MOBI-DBS-B: Datenbanksysteme Transaktionen

Vorlesung Sommersemester 2019

Tanya Braun, Universität zu Lübeck

Lehrauftrag SoSe 19, Universität Bamberg

Transaktionen – DB im Mehrbenutzerbetrieb

Inhalte

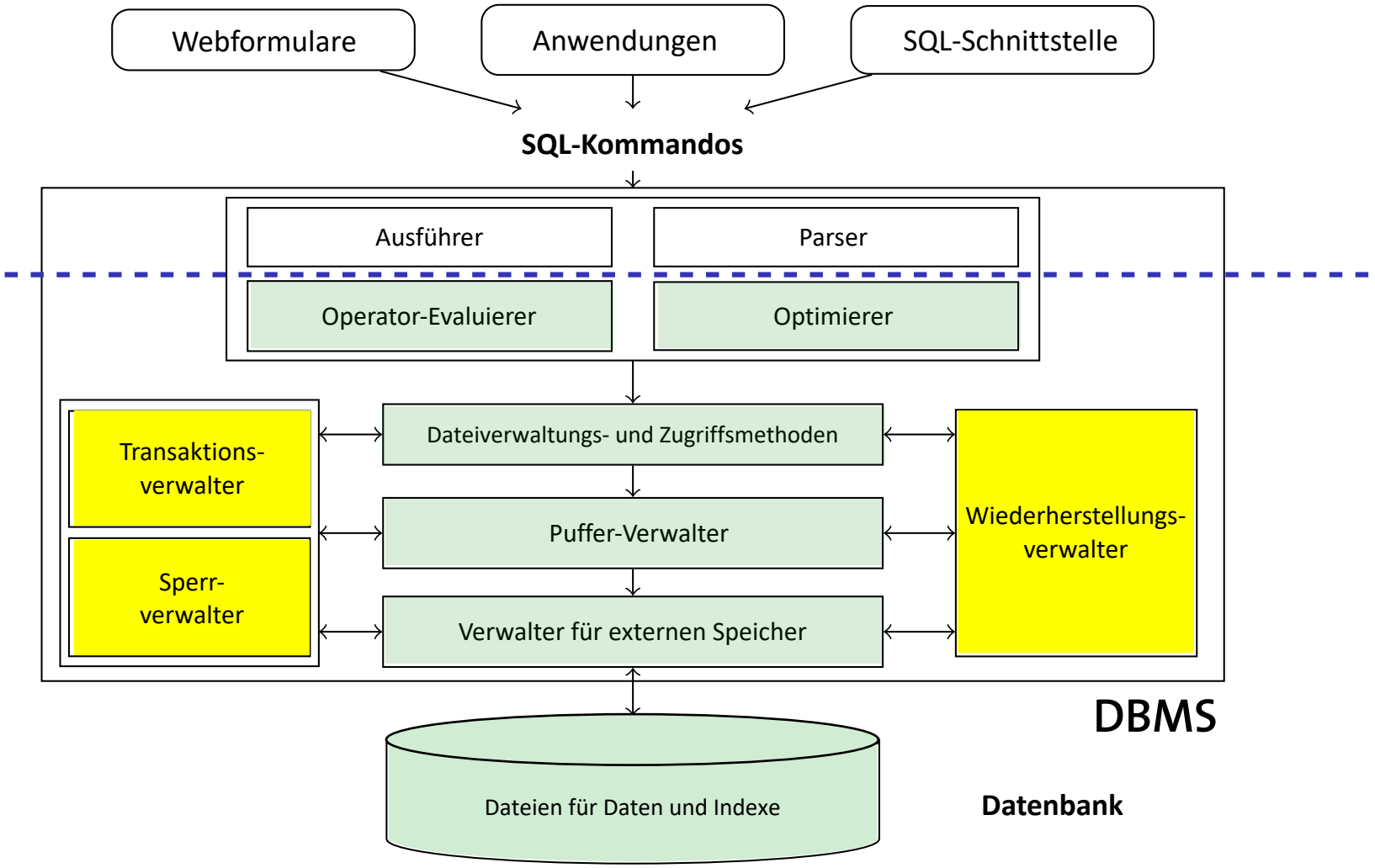
- Transaktionen, ACID
- Transaktionsverarbeitung
  - Algorithmen
  - Probleme/Fehlersituationen
- Schedules
  - Korrektheit
  - Serialisierbarkeit
  - Äquivalenzen
- Sperren, Verklemmung
  - Klassisch
  - Zeitstempel
  - Multiversionen?
- Logging/Recovery

Bezug zu Phasen des DB-Entwurfs

Kompetenzen

- Transaktionskonzept von Datenbanken kennen und einsetzen können
- Probleme beim Datenzugriff im Mehrbenutzerbetrieb verstehen und erkennen
- Algorithmen zum sicheren Datenzugriff im Mehrbenutzerbetrieb kennen
- Grundsätzlich verstehen, wie relationale Datenbanken die Dauerhaftigkeit (Durability) sicherstellen

Architektur eines DBMS



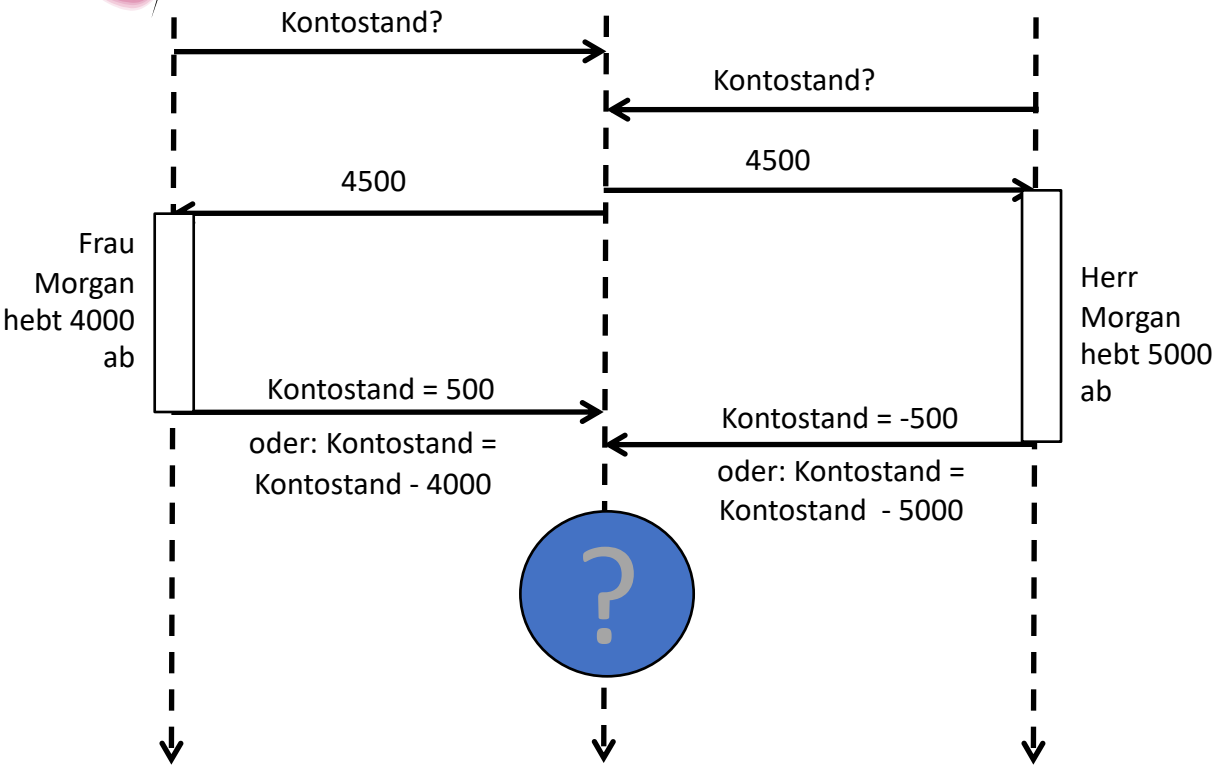
Übersicht

- Transaktionen
  - ACID-Eigenschaften und Transaktionsnotationen
  - Basisoperationen zur Transaktionsverarbeitung
- Probleme und Fehlersituationen bei der Transaktionsverarbeitung
- Ausführungspläne/Schedules
  - Korrektheit von Schedules und Serialisierbarkeit
  - Konfliktäquivalenz und Konfliktserialisierbarkeit
- Sperren
- Wiederherstellungsverwaltung

ACID, Notation, Basisoperationen

Transaktionsverarbeitung

Viele gleichzeitige Benutzer...



