Problem: Kann zu **Deadlock** führen
    - Beide Transaktionen aktiv + warten auf Sperre

$T'_1$	$T'_2$
<code>read_lock(Y)</code>	<code>read_lock(X)</code>
<code>read_item(Y)</code>	<code>read_item(X)</code>
<code>write_lock(X)</code>	<code>write_lock(Y)</code>
<code>unlock(Y)</code>	<code>unlock(X)</code>
<code>read_item(X)</code>	<code>read_item(Y)</code>
$X := X + Y$	$Y := X + Y$
<code>write_item(X)</code>	<code>write_item(Y)</code>
<code>unlock(X)</code>	<code>unlock(Y)</code>

## Varianten des Zwei-Phasen-Sperrprotokolls

- **Konservatives 2PL:** Transaktion  $T$  sperrt erst alle Objekte, bevor sie mit der Verarbeitung beginnt
  - Keine Deadlocks, aber alle benötigten Sperren müssen vorab bekannt und deklariert sein
- **Striktes 2PL:** Transaktion  $T$  hebt *Schreibsperren* erst auf, wenn  $T$  beendet wurde
  - In der Praxis recht verbreitet, garantiert aber nicht Deadlock-Freiheit
- **Rigoroses 2PL:**  $T$  hebt *Lese- und Schreibsperren* erst auf, wenn  $T$  beendet wurde
  - Einfacher umzusetzen als die strikte Variante, verhindert aber auch nicht Deadlocks



## Beispiel

- Transaktionen  $T_1, T_2$

$T_1$	$T_2$
read_lock( $X$ )	write_lock( $Y$ )
read_item( $X$ )	read_item( $Y$ )
read_lock( $Y$ )	$Y := Y - 100$
read_item( $Y$ )	write_lock( $Z$ )
read_lock( $Z$ )	read_item( $Z$ )
read_item( $Z$ )	$Z := Z + 100$
$S := X + Y + Z$	write_item( $Y$ )
unlock( $X$ )	write_item( $Z$ )
unlock( $Y$ )	unlock( $Y$ )
unlock( $Z$ )	unlock( $Z$ )

$S$	$T_1$	$T_2$	$X$	$Y$	$Z$	
1	begin_transaction					
2	read_lock( $X$ )					
3	read_item( $X$ )					
4						
5		begin_transaction				
6		write_lock( $Y$ )				
7	read_lock( $Y$ )	read_item( $Y$ )				
8						
9		$Y := Y - 100$				
10		write_lock( $Z$ )				
11		read_item( $Z$ )				
12		$Z := Z + 100$				
13		write_item( $Y$ )				
14		write_item( $Z$ )				
15		end_transaction				
16		unlock( $Y$ )				
17		read_item( $Y$ )				
18		read_lock( $Z$ )				
19		unlock( $Z$ )				
20		commit				
21		$S := X + Y + Z$				
22		end_transaction				
23		unlock( $X$ )				
24		unlock( $Y$ )				
25		unlock( $Z$ )				
26		commit				

Transaktionen

Deadlocks?

Konservativ?  
Strikt? Rigoros?

2PL?

## Probleme bei der Verwendung von Sperren

- Deadlocks können bei der Verwendung von Sperren entstehen
- **Deadlock** (Verklemmung):
  - Transaktion wartet auf ein Objekt, das eine andere Transaktion gesperrt hat – und umgekehrt
  - Kann auch zwischen mehr als zwei Transaktionen auftreten
  - Lösungsansätze folgen, können aber führen zu:
- **Starvation** (Verhungern):
  - Eine Transaktion wird über längere Zeit nicht abgearbeitet, da andere Transaktionen vorgezogen werden
  - Kompensationsmechanismen folgen

## Deadlock

- **Deadlock** liegt vor, wenn jede Transaktion  $T$  einer Menge von zwei oder mehr Transaktionen  $\mathbf{T}$  auf ein Objekt wartet, das von einer anderen Transaktion  $T' \in \mathbf{T}, T' \neq T$ , gesperrt wurde
- Beispiel:
  - $T_1$  und  $T_2$  sperren wechselseitig Objekte  $X$  und  $Y$
  - Behandlung von Deadlocks
    - Vermeidung
    - Erkennung und Auflösung

$S$	$T_1$	$T_2$
1	read_lock( $Y$ )	
2	read_item( $Y$ )	
3		read_lock( $X$ )
4		read_item( $X$ )
5	write_lock( $X$ )	
6		write_lock( $Y$ )

## Vermeidung von Deadlocks

- Konservatives 2PL
  - Eine Transaktion muss alle Objekte sperren, bevor sie ausgeführt wird
  - Ansonsten wartet sie, bis die gewünschten Objekte zugreifbar sind
  - Ist in der Praxis jedoch meist nicht umsetzbar
- **Transaktionszeitstempel  $TS(T)$** 
  - Eindeutiger Identifikator für eine Transaktion, der als Zeitstempel (*timestamp*) bezeichnet wird
    - In der Regel der Start-Zeitpunkt (Ablesen der internen Systemuhr) von  $T$
  - Wird Transaktion  $T'$  vor  $T''$  gestartet, so gilt  $TS(T') < TS(T'')$ 
    - $T'$  ist die ältere,  $T''$  die jüngere Transaktion
  - Darauf basierende Verfahren:
    - Wait/Die
    - Wound/Wait

## Vermeidung von Deadlocks

- Transaktion  $T'$  versucht Objekt  $X$  zu sperren,  $X$  durch andere Transaktion  $T$  schon gesperrt
  - Ältere Transaktionen werden bevorzugt
- **Wait/Die** 
  - Wenn  $TS(T') < TS(T)$ :  $T'$  ist älter als  $T$  und wartet
  - Wenn  $TS(T') \geq TS(T)$ :  $T'$  ist jünger als  $T$  und stirbt
    - $T'$  bricht sich selbst ab (abort)
    - $T'$  startet später mit gleichem Zeitstempel  $TS(T')$  wieder (wird also älter sein, weniger häufig sterben)
  - $T$  behält in beiden Fällen seine Sperre
- **Wound/Wait:** 
  - Wenn  $TS(T') < TS(T)$ :  $T'$  ist älter als  $T$  und verwundet/tötet  $T$ 
    - $T$  wird abgebrochen (abort)
    - $T$  startet später mit dem gleichen Zeitstempel  $TS(T)$  wieder (wird also älter sein, weniger häufig verwundet)
  - Wenn  $TS(T') \geq TS(T)$ :  $T'$  ist jünger als  $T$  und wartet
  - $T$  behält seine Sperre nur, wenn  $T$  älter ist

## Vermeidung von Deadlocks

- Weitere Ansätze:
  - No Waiting (NW)
    - Wenn die Transaktion keine Sperre bekommt, wird sie sofort abgebrochen
    - Viele unnötige Abbrüche und Neustarts
  - Cautious Waiting (CW)
    - $T'$  versucht Sperre zu bekommen,  $T$  hat sie
      - Wenn  $T$  nicht blockiert ist, wird  $T'$  blockiert und wartet auf  $T$
      - Sonst ( $T$  ist blockiert):  $T'$  wird abgebrochen (könnte länger dauern)
- Problem aller bisherigen Ansätze:
  - Sind zwar Deadlock frei
  - Erzeugen aber u.U. unnötige Abbrüche und Neustarts von Transaktionen, die nie einen Deadlock verursacht hätten → schlechtere Performanz

## Erkennung von Deadlocks

- Optimistisches Verfahren: statt Vermeidung **nachträgliche Erkennung und Auflösung**
  - Gut, wenn wenig Deadlocks zu erwarten sind
- Grundideen:
  - **(Physische) Zeitbeschränkungen**
    - Wartet eine Transaktion  $T_i$  länger als eine Zeitbeschränkung  $t$  vorgibt:
      - Dann ist die Annahme: Transaktion ist in Deadlock-Situation
      - $T_i$  wird abgebrochen
      - Vorteil: kann einfach geprüft werden
      - Nachteil: evtl. unnötige Transaktionsabbrüche
        - Schwierig Zeitbeschränkung richtig zu wählen:  
zu kurz → zu viele möglicherweise vermeidbare Abbrüche; zu lang → zu lange warten
    - **(Logischer) Wartegraph**
      - Nächste Folie...

## Erkennung von Deadlocks: Wartegraph

- Wartegraph: Gerichteter Graph
  - Enthält für jede aktive Transaktion  $T_i$  einen Knoten
  - Wenn  $T_i$  auf eine Sperre von  $T_j$  wartet:
    - Füge Kante  $(T_i \rightarrow T_j)$  in den Graphen ein (Kante bei Freigabe löschen)
    - Weist der Wartegraph Zyklen auf, so liegt ein Deadlock vor
- Beispiel
  - $T_1, T_2$  starten (aktiv)  $\rightarrow$  jeweils Knoten einfügen
  - Zeile 1 + 3: Objekte nicht gesperrt, alles ok
  - Zeile 5:  $T_1$  fordert Sperre für  $X$  an
    - $X$  gesperrt von  $T_2$ : Kante  $(T_1 \rightarrow T_2)$   $\rightarrow$  Kein Zyklus, alles ok
  - Zeile 6:  $T_2$  fordert Sperre für  $Y$  an
    - $Y$  gesperrt von  $T_1$ : Kante  $(T_2 \rightarrow T_1)$   $\rightarrow$  Zyklus



S	$T_1$	$T_2$
1	read_lock( $Y$ )	
2	read_item( $Y$ )	
3		read_lock( $X$ )
4		read_item( $X$ )
5		write_lock( $X$ )
6		write_lock( $Y$ )

## Behandlung von Deadlock - Opferauswahl

- Deadlock erkannt:
  - Transaktion  $T_i$  bestimmen, die abgebrochen werden soll, um Zyklus aufzulösen
- Folgende Heuristiken denkbar:
  - Wähle möglichst keine Transaktion, die bereits lange läuft
    - Bricht vor allem junge Transaktionen ab
    - Idee: Alte Transaktionen haben eine Chance endlich durchzulaufen
  - Wähle möglichst keine Transaktion, die bereits viele Aktualisierungen der DB durchgeführt hat
    - Idee: Nicht die ganze Arbeit verwerfen und alle Änderungen wieder rückgängig machen müssen
- **Problem** bei Wartegraphen-Ansatz:
  - Wann soll auf Existenz von Zyklen geprüft werden?
    - Zu oft → Kostet Zeit
    - Zu selten → Deadlocks können lange bestehen

## Starvation

- **Starvation:** Transaktion wird über längere Zeit nicht abgearbeitet, da andere Transaktionen vorgezogen werden
  - Als Folge der Deadlock-Vermeidung oder -auflösung
- Kompensationsmechanismen:
  - **First-Come-First-Served**
    - Abarbeitung in Reihenfolge
  - **Prioritäten (vergeben/anpassen)**
    - Durch Transaktionsverwaltung abgebrochene Transaktion erhält höhere Priorität, wird deshalb bei der Neuausführung mit geringerer Wahrscheinlichkeit als Opfer gewählt
  - Wait/Die und Wound/Wait vermeiden Verhungern
    - Ältere Transaktionen werden bevorzugt
    - Abgebrochene Transaktionen behalten ihre ID → Irgendwann ist jede Transaktion alt genug

# Weitere Verfahren zur Mehrbenutzerkontrolle

Sperrverwaltung

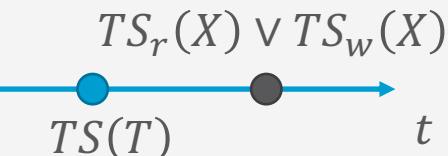
## Verfahren mit Zeitstempelung

- Zeitstempel  $TS(T)$  der Transaktion  $T$ 
  - Wie vorher definiert: Eindeutiger Identifikator, der vom DBMS generiert wird, um eine Transaktion zu identifizieren
    - In der Regel der Start-Zeitpunkt (Ablesen der internen Systemuhr) von  $T$
  - Zeitstempelbasierte Ansätze verwenden *keine Sperren* → *keine Deadlocks*
  - Idee: Algorithmus stellt sicher, dass der Zugriff von konfliktären Operatoren nicht die Reihenfolge (mindestens) eines äquivalenten seriellen Schedules verletzt
    - $TS_r(X)$ : Lesezeitstempel von  $X$ 
      - Größter (aktuellster)  $TS(T)$  aller Transaktionen, die  $X$  erfolgreich gelesen haben
    - $TS_w(X)$ : Schreibzeitstempel von  $X$ 
      - $TS(T)$  der Transaktion  $T$ , die  $X$  erfolgreich geschrieben hat

## Basis-Zeitstempelordnung

- $T$  möchte  $\text{write\_item}(X)$  durchführen

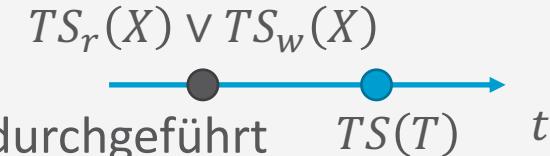
- Wenn  $TS_r(X) > TS(T)$  oder  $TS_w(X) > TS(T)$ :



- Jüngere Transaktionen haben  $X$  schon gelesen oder geschrieben
- Operation wird abgewiesen,  $T$  wird abgebrochen („zu spät“)
- $T$  neu starten mit neuem Zeitstempel

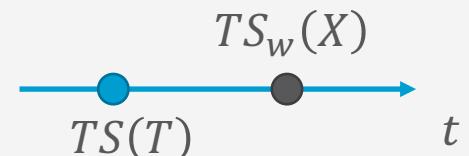
- Sonst:

- $\text{write\_item}(X)$  wird durchgeführt
- $TS_w(X) := TS(T)$



- $T$  möchte  $\text{read\_item}(X)$  durchführen:

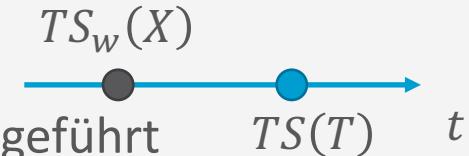
- Wenn  $TS_w(X) > TS(T)$ :



- Jüngere Transaktionen haben  $X$  schon geschrieben
- Operation wird abgewiesen,  $T$  wird abgebrochen („zu spät“)
- $T$  neu starten mit neuem Zeitstempel

- Sonst:

- $\text{read\_item}(X)$  wird durchgeführt
- $TS_r(X) := \max\{TS_r(X), TS(T)\}$



## Zeitstempelung: Beispiel

- Schedule:  $r_8(X), r_6(X), r_9(X), w_8(X), w_{11}(X), r_{10}(X)$ 
  - Notation: Operation  $p_i(X)$  der Transaktion  $T_i$  mit Zeitstempel  $i = TS(T)$
- Initial:  $TS_r(X) := 0, TS_w(X) := 0$

Anfrage	Antwort	Änderungen
$r_8(X)$	OK	$TS_r(X) := 8$
$r_6(X)$	OK	$TS_r(X) := 8$
$r_9(X)$	OK	$TS_r(X) := 9$
$w_8(X)$	abgelehnt	$a_8$
$w_{11}(X)$	OK	$TS_w(X) := 11$
$r_{10}(X)$	abgelehnt	$a_{10}$

## Multiversionsprotokolle

- Multiversionsprotokolle verwalten unterschiedliche Versionen eines Datenobjekts, also auch die mit alten Werten
- Bei Zugriff wird die jeweils passende Version des Datenobjekts an die Transaktion geliefert
- Umsetzungen mit
  - Zeitstempelordnung
  - Zertifizierungssperren

## Multiversionsprotokoll mit Zeitstempelordnung

- Mehrere Versionen  $X_1, \dots, X_k$  von einem Datenobjekt  $X$
- Für jedes  $X_i$  werden Zeitstempel gespeichert:
  - $TS_r(X_i)$ : der größte (aktuellste) aller Zeitstempel von Transaktionen, die diese Version gelesen haben
  - $TS_w(X_i)$ : Zeitstempel der Transaktion  $T$ , die den Wert dieser Version geschrieben hat

# Multiversionsprotokoll mit Zeitstempelordnung

- Zwei Regeln für die Serialisierung:
    1. Schreiben:
      - Wenn Transaktion  $T$  eine  $\text{write\_item}(X)$ -Operation ausführen will und
        - Für die Version  $i$  von  $X$  mit dem größten  $TS_w(X_i)$  aller Versionen von  $X$  gilt, dass
          - $TS_w(X_i) < TS(T)$  und (ältere Transaktion als  $T$  hat  $X_i$  geschrieben)
          - $TS_r(X_i) > TS(T)$ , (jüngere Transaktion als  $T$  hat  $X_i$  gelesen)
      - Sonst
        - Erstelle eine neue Version  $X_{k+1}$  von  $X$
        - Setze  $TS_w(X_{k+1})$  und  $TS_r(X_{k+1})$  auf  $TS(T)$

## Multiversionsprotokoll mit Zeitstempelordnung

- Zwei Regeln für die Serialisierung

### 2. Lesen:

- Wenn Transaktion  $T$  eine  $\text{read\_item}(X)$ -Operation ausführen will,
  - Gibt es eine Version  $i$  von  $X$  mit dem größten  $TS_w(X_i)$  aller Versionen von  $X$ , für die gilt
    - $TS_w(X_i) < TS(T)$ ,

dann wird  $T$  der Wert von  $X_i$  geliefert und  $TS_r(X_i)$  wird auf den größeren Wert von  $TS(T)$  und  $TS_r(X_i)$  gesetzt.

→  $\text{read\_item}(X)$ -Operationen können immer ausgeführt werden

## Multiversionsprotokoll mit Zertifizierungssperren

- Standardfall: Wenn Transaktion  $T'$  eine Schreibsperre auf Objekt  $X$  hat, kann kein andere Transaktion  $T$  auf  $X$  zugreifen
- Beim Multiversionsprotokoll mit Zertifizierungssperren ist es  $T$  gestattet,  $X$  zu lesen, während eine Transaktion  $T'$  eine Schreibsperre auf  $X$  hält
  - Für jedes Objekt  $X$  sind zwei Versionen  $X', X''$  zugelassen:
    - Version  $X'$  muss dabei immer von einer bestätigten Transaktion geschrieben worden sein
    - Version  $X''$  wird erzeugt, wenn eine Transaktion  $T'$  eine Schreibsperre auf  $X$  anfordert
- Andere Transaktionen können die bestätigte Version von  $X$ , nämlich  $X'$ , weiterhin lesen, während  $T'$  eine Schreibsperre erhält und mit  $X''$  arbeitet
- Wenn  $T'$  bereit ist ein COMMIT durchzuführen, muss  $T'$  **Zertifizierungssperren** für *jedes* Objekt  $X$  anfordern, für das es eine *Schreibsperre* hat
  - Evtl. warten, bis andere Transaktionen ihre Lesesperren auf  $X$  freigeben