DBMS: Eigenschaften von Transaktionen

- Transaktion
  - Zusammenhängende Abfolge von Datenbank-Befehlen
  - Z.B. Kontostand abfragen, Überweisung von Konto 1 auf Konto 2

- **ACID**-Eigenschaften (kurz, später im Kapitel „Transaktionen“ mehr)
  - **A**tomicity (Atomarität):
    Alles oder nichts
  - **C**onsistency (Konsistenz):
    Vorher OK, hinterher OK
  - **I**solation (Isolation):
    Jeder denkt, er sei alleine auf der DB
  - **D**urability (Dauerhaftigkeit):
    Transaktionen bestätigt? Dann sind die Daten jetzt sicher

Es ist später!

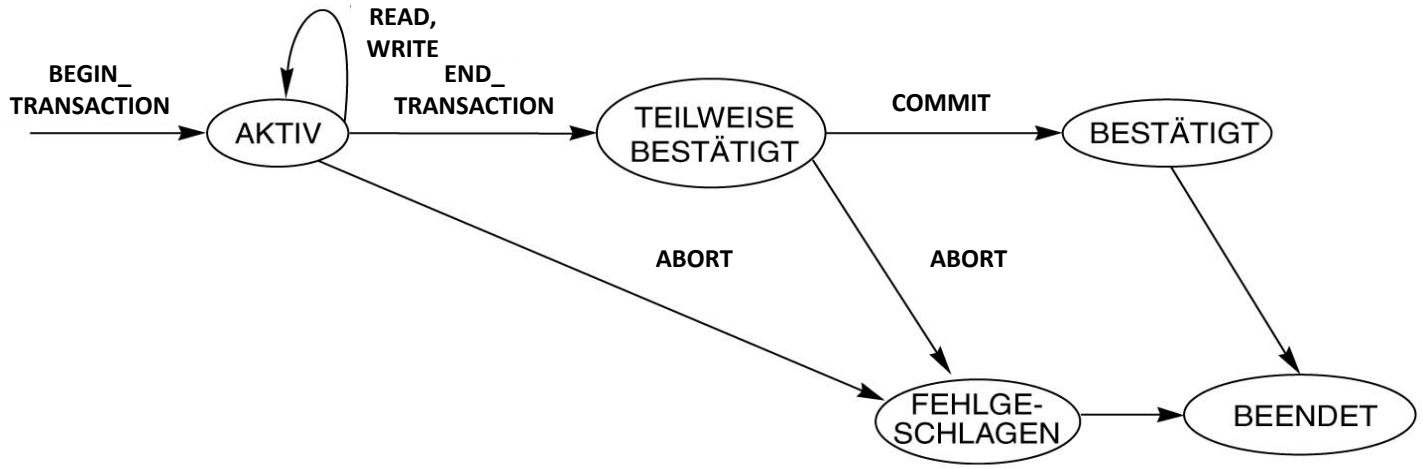
Insbesondere die Isolation-Eigenschaft macht klar, dass man Transaktionen nicht „unkontrolliert“ nebeneinander ablaufen lassen kann

DB-Transaktion - Definition

- Eine (DB-)Transaktion ist eine logische Verarbeitungseinheit auf einer DB, die typischerweise mehrere DB-Operationen (Lesen, Einfügen, Aktualisieren, Löschen) enthält
- Transaktionen …
  - können aus Anwendungsprogrammen heraus oder im Rahmen manueller Eingaben über eine SQL-Schnittstelle realisiert werden
  - werden mit einem speziellen Schlüsselwort gestartet (BEGIN_TRANSACTION) und beendet (END_TRANSACTION)
  - 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
- 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



Transaktionszustandsdiagramm

Lese- und Schreib-Operationen

- Für diese Lerneinheit: Datenbank = Sammlung von Datenobjekten
  - Wir abstrahieren von deren Größe (Granularität)
  - Datenobjekt kann Feld (Attribut) eines Datensatzes (Tupels), ein gesamter Datensatz oder ein Plattenblock (Zusammenfassung vieler Datensätze) sein
- Eine Transaktion kann die folgenden grundlegenden DB-Zugriffs-operationen beinhalten:
  - 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

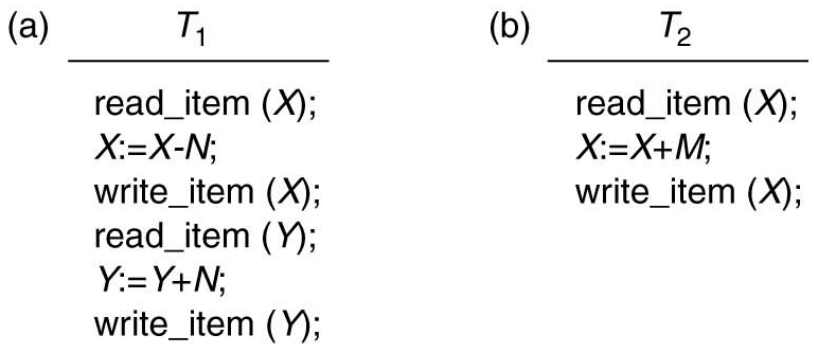
Realisierung von Read und Write

- 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.

READ- und WRITE-Set

- Eine Transaktion umfasst i.Allg. eine Menge unterschiedlicher READ_ITEM- und WRITE_ITEM-Anweisungen
  - Read-Set einer Transaktion:
    Menge der von einer Transaktion gelesenen Datenobjekte
  - Write-Set einer Transaktion:
    Menge der von einer Transaktion geschriebenen Datenobjekte
- Beispiel:
  - Read-Set von T_1 ist {X,Y}, Write-Set von T_1 ist {X,Y}
  - Read-Set von T_2 ist {X}, Write-Set von T_2 ist {X}



Fehlerszenarien

Transaktionsverarbeitung

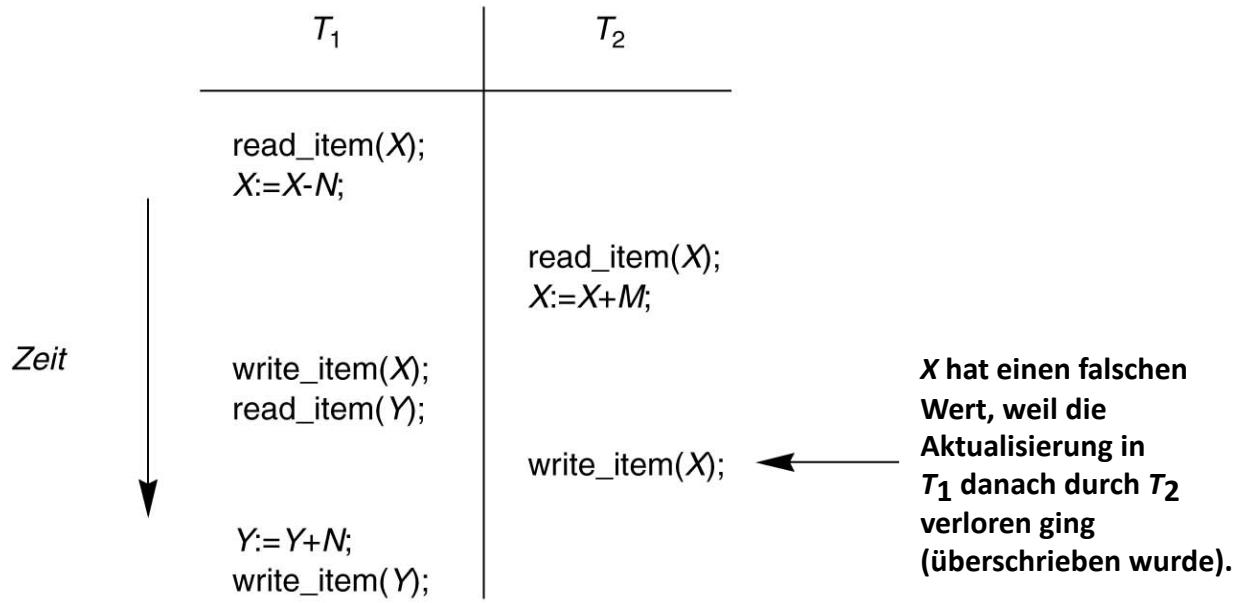
Fehlermöglichkeiten

- 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

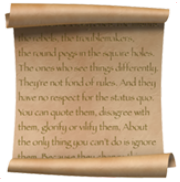
- Für die vier grundsätzlichen Fehlersituationen folgen je ein Beispiel zur Erläuterung. Die Fehler entstehen je nach konkreter zeitlicher Abfolge der einzelnen Operationen in nebenläufigen Transaktionen.