## Multiversionsprotokoll mit Zertifizierungssperren

- Erweitertes Sperrkonzept mit Mehrfachmodus-Sperren
  - Statt Zeitstempelordnung
- Einführung eines neuen Zustands von Sperren, damit vier:
  - 🔒 Lesegesperrt (S)
  - 📝 Schreibgesperrt (X)
  - 🔒 **Zertifizierungsgesperrt (C)**
  - 🔓 Entsperrt (0)
- Idee: Auch bei Lesesperrre noch Schreiben und bei Schreibsperrre noch Lesen zu erlauben, wenn dies geordnet geschieht
  - Kompatibilitätsmatrix rechts

Anforderung \ Zustand	0	S	X	C
S	✓	✓	✓, aber...	-
X	✓	✓, aber...	-	-
C	✓	-	-	-

# Optimistische Verfahren

- Bisherige Verfahren: pessimistisch, Kontrolle vor Ausführung
- Optimistische Verfahren: während der Ausführung keine Kontrolle

## 1. Lesephase

- Bestätigte Datenobjekte der DB lesen
- Aktualisierungen werden nur in lokale Kopien geschrieben

## 2. Validierungsphase

- Kontrollmechanismen werden angestoßen, die Serialisierbarkeit/Korrektheit (nachträglich) prüfen

## 3. Schreibphase

- Wenn Validierungsphase erfolgreich: Aktualisierungen werden auf der DB ausgeführt
- Sonst: Transaktion wird abgebrochen und neu angestoßen

- Lohnt sich, wenn es nur selten Konflikte gibt

- Z.B. bei großer DB → verteiltes Arbeiten (an unterschiedlichen Stellen der DB)

## Granularität des Sperrens

- Datenobjekt kann alles sein, von Datenbank bis hin zu Datensatz (Zeile einer Tabelle)
- Granularität des Sperrens unterliegt Abwägung
- Sperren mit multipler Granularität

### Level Sperren



## Sperren mit multipler Granularität

- Granularität von Sperren für jede Transaktion entscheiden (abhängig von Charakteristik)
  - Zeilenweise Sperre z.B. für

```
select *          Q1
from Kunden
where Kunden_ID = 42
```

- und eine Tabellen-Sperre für

```
select *          Q2
from Kunden
```

- Wie können die Sperren für die Transaktionen koordiniert werden?
  - Für  $Q_2$  sollen nicht für alle Tupel umständlich Sperrkonflikte analysiert werden

## Vorhabens-Sperren

- DBs setzen Vorhabens-Sperren (*intention locks*) für verschiedene Sperrgranularitäten ein
  - Sperrmodus Intention Share (IS)
  - Sperrmodus Intention Exclusive (IX)
  - Kompatibilitätsmatrix rechts
- Eine Sperre I auf einer größeren Ebene bedeutet, dass es eine Sperre \_ auf einer niederen Ebene gibt

Anforderung \ Zustand	0	S	X	IS	IX
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

## Vorhabens-Sperren

- Protokoll für Sperren auf mehreren Ebenen:
  1. Eine Transaktion kann jede Ebene  $g$  in Modus  $\underline{\_} \in \{S, X\}$  sperren
  2. Bevor Ebene  $g$  in Modus  $\underline{\_}$  gesperrt werden kann, muss eine Sperre  $\underline{I}$  für alle größeren Ebenen gewonnen werden

- Beispiel

- Anfrage  $Q_1$  würde anfordern
  - $IS$ -Sperre für Tabelle Kunden (plus Tablespace + DB)
  - $S$ -Sperre auf dem Tupel mit Kunden\_ID=42
- Anfrage  $Q_2$  würde anfordern
  - ( $IS$ -Sperren für Tablespace + DB)
  - $S$ -Sperre für die Tabelle Kunden

```
select *          Q1
from Kunden
where Kunden_ID = 42

select *          Q2
from Kunden
```

Anforderung	Zustand	0	S	X	IS	IX
S		✓	✓	–	✓	–
X		✓	–	–	–	–
IS		✓	✓	–	✓	✓
IX		✓	–	–	✓	✓

# Entdeckung von Konflikten

- Beispiel
  - Momentane Sperren auf Tabelle Kunden oder tiefer
    - Tabelle Kunden: **IS** von  $Q_1$ , **S** von  $Q_2$
    - Tupel mit Kunden\_ID=42: **S** von  $Q_1$
  - Nehmen wir an, Anfrage  $Q_3$  ist auch noch zu bearbeiten
  - Benötigte Sperren
    - **IX**-Sperre auf Tabelle Kunden (plus Tablespace + DB)
    - **X**-Sperre auf dem Tupel mit Kunden\_ID=17
      - Kompatibel mit  $Q_1$  (kein Konflikt zwischen **IX** und **IS** auf Tabellenebene)
      - Inkompaktil mit  $Q_2$  (**S**-Sperre auf Tabellenebene von  $Q_2$  steht in Konflikt mit der **IX**-Sperre bzgl.  $Q_3$ )

Anforderung	Zustand 0	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

$Q_1$

```
select *  
from Kunden  
where Kunden_ID = 42
```

$Q_2$

```
select *  
from Kunden
```

$Q_3$

```
update Kunden  
set Name='John Doe'  
where Kunden_ID = 17
```

## Konsistenzgarantien

- In einigen Fällen kann man mit einigen kleinen Fehlern im Anfrageergebnis leben
  - Fehler bezüglich einzelner Tupel machen sich in Aggregatfunktionen evtl. kaum bemerkbar
    - Lesen inkonsistenter Werte (inconsistent read Anomalie)
  - Ab SQL-92 kann man Isolations-Modi spezifizieren:  
**SET ISOLATION <MODE>**
    - Verfügbare Modi: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE
      - Je weiter durch in der Liste, desto strikter (weniger Fehler)
      - Je weiter durch in der Liste, umso mehr Verwaltungsaufwand, z.B. für Sperren (weniger Durchsatz)

```
set isolation <mode>;
```

## Isolations-Modi

- **Read uncommitted** (auch: ‘dirty read’ oder ‘browse’)
  - Nur Schreib sperren akquiriert (nach 2PL)

S	$T_1$	$T_2$ (read uncommitted)	DB: <i>acct</i>
1	read_item( <i>acct</i> )		1200
2	$acct := acct - 100$		1200
3	write_item( <i>acct</i> )		1200
4		read_item( <i>acct</i> )	1100
5		$acct := acct - 200$	1100
6	abort		1200
7		write_item( <i>acct</i> )	1200

Es muss kein read lock  
erworben werden

Read uncommitted nur  
für lesende Transaktion

## Isolations-Modi

- **Read committed** (auch ‘cursor stability’)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
    - Transaktion sieht Daten, wie sie zum Zeitpunkt der aktuellen Operation committed sind
  - Schreib sperren nach 2PL

$S$	$T_1$	$T_2$ (read committed)
1		read_item( $A$ )
2	write_item( $A$ )	
3	write_item( $B$ )	
4	commit	
5		read_item( $B$ )
6		read_item( $A$ )

Nur committed gelesen

Aber:  
Unrepeatable Read  
nicht ausgeschlossen

## Isolations-Modi

- **Repeatable read** (auch ‘read stability’)
  - Lesen und Schreiben mittels Lese- und Schreibsperrn eines konsistenten Zustandes zu Beginn
    - Transaktion sieht Daten, wie sie zu Beginn der Transaktion (Sperrzuteilung) committed sind
  - Problem: Serialisierung nicht garantiert
    - Anfrage  $Q_1$ 
      - `select sum(Gehalt)`  
`from Mitarbeiter`  
`where AbtNr=5`
      - Anfrage  $Q_2$ 
        - `insert into Mitarbeiter`  
(SVN, NName, VName, Adresse, Gehalt, GebDatum, AbtNr, Vorges SVN)  
`values ('90123456G789 ', 'Marini', 'Richard',`  
`'98 Oak Forest, Katy, TX', 37000, '30.12.1962', 5, '67890123D456')`
      - Neue Daten erscheinen während der Ausführung von  $Q_1 \rightarrow$  **Phantom Read**

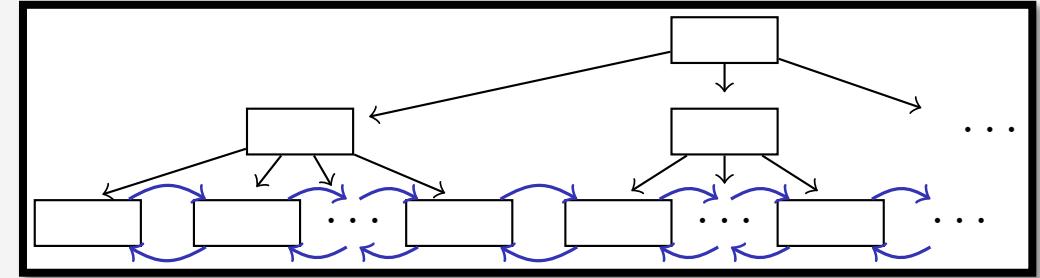
## Isolations-Modi

- **Read uncommitted** (auch: ‘dirty read’ oder ‘browse’)
  - Nur Schreibsperren akquiriert (nach 2PL)
- **Read committed** (auch ‘cursor stability’)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
  - Schreibsperren nach 2PL
- **Repeatable read** (auch ‘read stability’)
  - Lese- und Schreibsperren nach 2PL
- **Serializable**
  - Zusätzliche Sperranforderungen I, um Phantomproblem zu begegnen

## Resultierende Konsistenzgarantien

Isolationsmodus	dirty read	non-repeatable read	phantom read
<b>read uncommitted</b>	möglich	möglich	möglich
<b>read committed</b>	–	möglich	möglich
<b>repeatable read</b>	–	–	möglich
<b>serializable</b>	–	–	–

- Einige Implementierungen unterstützen mehr, weniger oder andere Isolationsmodi
  - PostgreSQL: *predicate locks*, z.B. Bedingungen auf Spalten
    - Blockieren nicht, sondern markieren mögliche Abhängigkeiten zwischen Transaktionen, die bei Verdacht auf Anomalie abgebrochen werden
  - Nur wenige Anwendungen benötigen (volle) Serialisierbarkeit

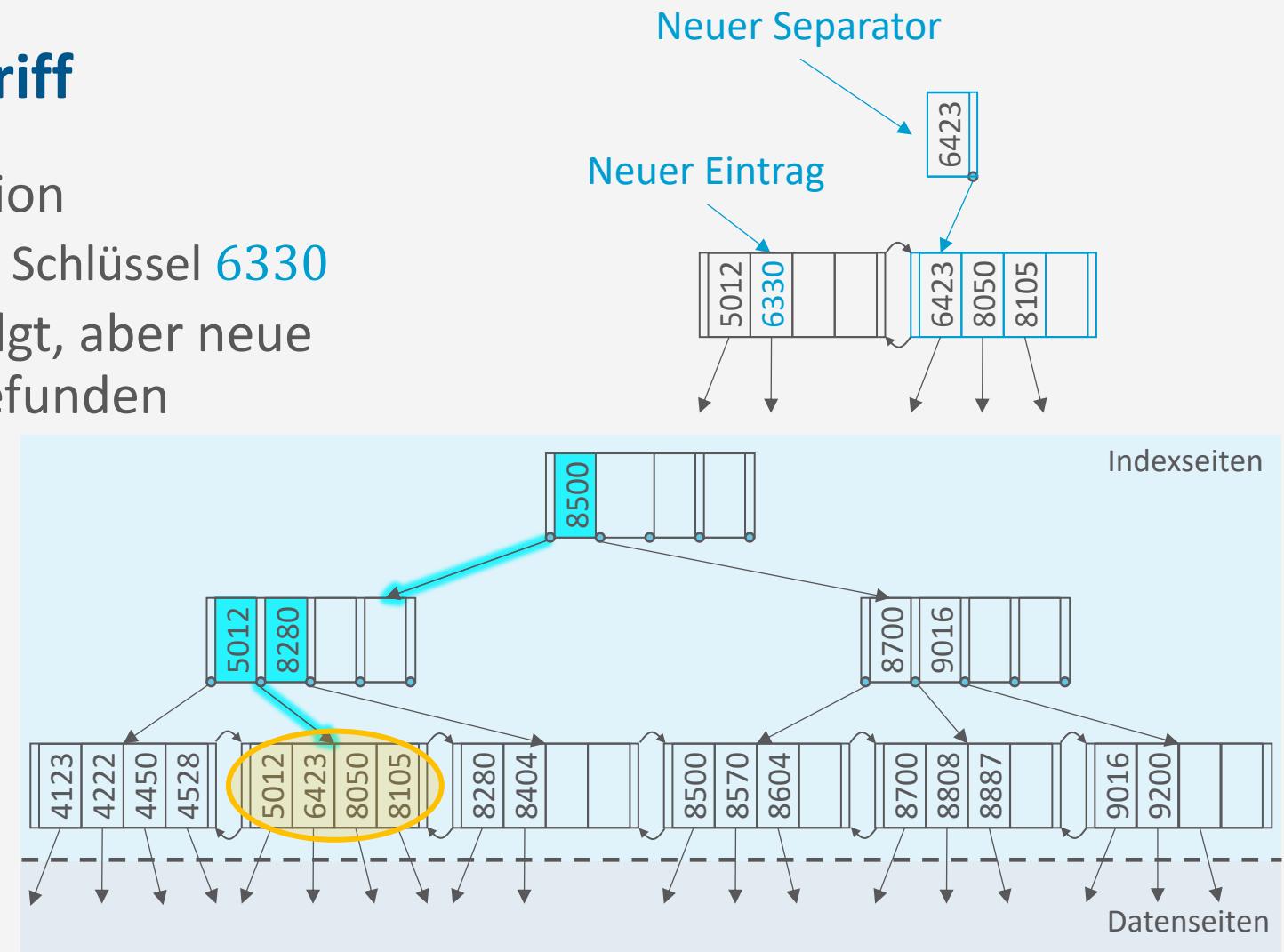


# Sperren in Index-Strukturen

Sperrverwaltung

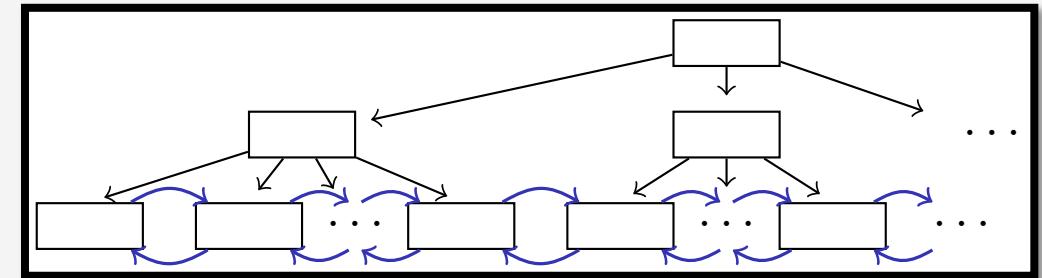
## Nebenläufigkeit beim Indexzugriff

- Transaktion  $T_w$  führt zu Splitoperation
  - Beispiel: Einfügen eines Eintrags mit Schlüssel **6330**
- Angenommen, Aufspaltung ist erfolgt, aber neue Verzeigerung hat noch nicht stattgefunden
- Weiterhin: nebenläufiges Lesen in Transaktion  $T_r$ , sucht nach **8050**
  - Verzeigerung zeigt auf zweiten Blattknoten mit 5012,6330 mit Zeigern auf dritten Blattknoten mit 8280,8404
- Datensatz wird nicht gefunden  
→ Sperren nötig



## Sperren und B<sup>+</sup>-Baum-Indexe

- B<sup>+</sup>-Baum-Operationen
  - Für die Suche erfolgt ein Top-Down-Zugriff
  - Für Aktualisierungen
    - Suche
    - Daten in Blatt eintragen / löschen
      - Ggf. Aufspaltungen / Merge von Knoten nach oben
- Nach 2PL
  - Müssen Lese/Schreib-Sperren auf dem Weg nach unten akquiriert werden (Konversion provoziert ggf. Deadlocks)
  - Müssen alle Sperren bis zum Ende gehalten werden



- Reduziert die Nebenläufigkeit drastisch
- Während des Indexzugriffs einer Transaktion müssen alle anderen Transaktionen warten, um die Sperre für die Wurzel des Index zu erhalten
  - Wurzel wird zum Flaschenhals und serialisiert alle (Schreib-)Transaktionen
  - 2PL nicht angemessen für B<sup>+</sup>-Bäume

## Sperrprotokoll für B<sup>+</sup>-Bäume

- Protokoll Write-Only-Tree-Locking (WTL)
  1. Für alle Baumknoten  $n$  außer der Wurzel kann eine Sperre nur akquiriert werden, wenn die Sperre für den Elternknoten akquiriert wurde
    - Sperrkopplung
  2. Sobald ein Knoten entsperrt wurde, kann für ihn nicht erneut eine Sperre angefordert werden durch dieselbe Transaktion (2PL)

→ Garantiert Serialisierbarkeit
- Und damit gilt:
  - Alle Transaktionen folgen Top-Down-Zugriffsmuster
  - Keine Transaktion kann dabei andere überholen
  - WTL-Protokoll ist deadlockfrei

## Aufspaltungssicherheit

- Wir müssen auf dem Weg nach unten in den B+-Baum Schreib sperren wegen möglicher Aufspaltungen halten
- Allerdings kann man leicht prüfen, ob eine Spaltung von Knoten  $n$  die Vorgänger überhaupt erreichen kann
  - Wenn  $n$  weniger als  $2d$  Einträge enthält, kommt es nicht zu einer Weiterreichung der Aufspaltung nach oben
- Ein Knoten, der diese Bedingung erfüllt, heißt aufspaltungssicher (**split safe**)
- Ausnutzung zur frühen Sperrrückgabe
  - Wenn ein Knoten auf dem Weg nach unten als aufspaltungssicher gilt, können alle Sperren der Vorgänger zurückgegeben werden
  - Sperren werden weniger lang gehalten
  - Aufweichung des Zwei-Phasen-Sperrprotokolls

## Sperrkopplungsprotokoll (Variante 1)

**procedure** read\_lock(*key*)

Place S lock on *root*

*current*  $\leftarrow$  *root*

**while** *current* is not a leaf node **do**

    Find *child* for *key*

    Place S lock on *child*

    Release S lock on *current*

*current*  $\leftarrow$  *child*

**procedure** write\_lock(*key*)

Place X lock on *root*

*current*  $\leftarrow$  *root*

**while** *current* is not a leaf node **do**

    Find *child* for *key*

    Place X lock on *child*

*current*  $\leftarrow$  *child*

**if** *current* is safe **then**

        Release all locks on ancestors of *current*

## Erhöhung der Nebenläufigkeit

- Auch mit Sperrkopplung werden eine beträchtliche Anzahl von Sperren für innere Knoten benötigt
  - Mindert Nebenläufigkeit
- Innere Knoten selten durch Aktualisierungen betroffen
  - Wenn  $d = 50$ , dann Aufspaltung bei jeder 50. Einfügung (2% relative Auftretenshäufigkeit)
- Eine Einfügetransaktion könnte optimistisch annehmen, dass keine Aufspaltung nötig ist
  - Bei inneren Knoten werden während der Baumtraversierung nur Lesesperren akquiriert
    - Schreibsperrre dann nur für das betreffende Blatt
  - Wenn die Annahme falsch ist, traversiere Indexbaum erneut unter Verwendung korrekter Schreibsperren