LOST UPDATE

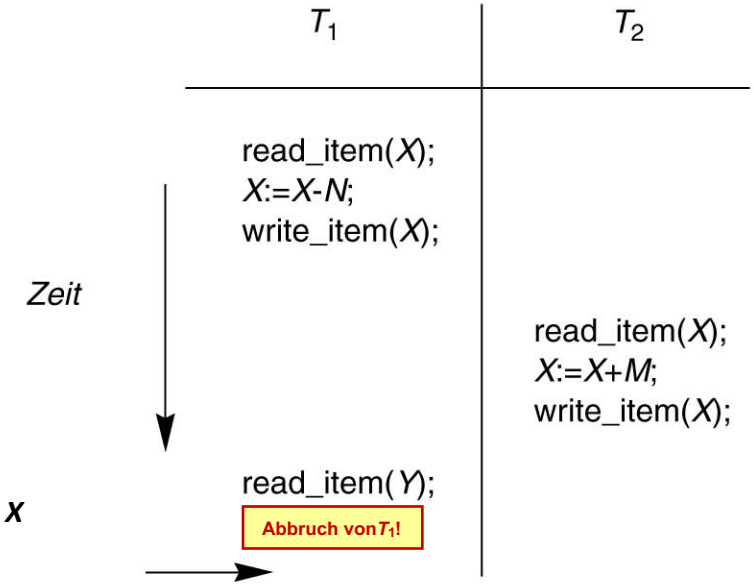
- Operationen zweier Transaktionen greifen auf die gleichen Datenobjekte zu
- Dabei überschneiden sie sich zeitlich derart, dass einzelne durchgeführte Aktualisierungen der Datenobjekte verloren gehen



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 durchgeführte Aktualisierung in T_1 verworfen wird

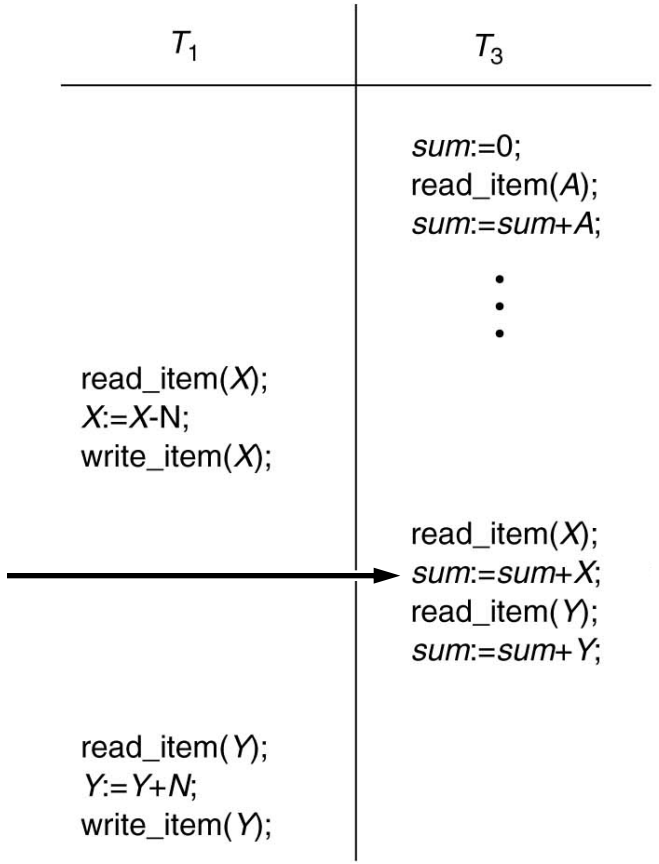


Transaktion T_1 schlägt fehl; der Wert von *X* muss auf den alten Wert zurückgesetzt werden; inzwischen hat T_2 aber den falschen Wert von *X* gelesen und damit „gerechnet“.

GHOST UPDATE

- 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

T_3 liest *X*, nachdem *N* subtrahiert wurde, und *Y*, bevor *N* addiert wird; dies ergibt ein um *N* falsches Resultat.



UNREPEATABLE READ

- Ein 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

T	T'
sum:=0;	
read_item(*A*);	
sum:=*sum*+*A*;	
.	
.	
.	
read_item(*X*);	
sum:=*sum*+*X*;	
.	read_item(*X*);
.	*X*:=*X*-N;
.	write_item(*X*);
read_item(*X*);	

Hier hat *X* einen um *N* niedrigeren Wert, als wenn der Read-Befehl vor Start von Transaktion *T'* erfolgt wäre.