## Sperrkopplungsprotokoll (Variante 2)

- Modifikationen nur für Schreibvorgänge

```
procedure optimistic_write_lock(key)
    Place S lock on root
    current ← root
    while current is not a leaf node do
        Find child for key
        if child is a leaf then
            Place X lock on child
        else
            Place S lock on child
            Release lock on current
            current ← child
        if current is unsafe then
            Release all locks and repeat with write_lock
```

```
procedure write_lock(key)
    Place X lock on root
    current ← root
    while current is not a leaf node do
        Find child for key
        Place X lock on child
        current ← child
        if current is safe then
            Release all locks on ancestors of current
```

## Diskussion

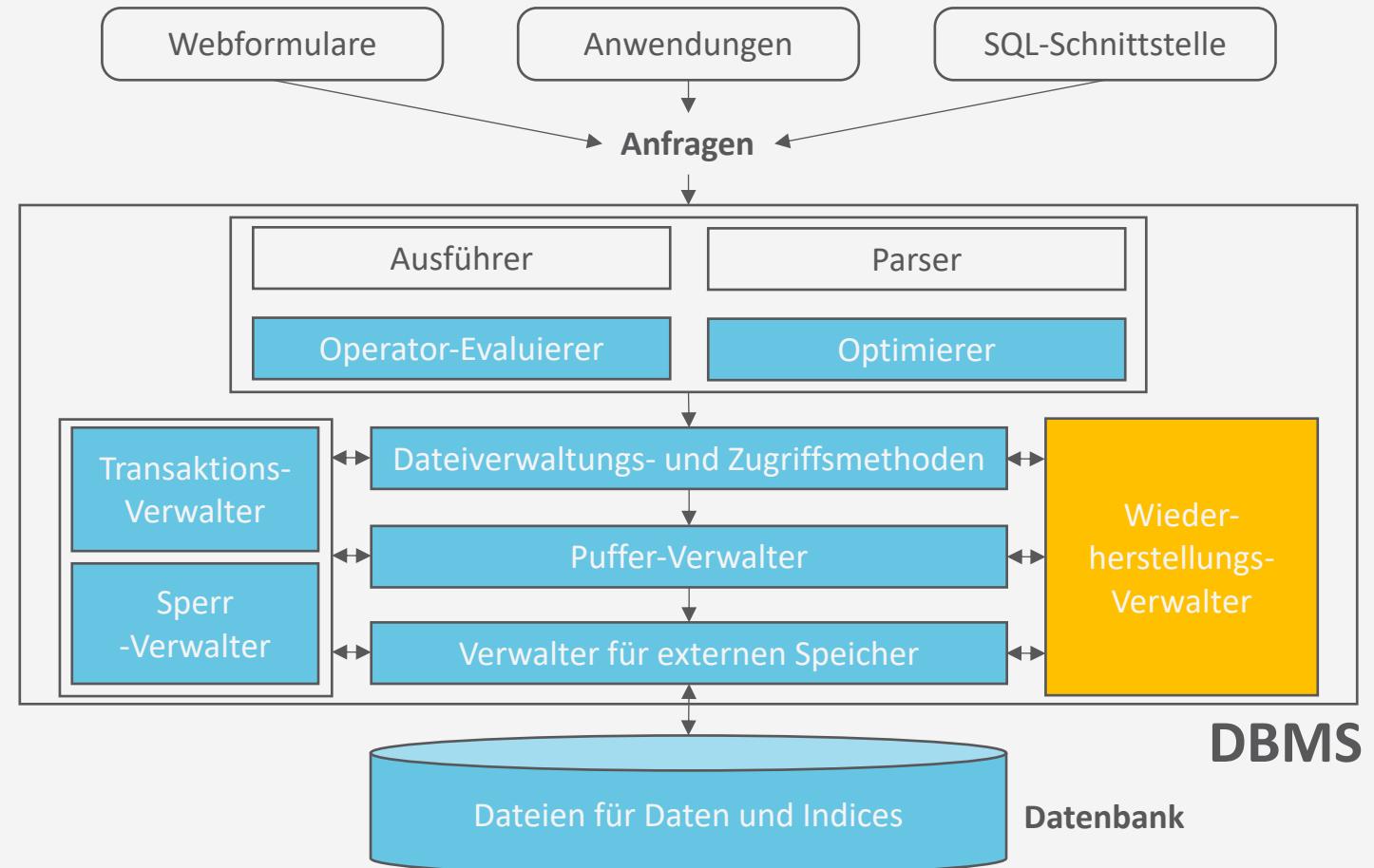
- Wenn eine Aufspaltung nötig ist, wird der Vorgang abgebrochen und erneut aufgesetzt
- Die resultierende Verarbeitung ist korrekt, obwohl es nach einem erneuten Sperren aussieht (was für WTL nicht erlaubt ist)
- Der Nachteil von Variante 2 ist, dass im Falle einer Blattaufspaltung Arbeit verloren ist
- Es gibt viele Varianten dieser Sperrprotokolle

## Zwischenzusammenfassung

- Binäre Sperren: Datenobjekt ist gesperrt oder nicht; sehr restriktiv
- Zwei-Phasen-Sperrprotokoll
  - Grundlegendes Modell für viele Protokolle, die auf Sperren basieren
  - Keine Sperre mehr anfordern, nachdem eine erste Sperre freigegeben worden ist
- Zeitstempelbasierte Protokolle
- Multiversionsprotokolle: Verschiedene Versionen eines Datenobjekts
  - Umsetzung: zeitstempelbasiert oder mit Sperren
- Optimistische Verfahren: Änderungen nur lokal durchführen; Validierung um Änderungen persistent zu machen
- Granularitäten, Vorhaben-Sperren, Isolationsmodi
- Sperren in Index-Strukturen

# Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
    - Sperren, Sperrprotokolle
    - Isolationsmodi
  - Wiederherstellungsverwaltung



# Überblick: 7. Transaktionen

## A. *Transaktionsverarbeitung*

- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

## B. *Sperrverwaltung*

- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

## C. *Wiederherstellungsverwaltung*

- Fehlersituationen
- Logging
- Recovery

## TRANSAKTIONSFehler (1/3)

- Verschiedene Gründe, weshalb eine DB-Transaktion fehlschlagen kann
- Sieben typische Fehlerfälle
  - Mit in der Literatur uneinheitlicher Bezeichnung
  - Meist in Kategorien untergliedert (z.B. System-, Medien-, Transaktionsfehler)

### 1. Systemabsturz [System- und Medienfehler]

- Hardware-, Software- oder Netzwerkproblem tritt während der Transaktionsverarbeitung auf
  - Keine adäquate Handhabung durch die Software
- „Systemabsturz“

### 2. Transaktionsabsturz [Transaktions-, System- und Medienfehler]

- Einzelne Operationen schlagen fehl, z.B. aufgrund einer Division durch den Wert Null, aufgrund fehlerhafter Parameterwerte oder aus Programmfehlern
- Benutzer bricht Transaktion (willkürlich) ab

## WEITERE TRANSAKTIONSFehler (2/3)

### 3. Lokale Fehler: [Transaktionsfehler]

- Während der Transaktionsausführung können Zustände auftreten, die einen Transaktionsabbruch notwendig machen
  - Beispiel: zur Transaktionsausführung benötigte Daten können nicht ermittelt werden

### 4. Speicherfehler: [Medienfehler]

- Partieller oder vollständiger Ausfall eines Speichersystems  
→ Lese-/Schreibfehler bei der Transaktionsausführung

### 5. „Katastrophen“: [Medienfehler]

- Subsumierung verschiedener Ausnahmesituationen, z.B. physischer Verlust eines Datenträgers durch Brand oder Diebstahl
- Ausnahmesituationen, die sich aus einer fehlerhaften Bedienung des Systems ergeben, wobei z.B. ein Datenträger versehentlich gelöscht wurde

## WEITERE TRANSAKTIONSFehler (3/3)

### 6. Transaktionsannullierung: [kein klassischer Fehlerfall]

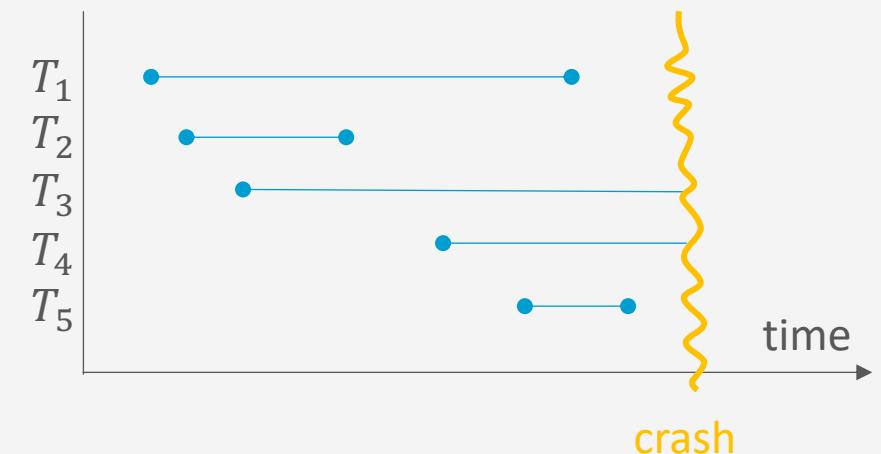
- Ein bestimmter DB-Zustand kann dazu führen, dass eine Transaktion „aus inhaltlichen Gründen“ nicht weiter ausgeführt wird, falls z.B. kein genügend hohes Guthaben für eine Abbuchung vorliegt

### 7. Nebenläufigkeitskontrolle: [kein klassischer Fehlerfall]

- Komponente zur Überwachung der nebenläufigen Transaktionsausführung veranlasst Unterbrechung (nicht Abbruch) einer Transaktion, um Interferenzen mit anderen Transaktionen zu vermeiden
  - Gestoppte Transaktion wird später wieder gestartet
- Die aufgezählten und weitere Fehlerfälle kann man durch nachfolgend beschriebene, einfache Basis-Mechanismen zu „reparieren“ versuchen

## Beispiel: System- oder Medienfehler

- Transaktionen  $T_1, T_2, T_5$  wurden vor dem Ausfall erfolgreich beendet
  - **Dauerhaftigkeit**: Es muss sicher gestellt werden, dass die Effekte beibehalten werden oder wiederhergestellt werden können (*redo*)
- Transaktionen  $T_3, T_4$  wurden noch nicht beendet
  - **Atomarität**: Alle Effekte müssen rückgängig gemacht werden (*undo*)

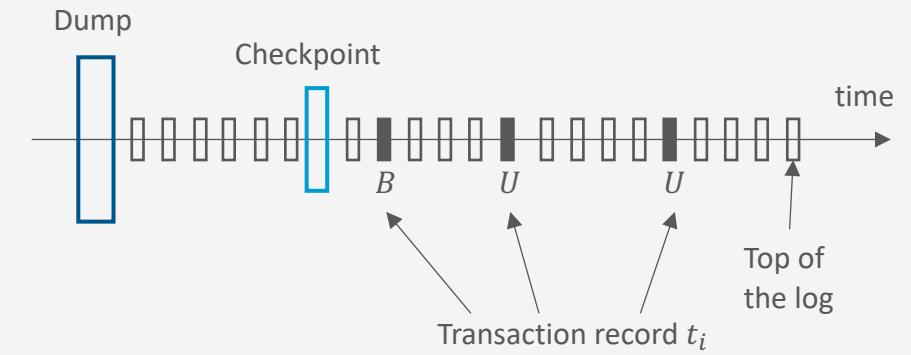


## Wiederherstellungsverwaltung: Aufgabe

- Zentrale Komponente der Transaktionsverarbeitung
- Sichert **Atomarität** und **Dauerhaftigkeit**
- Koordiniert transaktionsübergreifend die Abarbeitung der DB-Operationen:
  - BEGIN\_TRANSACTION und END\_TRANSACTION
  - COMMIT und ABORT
- Bietet eigene Primitive für Recovery nach Fehlersituationen:
  - Warm restart
  - Cold restart
- Grundlage zur praktischen Umsetzung dieser Funktionalität sind Aufbau und Verwaltung eines **Log-File**

# Logging

Wiederherstellungsverwaltung

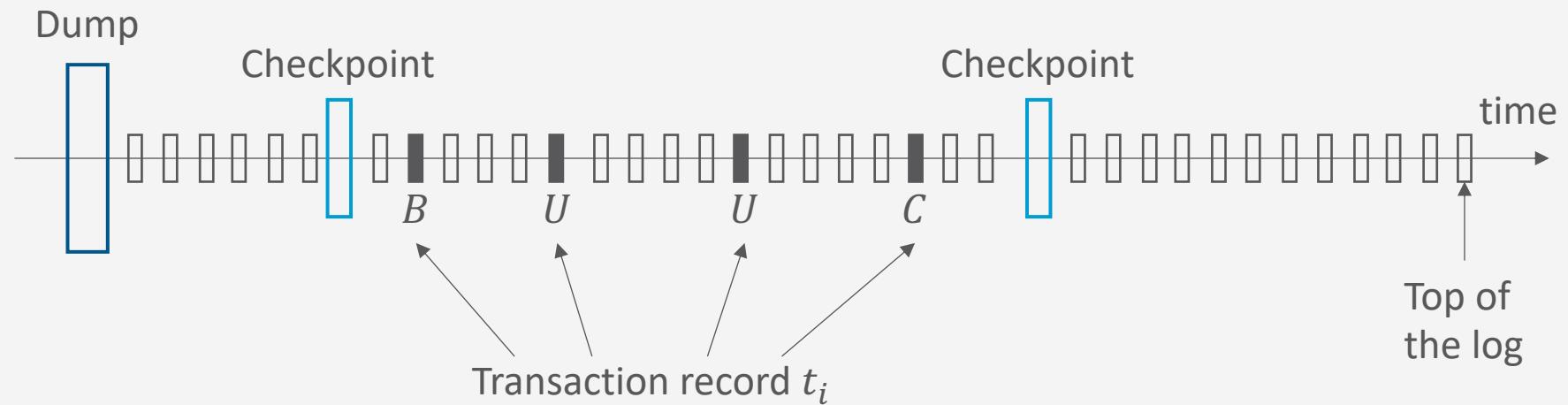


## Organisation des Log-File

- Sequentielle Struktur, die DB-Änderungen chronologisch speichert
    - Liegt auf persistentem Speicher
    - Verwaltung: Wiederherstellungsverwalter
  - EOT und Commit fallen zusammen
    - Commit heißt jetzt erstmal nur logisches Ende (vorher EOT), aber nicht zwangsweise persistente Speicherung
      - Der Dateiverwaltung soll überlassen werden, wann was geschrieben wird
      - Transaktionen, die einen Commit Eintrag haben, sollen bei der Wiederherstellung wiederholt werden, wenn nicht im Speicher
- Zwei Typen von Log-Einträgen:
    1. Transaktionsbezogene Einträge:
      - $BS / AS$ : Wert von  $X$  vor / nach  $T$  (*before / after state*)
      - BEGIN\_TRANSACTION :  $B(T)$
      - INSERT\_ITEM :  $I(T, X, AS)$
      - DELETE\_ITEM :  $D(T, X, BS)$
      - UPDATE\_ITEM :  $U(T, X, BS, AS)$
      - COMMIT :  $C(T)$
      - ABORT :  $A(T)$
    2. Systembezogene Einträge:
      - DUMP (vollständige DB-Kopie; eher selten)
      - CHECKPOINT (partielle Aktualisierung; häufig)

## Organisation des Log-File

- Transaction Record:
  - Zusammenfassung der INSERT\_ITEM-, UPDATE\_ITEM- und DELETE\_ITEM-Operationen einer einzelnen Transaktion
  - Beginnt mit (B)egin, endet mit (C)ommit oder (A)bort
  - Beispiel:

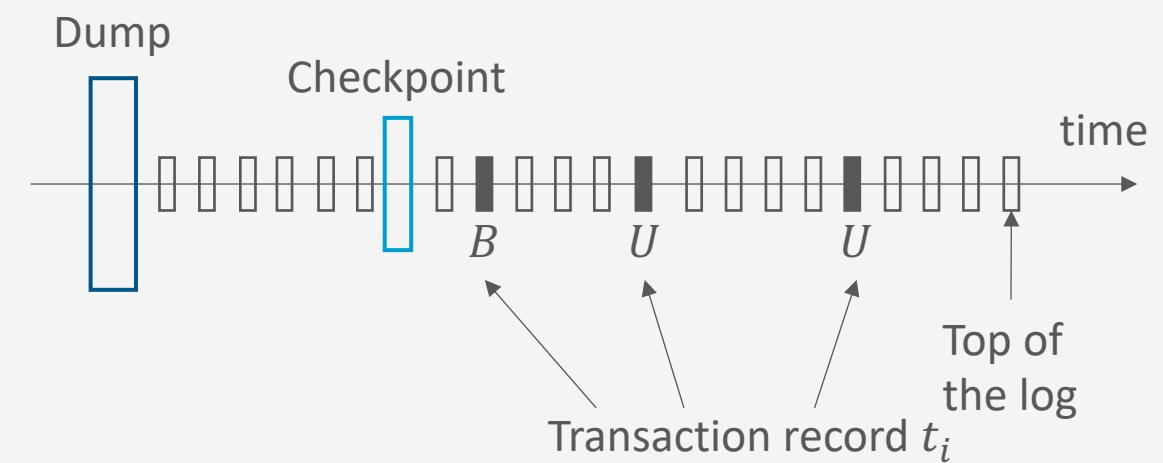


## Log-Information COMMIT und Fehler

- COMMIT-Point einer Transaktion  $T$  bezeichnet den Zeitpunkt, an dem alle DB-Zugriffsoperationen von  $T$  erfolgreich ausgeführt und im Log erfasst wurden
  - Zu diesem Zeitpunkt gilt Transaktion als bestätigt, ihre Wirkung soll persistent in der DB gespeichert werden
    - [COMMIT,  $T$ ]-Eintrag im Log wird generiert, als Zeichen der logischen, aber noch nicht physischen Beendigung der Transaktion
- Im Fehlerfall
  - Transaktionen  $T$  mit BEGIN\_TRANSACTION und ohne COMMIT werden – soweit nötig – rückgängig gemacht → UNDO
  - Transaktionen  $T$  mit BEGIN\_TRANSACTION und mit COMMIT können bzgl. ihrer Änderungsoperationen vollständig wiederholt werden, falls der DB-Zustand noch nicht persistent auf Platte gesichert wurde → REDO