Schedules

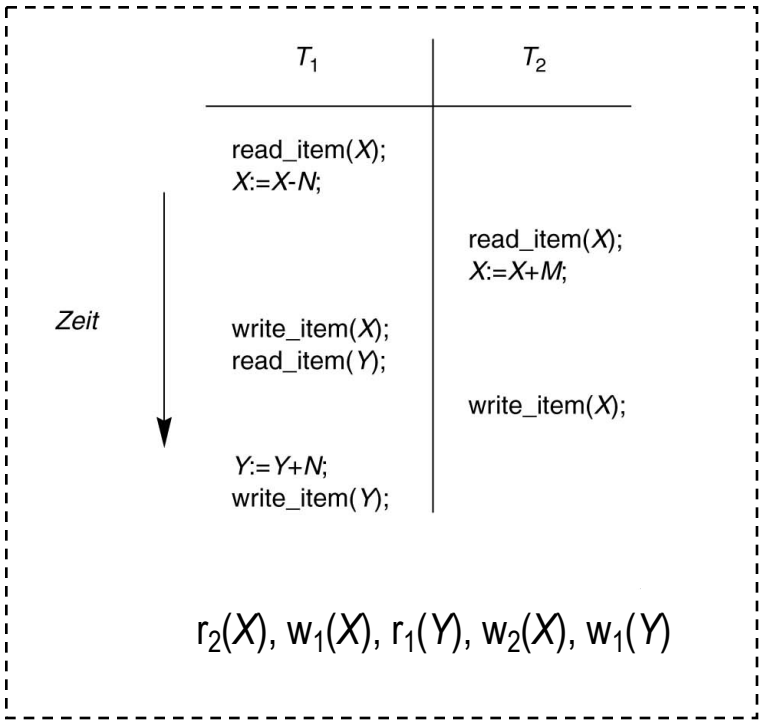
Transaktionsverarbeitung

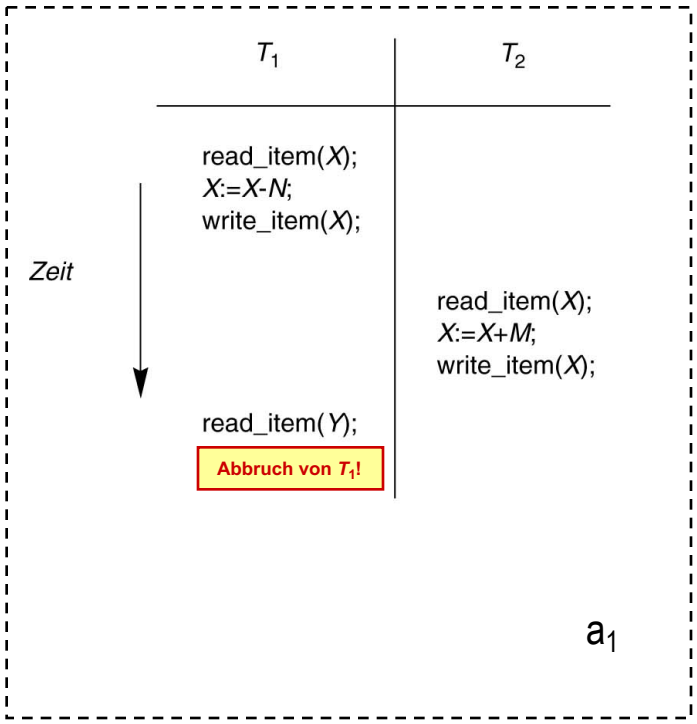
Schedules von Transaktionen

- Ausführungsplan / Schedule:
  - Integrierte Abfolge der Operationen überlappend ausgeführter Transaktionen
  - Ein Schedule S von n Transaktionen T_1, …, 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 – auch wenn zwischen ihnen Operationen anderer Transaktionen liegen

- Abkürzungen für die folgenden Folien:
  - Operation 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





Konflikte bei Operationen

- 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 ($p_i(X) = w_i(X)$ v $p_j(X) = w_j(X)$)

  → Reihenfolge von $p_i(X)$, $p_j(X)$ ist relevant und muss spezifiziert werden
- Ein Schedule S mit n Transaktionen $T_1, \ldots, T_n$ ist ein vollständiger Schedule, wenn folgende Bedingungen gelten:
  - Die Operationen in S sind exakt diejenigen aus $T_1, \ldots, T_n$, einschließlich einer COMMIT- oder ABORT-Operation, die jede Transaktion abschließt
  - Jedes Paar von Operationen der gleichen Transaktion T_i kommt in S und in T_i in der gleichen Reihenfolge vor
  - Für jeweils zwei konfliktäre Operationen muss eine der beiden explizit vor der anderen im Schedule stehen

Nicht in Konflikt stehende Operationen können auch parallel ausgeführt werden (solche Schedules heißen partiell geordnet).

Projektion auf vollständige Transaktionen

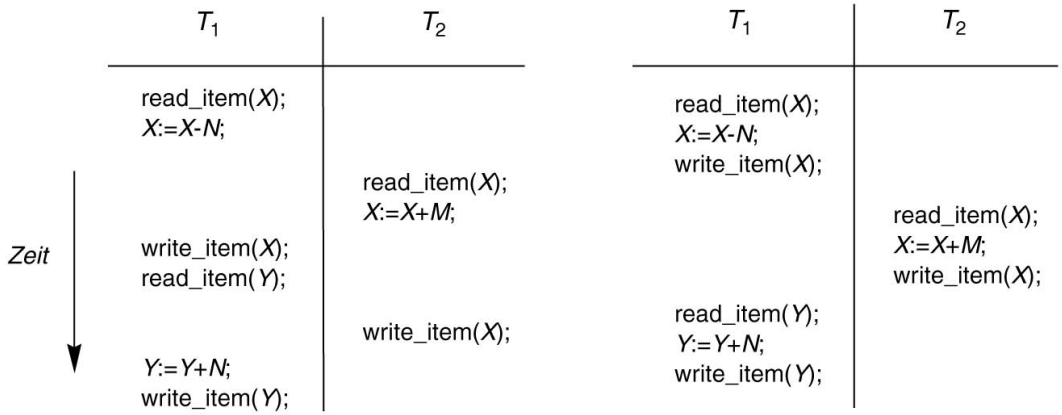
- Vollständige Schedules treten in Transaktionverarbeitungssystemen selten auf, da ständig neue Transaktionen an solchen Systemen eintreffen (und damit Vollständigkeit wegen nicht abgeschlossener Transaktionen verhindern).
- C(S) ist die Projektion eines Schedules S, die um Operationen nicht bestätigter Transaktionen in S bereinigt ist.
  - Alle in S integrierten Transaktionen T_i sind in C(S) analog (komplett und in gleicher Reihenfolge) wiedergegeben und mit einem COMMIT (c_i) abgeschlossen.
- Im Folgenden werden wir Transaktionsverarbeitungssysteme (Scheduler, Transaktionsmonitore) mit bestätigten Transaktionen betrachten.

Korrekte Schedules

- Bisher nur „Vollständigkeit“, nicht aber Korrektheit betrachtet!
- Ein Schedule ist korrekt, wenn die enthaltenen Transaktionen „problemlos“ ausgeführt werden können, d.h. zum gleichen Ergebnis führen wie eine streng sequenzielle Ausführung der jeweils vollständigen Transaktionen.
- Enthält ein Schedule S z.B. die Transaktionen T_1 und T_2, die jeweils für sich „ohne Unterbrechung“ ausgeführt werden, so ist deren problemlose Ausführung auf triviale Weise gewährleistet, da sie sich „nicht stören“:

Korrektheit verschachtelter Transaktionen

- Wird aus Performance-Gründen die verschachtelte (nicht streng sequenzielle) Ausführung von Operationen verschiedener Transaktionen eines Schedules zugelassen, so ist die Korrektheit der verarbeiteten Schedules zu gewährleisten.
- Verschachtelte Schedules sind z.B.



Schedule *C*

Schedule *D*

- Das nachfolgend vorgestellte Konzept der Serialisierbarkeit dient zur Prüfung von Schedules auf ihre Korrektheit.

Serialisierbarkeit

- Definition: Ein Schedule S ist seriell, wenn die Operationen jeder in S enthaltenen Transaktion T vollständig hintereinander (streng sequenziell) ausgeführt werden. Andernfalls ist ein Schedule nicht-seriell.
- Definition: Ein Schedule S mit n Transaktionen ist serialisierbar, wenn er zu einem seriellen Schedule S' äquivalent ist, d.h. den gleichen DB-Zustand erreicht.

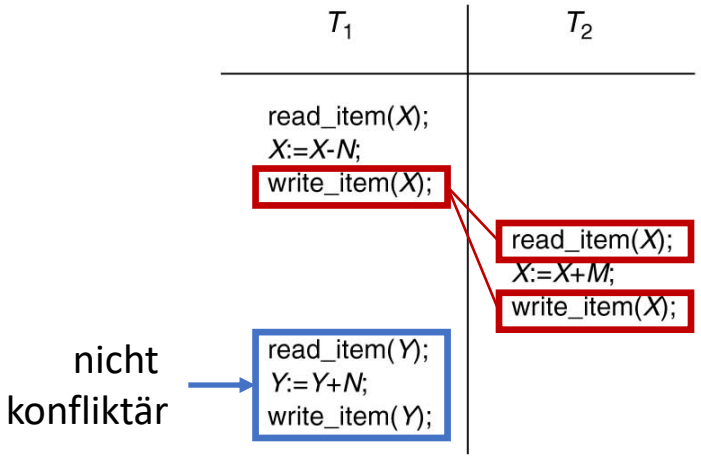
- Serialisierbarkeit: Worin liegt der Gewinn dieser Überlegung?
  - Serialisierbar bedeutet korrekt – bzw. äquivalent zu einem korrekten Schedule
  - Frage 1 (in dieser Vorlesung): Wie erhält man serielle Schedules?
  - Frage 2 (weiterführend): Welche der serialisierbaren Schedules sind effizient ausführbar, bringen also einen großen „Durchsatz“ der Transaktionen?