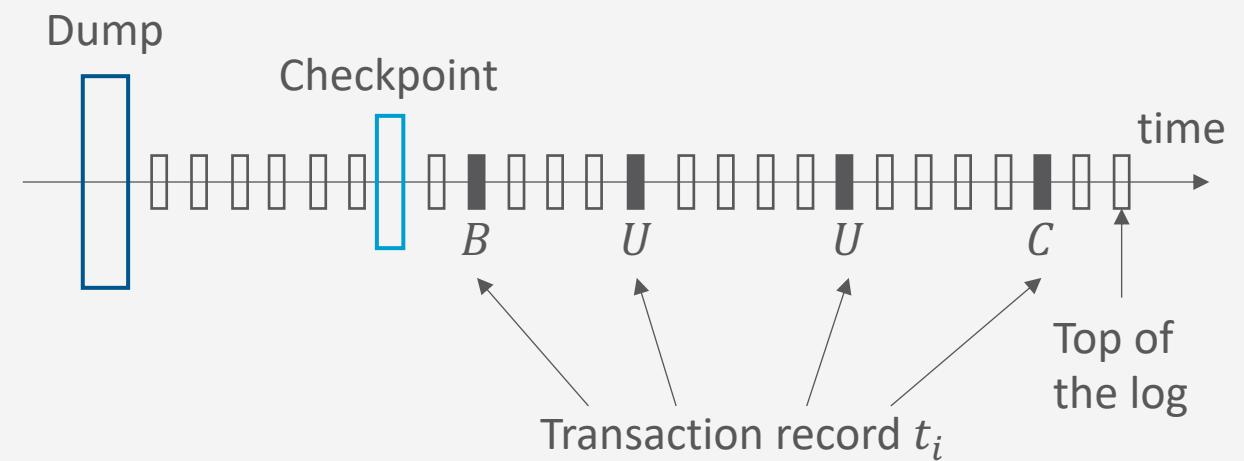
## UNDO

- Abbruch einer Transaktion  $T$
- DB muss in Zustand vor Transaktionsausführung gebracht werden
- Log rückwärts bis zum Beginn von  $T$  durchsuchen, um Operationen von  $T$  zu identifizieren und betroffene Datenobjekte  $X$  zurückzusetzen
- Beispiel: UNDO für ein Datenobjekt  $X$ 
  - UPDATE, DELETE: kopiere den Wert  $BS$  in  $X$
  - INSERT: lösche das Objekt  $X$



## REDO

- Systemfehler in der DB
- Alle bestätigten Transaktionen  $T$ , die noch nicht im Hintergrundspeicher sind, müssen wiederholt werden
- Log vom Beginn von  $T$  vorwärts durchsucht und alle auftretenden Operationen erneut realisiert
- Beispiel: REDO für ein Datenobjekt  $X$ 
  - INSERT, UPDATE:
    - Kopiere den Wert  $AS$  in  $X$
  - DELETE:
    - Lösche das Objekt  $X$



## Log-Information: Sicherer Speicher

- Um während des DBMS-Betriebs eine verlässliche Grundlage für UNDO/REDO im Fehlerfall zu liefern, muss das Log – neben der Protokollierung im Hauptspeicher (intern) – auch verlässlich auf Platte (extern) gespeichert werden
  - Nach Systemabsturz können nur solche Log-Einträge bei der Fehlerbehandlung berücksichtigt werden, die bereits auf Platte gespeichert sind – Hauptspeicherinhalte stehen u.U. nicht mehr zur Verfügung
  - Nicht jeder Log-Eintrag wird direkt auch auf Platte gespeichert, um hohe Verzögerungszeiten beim Log-Zugriff zu vermeiden
- Forcewriting**
  - Transaktion  $T$  kann erst dann ihren COMMIT-Point erreichen, wenn der  $T$  zugehörige Log-Inhalt verlässlich auf Platte gespeichert ist

## Checkpoints

- Zeichnet alle aktiven Transaktionen auf
- Aktualisiert Hintergrundspeicher partiell aufgrund abgeschlossener Transaktionen
- Wird periodisch angestoßen
  - Währenddessen keine COMMIT-Anweisungen für aktive Transaktionen
- Dienst-Ende: es wird synchron (*force*) ein CHECKPOINT-Eintrag im Log-File generiert
  - DB-Veränderungen abgeschlossener Transaktionen dauerhaft in die DB einfügen
- Checkpoint-Eintrag:
  - $CK(T_1, T_2, T_3, \dots, T_n)$ , enthält die Identifier  $T_i$  der aktiven Transaktionen
- Bei einem Recovery-Vorgang muss dann nur bis zum letzten Checkpoint-Eintrag zurückgegangen werden, um von dort an die aktiven Transaktionen abhandeln (UNDO/REDO)

## DUMP

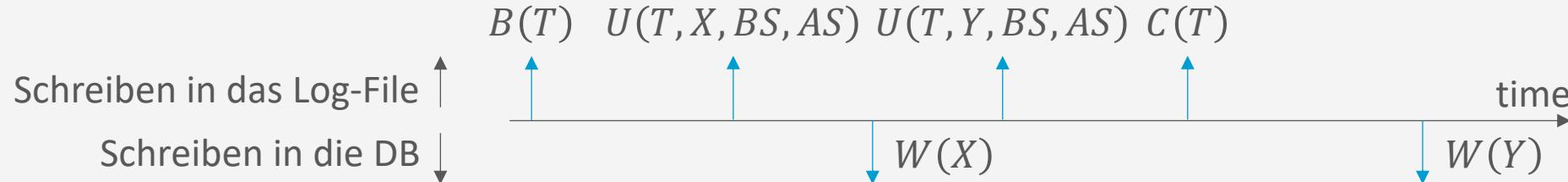
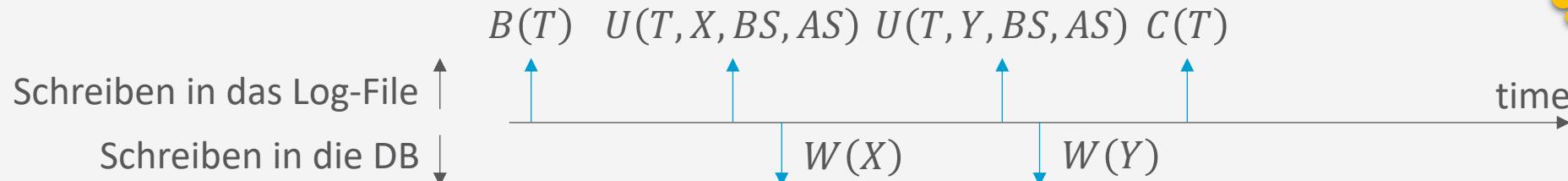
- Vollständige Kopie der DB
  - Backup
- Wird in der Regel erzeugt, wenn DB nicht operativ ist
  - (z.B. einmal pro Nacht)
- Wird im Allgemeinen im persistenten Speicher abgelegt
- Nach Erzeugung des DUMP entsprechender Eintrag im Log-File

## Grundregeln für Log-Einträge

- Zwei Grundregeln:
  - **Write-Ahead Log (WAL)**  
BS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die entsprechende Operation auf der DB ausgeführt wird
  - **Commit-Precendence (CP)**  
AS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die Transaktion abgeschlossen wird
- Damit führt Verlust des Hauptspeichers nicht zu Datenverlust
- Mögliche Situationen
  - Aktualisierung auf Hintergrundspeicher vor dem COMMIT → Kein REDO bei Recovery nötig
  - COMMIT vor Aktualisierung auf Hintergrundspeicher → Kein UNDO bei Recovery nötig

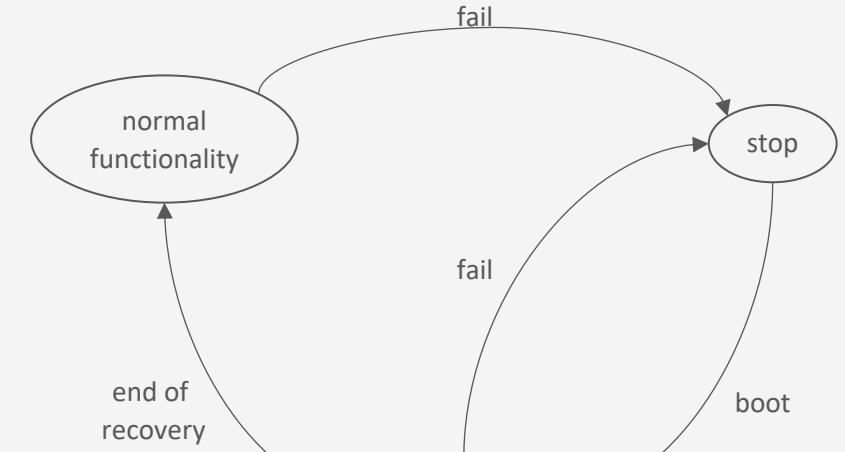
## Beispiele

Was geschieht  
bei Abbrüchen?

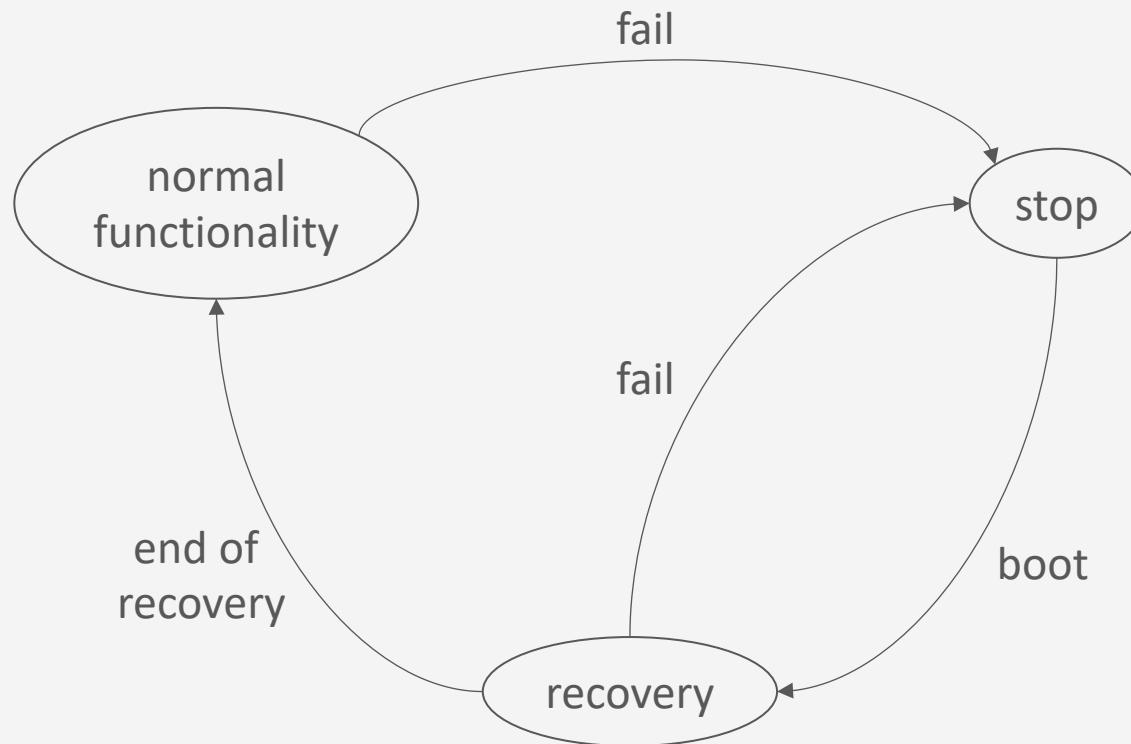


# Recovery

Wiederherstellungsverwaltung



## (Ideales) Fail-Stop-Model eines DBMS



## Boot: Warm / Cold Restart

- Fehlersituationen beim Datenmanagement ...
  - Systemfehler
    - Softwarefehler (z.B. Betriebssystemfehler) oder
    - Fehler von Geräten (z.B. aufgrund von Fehlern in der Stromversorgung).
    - Hauptspeicher geht verloren, Hintergrundspeicher bleibt erhalten
  - Gerätefehler
    - Fehler des Hintergrundspeichers (z.B. aufgrund eines „Platten-Crashes“)
    - Hauptspeicher und Hintergrundspeicher gehen verloren
    - Stabiler Speicher bleibt erhalten (Band, RAID-System)
- Protokolle zum Restart des DBMS bzw. der DB
  - **Warm Restart** ist im Fall von Systemfehlern geeignet
  - **Cold Restart** ist im Fall von Gerätefehlern geeignet

## Restart (Boot)

- Zustand von Transaktionen vor Stop
  - Abgeschlossen
    - Resultat abgeschlossener Transaktionen ist im stabilen Speicher gespeichert
  - COMMIT vorhanden, aber u.U. nicht abgeschlossen:
    - DB-Veränderungen der Transaktion müssen wiederholt werden
  - Kein COMMIT vorhanden
    - DB-Veränderungen der Transaktion müssen verworfen werden

## Warm Restart

- Phase 1: Suche im Log-File die Position des jüngsten CHECKPOINT-Eintrags
- Phase 2: Bilde UNDO- und REDO-Menge
  - UNDO-Menge: Transaktionen, die verworfen werden müssen
  - REDO-Menge: Transaktionen, die reproduziert werden können
- Phase 3: Behandle Transaktionen der UNDO-Menge
  - Durchlaufe das Log-File zurück zur Position der ersten Operation der ältesten Transaktion aus der UNDO- *und* der REDO-Menge
  - Führe UNDO für alle Operationen der Transaktionen der UNDO-Menge aus
- Phase 4: Behandle Transaktionen der REDO Menge
  - Durchlaufe das Log-File vorwärts und führe REDO für alle Operationen der Transaktionen in der REDO-Menge aus
- Vorgehen sichert **Atomarität** und **Dauerhaftigkeit**

## Cold Restart

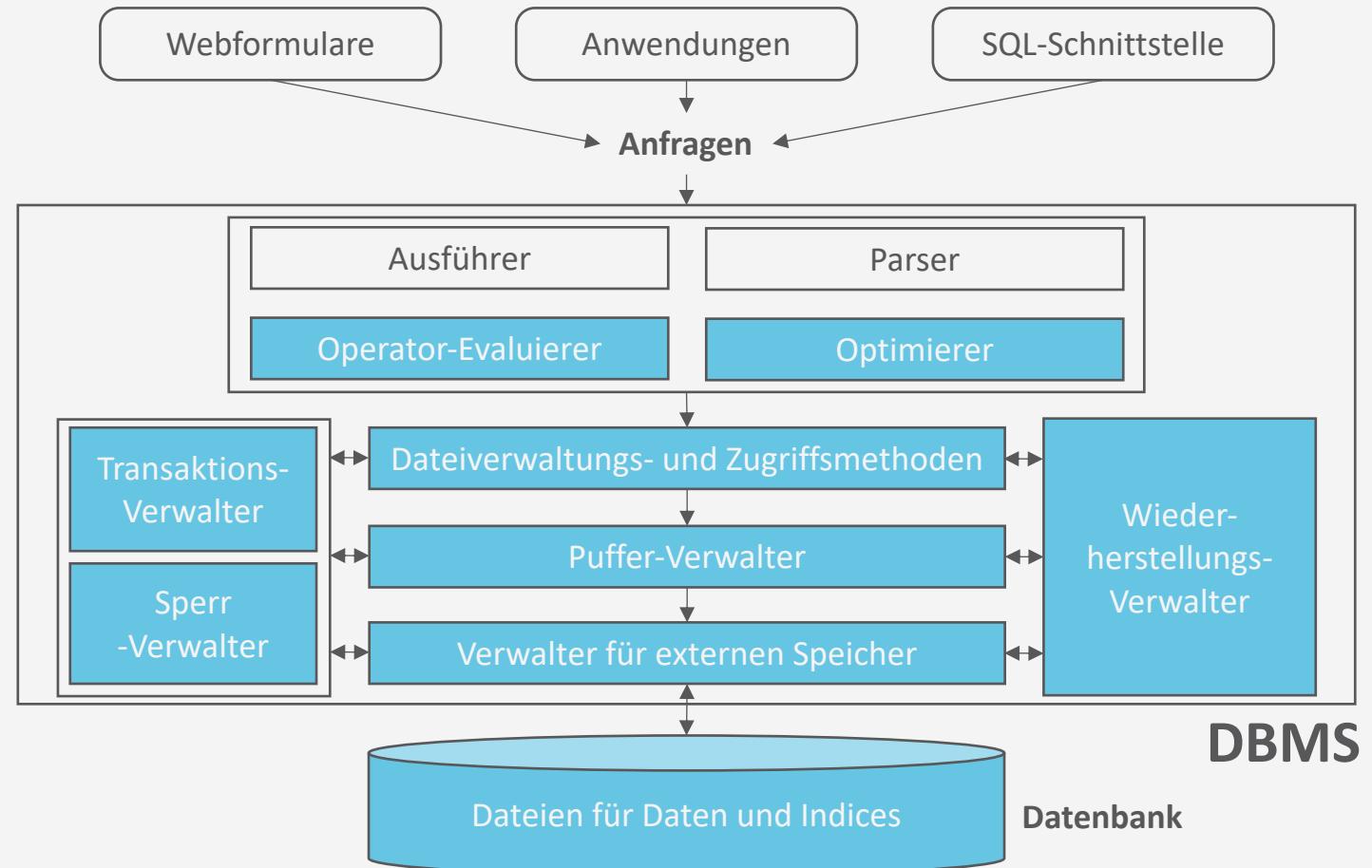
- Phase 1:
  - Der aktuellste DUMP wird genutzt, um die verlorenen bzw. zerstörten Teile der DB wieder neu im Hintergrundspeicher anzulegen
- Phase 2:
  - Das Log-File wird vorwärts durchlaufen und die dort vermerkten Operationen werden auf der Rekonstruktion der DB neu ausgeführt
- Phase 3:
  - Ein Warm Restart wird initiiert
  - Auch dieses Vorgehen sichert **Atomarität** und **Dauerhaftigkeit**

## Zusammenfassung

- Fehlersituationen
  - System-, Medien-, Transaktionsfehler
  - Dauerhaftigkeit, Atomarität möglicherweise verletzt
- Logging
  - Log-Information
  - Konzept und Organisation von Log-Files
  - Inhalt und Nutzung bei Fehlern
- Recovery
  - Wiederherstellung eines DBMS
    - Warm restart (Systemfehler) unter Nutzung des letzten Checkpoints
    - Cold restart (Gerätefehler) unter Nutzung des letzten Dumps

# Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung
    - Logging
      - Undo, Redo
      - Checkpoints, Dump
    - Recovery:
      - Warm / Cold Restart



# Überblick: 7. Transaktionen

## A. *Transaktionsverarbeitung*

- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

## B. *Sperrverwaltung*

- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

## C. *Wiederherstellungsverwaltung*

- Fehlersituationen
- Logging
- Recovery

→ Verteilte Datenbanken