Äquivalenz von Schedules

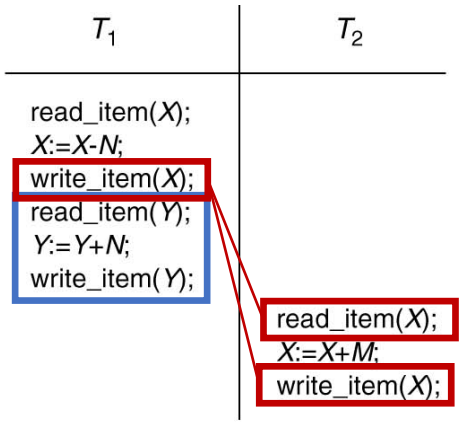
- Es gibt unterschiedliche Möglichkeiten, um die Äquivalenz von Schedules auszudrücken.
- Häufig wird die Konflikt-Äquivalenz genutzt
  - Andere Varianten existieren, z.B. View-Äquivalenz, da Konfliktäquivalenz sehr strikt ist
- Definition – Konfliktäquivalenz
  - Zwei Schedules sind konfliktäquivalent, wenn die Reihenfolge von je zwei in Konflikt stehenden Operationen in beiden Schedules gleich ist.
- Definition – Konfliktserialisierbarkeit
  - Ein Schedule S ist konfliktserialisierbar, wenn S mit einem serialisierbaren Schedule S' konfliktäquivalent ist.

Äquivalenz von Schedules

- Im Fall der Konfliktserialisierbarkeit von S können die nicht konfliktären Operationen so lange umgestellt werden, bis ein äquivalenter serieller Schedule S' vorliegt.
  - Der folgende **Schedule D** ist konfliktserialisierbar.
  - Konfliktär: $w_1(X)$ und $r_2(X)$ (gilt auch für $w_1(X)$ und $w_2(X)$)
  - Der zu **D** konfliktäquivalente **Schedule A** ist rechts gezeigt:



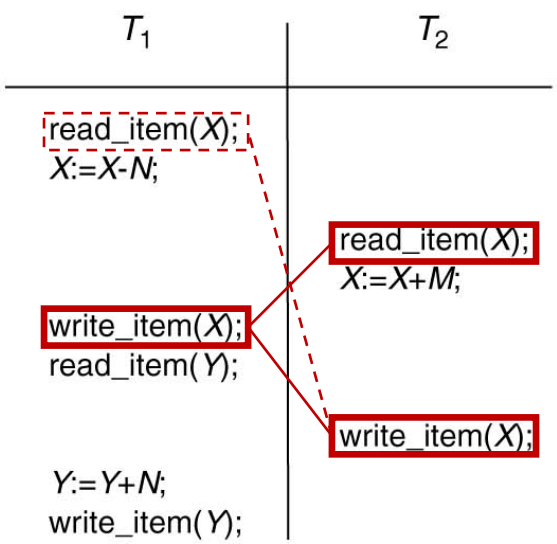
Schedule *D*



Schedule *A*
(seriell; $T_1 | T_2$)

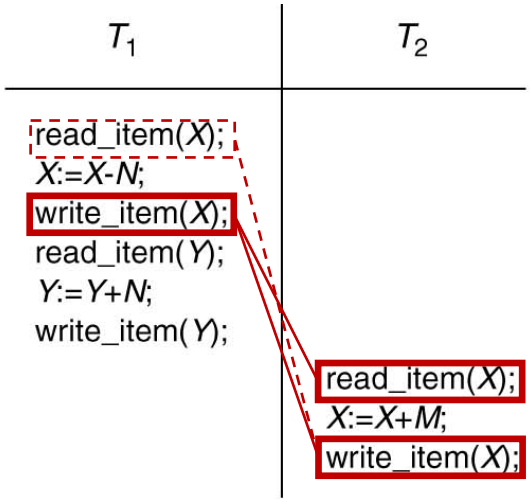
Äquivalenz von Schedules

- Der folgende Schedule C ist nicht konfliktserialisierbar, da C zu keinem serialisierbaren Schedule äquivalent ist
  - C kann insbesondere nicht in serielle Schedules A oder B überführt werden, d.h. er ist nicht korrekt



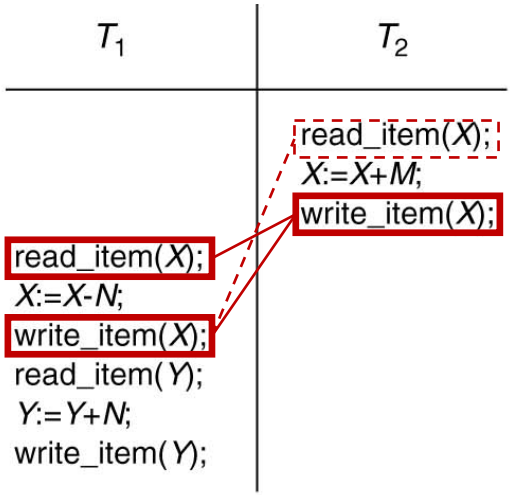
Schedule *C*

$r_2(X), w_1(X), w_2(X)$



Schedule *A*
(seriell; $T_1|T_2$)

$w_1(X), r_2(X), w_2(X)$



Schedule *B*
(seriell; $T_2|T_1$)

$w_2(X), r_1(X), w_1(X)$

Prüfung der Konfliktserialisierbarkeit

- Algorithmus auf Basis des sog. Serialisierungsgraph
- Ein Serialisierungsgraph SG für ein Schedule S ist ein gerichteter Graph G=(N,E), wobei
  - $N=\{T_1, \ldots, T_n\}$ Menge von Knoten
  - $E=\{e_1, \ldots, e_m\}$ Menge gerichteter Kanten
- Für jede Transaktion T_i in S enthält der Graph einen Knoten
- Jede Kante e_i im Graphen hat die Form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, wobei T_j der Start- und T_k der Endknoten von e_i ist.
- Eine Kante $T_j \rightarrow T_k$ wird erzeugt, wenn in S eine der Operationen von T_j vor einer mit ihr in Konflikt stehenden Operation in T_k vorkommt.

Serialisierbarkeitstheorem:
Schedule S ist konfliktserialisierbar, wenn der zugehörige Serialisierungsgraph keine Zyklen aufweist.

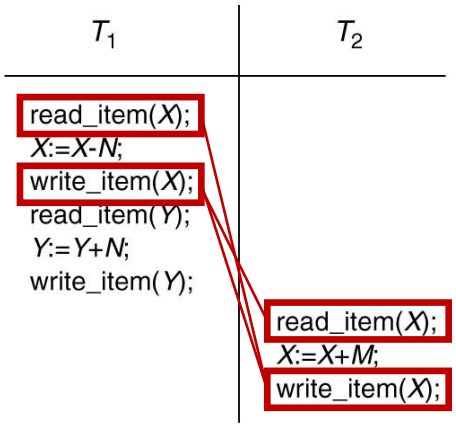
Prüfung der Konfliktserialisierbarkeit

Algorithmus zum Test auf Konfliktserialisierbarkeit:

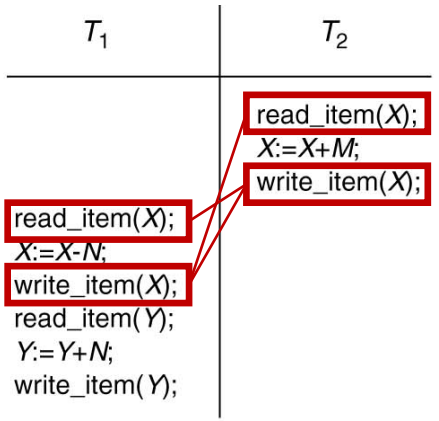
- Eingabe: Schedule S
- Ausgabe: Konfliktserialisierbar ja/nein
- Algorithmus:
  1. Baue einen Serialisierungsgraph SG für S
     a) Erzeuge für jede Transaktion T_i, die in S auftritt, einen Knoten T_i im Serialisierungsgraph SG.
     b) Erzeuge für jeden Fall in S, bei dem erst T_i ein WRITE_ITEM(X) ausführt und dann T_j ein READ_ITEM(X) ausführt ($w_i(X), r_j(X)$), eine Kante $(T_i \rightarrow T_j)$ in SG.
     c) Erzeuge für jeden Fall in S, bei dem erst T_i ein READ_ITEM(X) ausführt und dann T_j ein WRITE_ITEM(X) ausführt ($r_i(X), w_j(X)$), eine Kante $(T_i \rightarrow T_j)$ in SG.
     d) Erzeuge für jeden Fall in S, bei dem erst T_i ein WRITE_ITEM(X) ausführt und dann T_j ein WRITE_ITEM(X) ausführt ($w_i(X), w_j(X)$), eine Kante $(T_i \rightarrow T_j)$ in SG.
  2. Prüfe, ob SG keinen Zyklus enthält.
     a) Wenn kein Zyklus vorhanden: S ist konfliktserialisierbar
     b) Sonst: S ist nicht konfliktserialisierbar

Beispiele

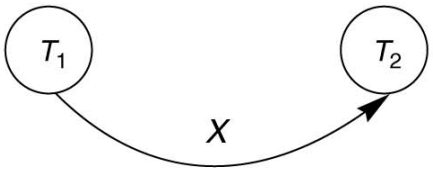
- Schedules mit zugehörigen Serialisierungsgraphen:

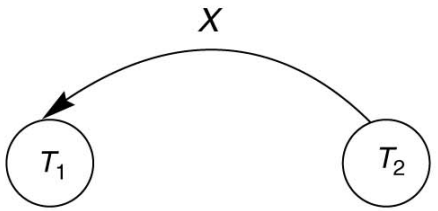


Schedule *A*



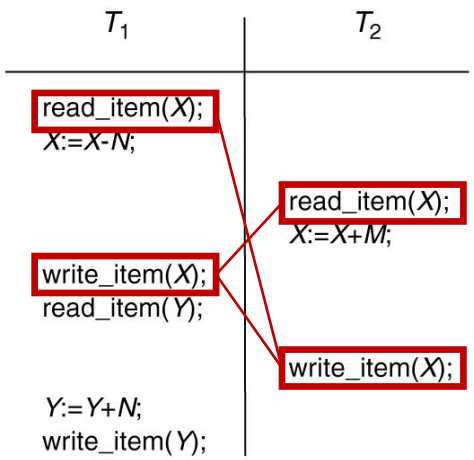
Schedule *B*



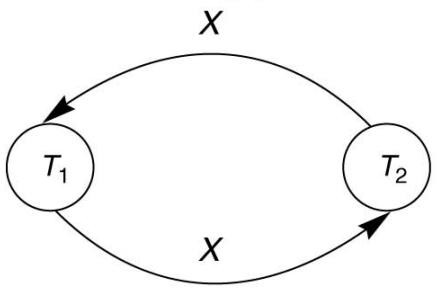


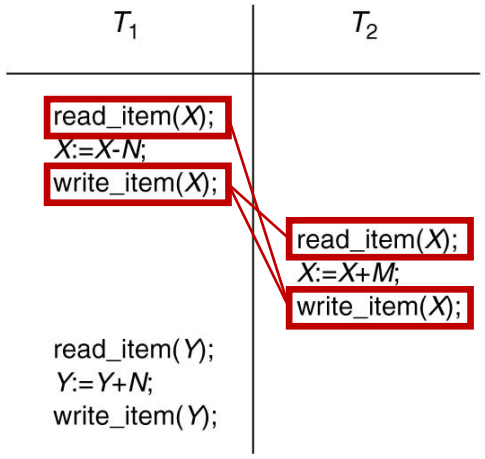
Beispiele

- Schedules mit zugehörigen Serialisierungsgraphen:

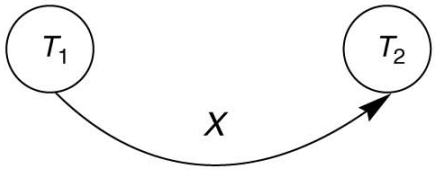


Schedule *C*



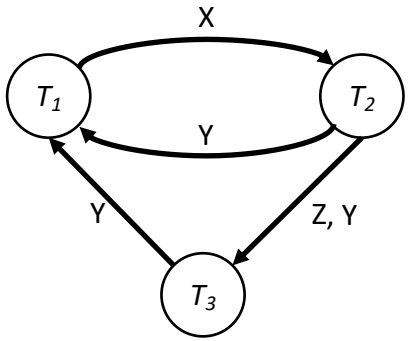


Schedule *D*



Beispiele

- Serialisierungsgraph
  - Aus Gründen der Lesbarkeit nur erste Vorkommen einer Kante im Schedule markiert

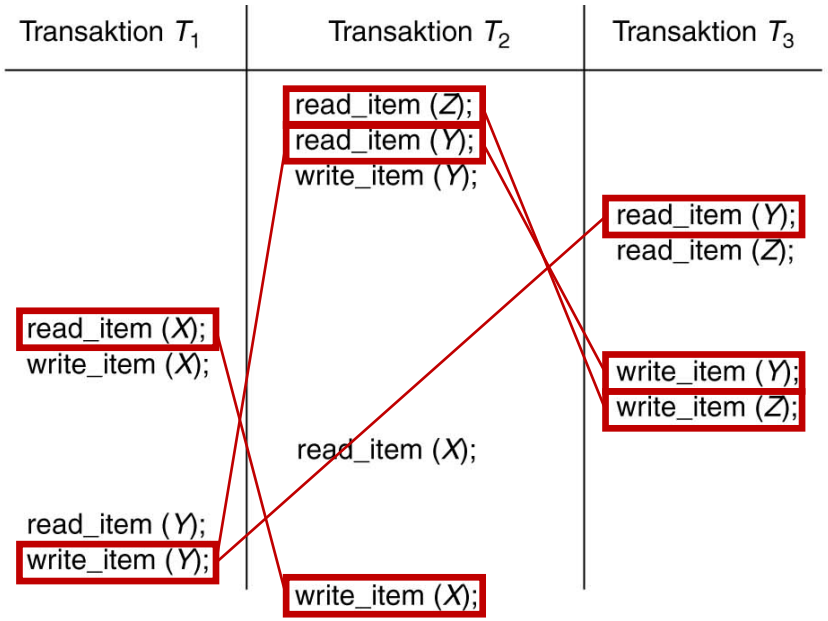


- Zyklen in Graph, z.B.
  - $T_1 \rightarrow T_2 \rightarrow T_1$
  - $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$

→ Nicht konfliktserialisierbar

  - D.h. keine Ausführungspläne von E, die äquivalent zu einem seriellen Schedule sind

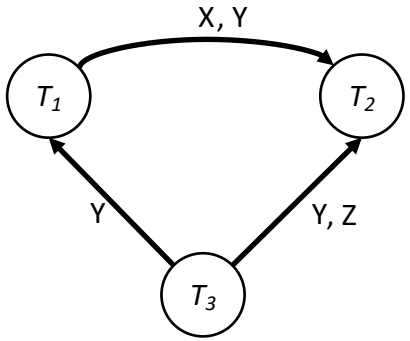
Transaktion T_1	Transaktion T_2	Transaktion T_3
read_item (X); write_item (X); read_item (Y); write_item (Y);	read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);	read_item (Y); read_item (Z); write_item (Y); write_item (Z);



Schedule *E*

Beispiele

- Serialisierungsgraph
  - Aus Gründen der Lesbarkeit nur erste Vorkommen einer Kante im Schedule markiert

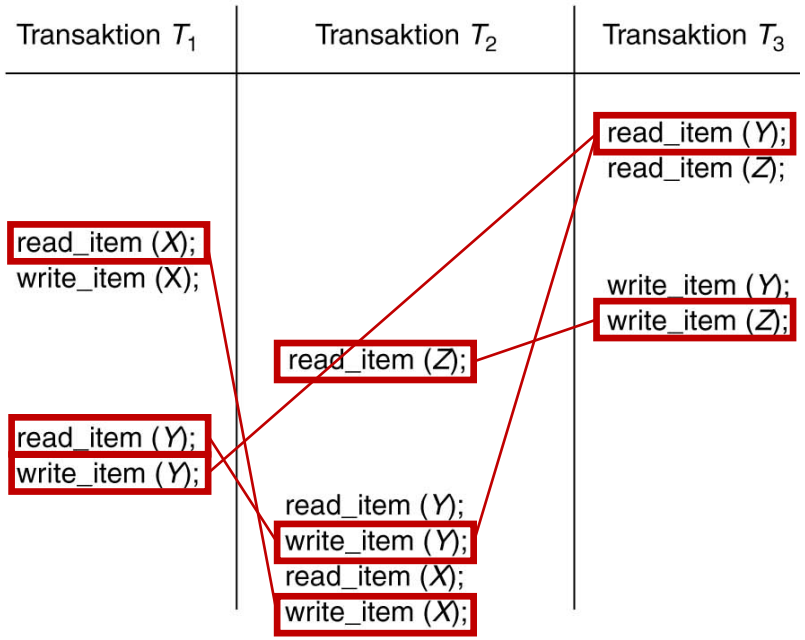


- Keine Zyklen in Graph

→ Konfliktserialisierbar

  - Äquivalenter, serieller Ausführungsplan zu F: $T_3 \mid T_1 \mid T_2$

Transaktion T_1	Transaktion T_2	Transaktion T_3
read_item (*X*); write_item (*X*); read_item (*Y*); write_item (*Y*);	read_item (*Z*); read_item (*Y*); write_item (*Y*); read_item (*X*); write_item (*X*);	read_item (*Y*); read_item (*Z*); write_item (*Y*); write_item (*Z*);



Schedule *F*

Zwischen-Rückblick

- Transaktionen
  - ACID-Eigenschaften (Atomicity, Consistency, Isolation und Durability)
  - Verschiedene Operationen und Zustände während der Transaktionsverarbeitung
    - READ_ITEM(X), WRITE_ITEM(X)
    - BEGIN_TRANSACTION, END_TRANSACTION
    - COMMIT, ABORT
- Probleme bei der Transaktionsverarbeitung
  - Lost Update, Dirty Read, Ghost Update, Unrepeatable Read
  - Weitere Fehlersituationen
- Schedules
  - Vollständigkeit
  - Korrektheit
  - Serielle Schedules
  - Serialisierbarkeit
- Konfliktäquivalenz und Konfliktserialisierbarkeit
  - Serialisierungsgraph und Test

Übersicht

- Transaktionen
- Schedules
- Sperren
  - Konzept von Sperren und Sperrprotokolle
    - Sperrverwaltung
    - Sperrtypen
    - Binäre Sperren
    - Mehrfachmodussperren
    - Zwei-Phasen-Sperrprotokoll
  - Vermeidung und Erkennung/Behandlung von Verklemmungen
  - Weitere Verfahren zur Transaktionsverarbeitung
    - Zeitstempelbasierte Transaktionsverarbeitung
    - Multiversionsprotokolle
    - Optimistische Verfahren zur Nebenläufigkeitskontrolle
    - Granularitäten, Vorhaben-Sperren, Isolationsmodi
  - Sperren in Indexbäumen
- Logging/Recovery

Sperren und Sperrprotokolle

Transaktionsverarbeitung

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
  - Der Verwalter 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)
  - Eine Transaktion, die eine Sperre erhält, wird fortgesetzt
- Sperren regeln die relative Ordnung der Einzeloperationen verschiedener Transaktionen
- Ziel: Automatisierte Generierung von konfliktserialisierbaren 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
- Wie muss eine Datenstruktur aussehen, mit der diese Anforderungen erfüllt werden können?
  - Datenstruktur zur Speicherung der Lock-Information zu jedem Objekt
  - Meist als Hash-Tabelle organisiert, um effizient auf Lock-Informationen zugreifen zu können → Sperrtabelle

Protokolle und Sperren

- Protokollen 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“
  - Lock(X) = 1: Objekt ist gesperrt
  - Lock(X) = 0: Objekt ist entsperrt
- Operationen:
  - LOCK_ITEM … sperrt das Objekt
  - UNLOCK_ITEM … 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:
    (Datenobjekt_ID, LOCK_Zustand, Transaktions_ID)

Binäre Sperren

- Protokoll mit binären Sperren nach folgenden Regeln für jede Transaktion T:
  1. Vor READ_ITEM(X) und WRITE_ITEM(X):
     → LOCK_ITEM(X)
  2. Nach allen READ_ITEM(X) und WRITE_ITEM(X):
     → UNLOCK_ITEM(X)
  3. Kein LOCK_ITEM(X) durch T, wenn T schon eine Sperre auf X hat
  4. Nur dann 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) = „Lesesperre“ :
    - Auf X wird lesend zugegriffen ("shared") → Als Sperrmodus S bezeichnet
    - Weiterer Lesezugriff möglich, Schreibzugriff potenziell problematisch

  - LOCK(X) = „Schreibsperre“ :
    - Auf X wird schreibend zugegriffen ("exklusive") → Als Sperrmodus X bezeichnet
    - Auf X kann durch andere Transaktion nicht zugegriffen werden

  - LOCK(X) = „entsperrt“ :
    - Auf X wird nicht zugegriffen

- Kompatibilitätsmatrix
  - Zustand: Gibt es eine Sperre?
  - Anforderung: Was für eine soll gesetzt werden?

Anforderung \ Zustand	*Keine Sperre*	S	X
S	✓	✓	–
X	✓	–	–

- Operationen:
  - READ_LOCK(X), WRITE_LOCK(X) und UNLOCK(X)

Sperren mit Mehrfachmodus

- Mögliche Implementierung:
  - Einträge in der Sperrtabelle:
    (Datenobjekt_ID, LOCK, #Zugriffsoperationen, [Transaktion_ID, …])
  - Idee: Zählen der lesenden Transaktionen (#Zugriffsoperationen)
  - Bei schreibenden Transaktionen (LOCK(X)=„Schreibsperre“) ist #Zugriffsoperationen = 1
- Regeln für Transaktion T:
  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 irgendeine Sperre auf X durch T besteht
  5. Kein WRITE_LOCK(X), falls irgendeine Sperre auf X durch T besteht
  6. Nur dann UNLOCK_ITEM(X), wenn eine Sperre auf X durch T besteht

Sperren mit Mehrfachmodus - Sperrenänderung

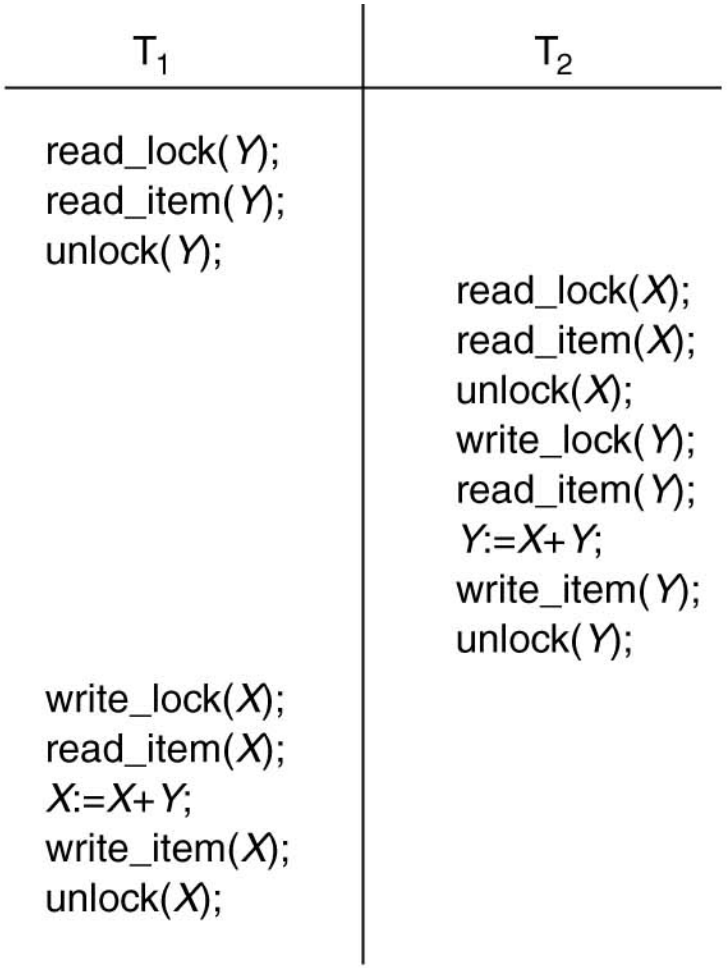
- Für weniger restriktive Lock-Mechanismen Änderung von Regeln 4 und 5 → Sperrenänderung
- Damit kann eine Schreibsperre zur Lesesperre gelockert oder eine Lesesperre zur Schreibsperre verschärft werden:
    1. READ_LOCK(X) oder WRITE_LOCK(X) anstoßen vor irgendwelchen READ_ITEM(X)
    2. WRITE_LOCK(X) 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_ITEM(X), wenn eine Sperre auf X durch T besteht

Sperren - Bewertung

- Binäre und Mehrfach-Sperren garantieren selbst noch keine Serialisierbarkeit

- 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

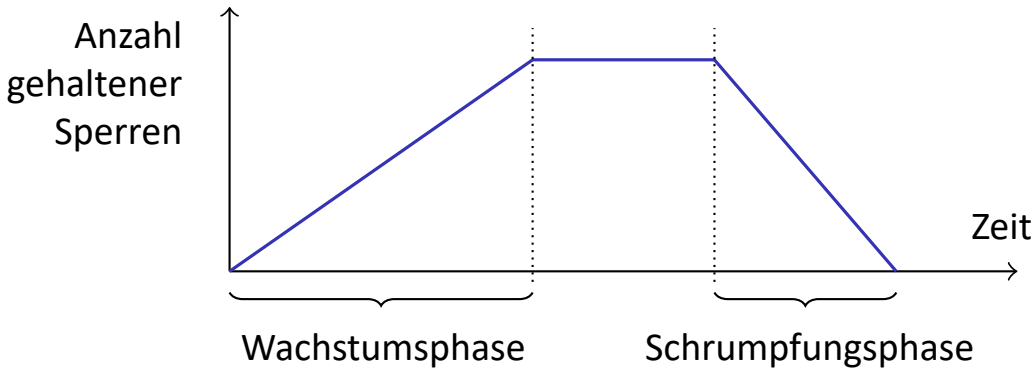
→ Besseres Protokoll gesucht!



Schedule *S*

Zwei-Phasen-Sperrprotokoll

- Idee: die Sperroperationen aller Objekte werden vor der ersten **Ent**sperroperation 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 (Lesesperre zu Schreibsperre)
  - Lockerungen nur in Schrumpfungsphase (Schreibsperre zu Lesesperre)
- Schedules, die dem Zwei-Phasen-Sperrprotokoll folgen, sind serialisierbar:
  - Schreiboperationen sind durch exklusive Sperren „abgesichert“

Beispiel

nicht Zwei-Phasen-Sperrprotokoll

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

- T_1 : write_lock(X) nach unlock(Y)
- T_2 : write_lock(Y) nach unlock(X)

Zwei-Phasen-Sperrprotokoll

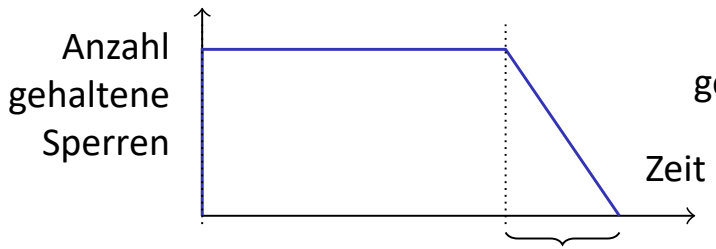
T_1'	T_2'
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

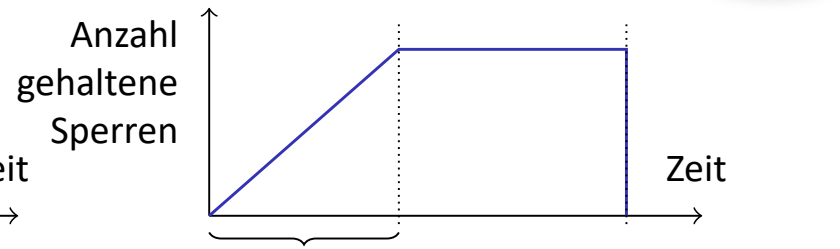
- Kann Verklemmung produzieren!
  - Verklemmung: beide Transaktionen sind aktiv und warten auf Sperre, z.B.
    - T'_2 wartet auf unlock(Y) von T'_1, T'_1 wartet auf unlock(X) von T'_2

Varianten des Zwei-Phasen-Sperrprotokolls

- Konservatives Zwei-Phasen-Sperrprotokoll (mit Voranforderung)
  - T sperrt erst alle Objekte, bevor sie mit der Verarbeitung beginnt
  - Problem: alle benötigten Sperren müssen vorab bekannt u. deklariert sein
- Striktes Zwei-Phasen-Sperrprotokoll
  - T hebt Schreibsperren erst dann auf, wenn T bestätigt/abgebrochen wurde
  - In der Praxis recht verbreitet, garantiert aber nicht Verklemmungsfreiheit
- Rigoroses Zwei-Phasen-Sperrprotokoll
  - T hebt Lese- und Schreibsperren erst auf, wenn T bestätigt/abgebrochen wurde
  - Einfacher umzusetzen als die strikte Variante, verhindert aber auch nicht Verklemmungen.

Was sind Beweggründe für die Varianten?





Beispiel

T_1

read_lock$_1$(X)
read_item$_1$(X)
read_lock$_1$(Y)
read_item$_1$(Y)
read_lock$_1$(Z)
read_item$_1$(Z)
$S := X + Y + Z$
unlock$_1$(X)
unlock$_1$(Y)
unlock$_1$(Z)

T_2

write_lock$_2$(Y)
read_item$_2$(Y)
$Y := Y - 100$
write_lock$_2$(Z)
read_item$_2$(Z)
$Z := Z+100$
write_item$_2$(Y)
write_item$_2$(Z)
unlock$_2$(Y)
unlock$_2$(Z)

T_1	T_2	X	Y	Z
begin_transaction		free	free	free
read_lock$_1$(X)		1:read		
read_item$_1$(X)				
	begin_transaction			
	write_lock$_2$(Y)		2:write	
	read_item$_2$(Y)			
read_lock$_1$(Y)			1:wait	
	$Y := Y - 100$			
	write_lock$_2$(Z)			2:write
	read_item$_2$(Z)			
	$Z := Z+100$			
	write_item$_2$(Y)			
	write_item$_2$(Z)			
	end_transaction			
	unlock$_2$(Y)		1:read	
read_item$_1$(Y)				
read_lock$_1$(Z)				1:wait
	unlock$_2$(Z)			1:read
read_item$_1$(Z)				
	commit			
$S := X + Y + Z$				
end_transaction				
unlock$_1$(X)		free		
unlock$_1$(Y)			free	
unlock$_1$(Z)				free
commit				

Probleme bei der Verwendung von Sperren

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

Verklemmungen

Transaktionsverarbeitung

Deadlock – Definition und Beispiel

- Ein Deadlock liegt vor, wenn jede Transaktion T einer Menge von zwei oder mehr Transaktionen auf ein Objekt wartet, das von einer anderen Transaktion T' der Menge gesperrt wurde
- Beispiel:
  - T_1 und T_2 sperren wechselseitig Objekte X und Y

T_1'	T_2'
read_lock(*Y*);	
read_item(*Y*);	
	read_lock(*X*);
	read_item(*X*);
write_lock(*X*);	
	write_lock(*Y*);

- Behandlung von Deadlocks
  - Vermeidung
  - Erkennung und Auflösung

Vermeidung von Deadlocks

- Konservatives Zwei-Phasen-Sperrprotokoll
  - 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)
  - TS(T) ist ein eindeutiger Identifikator für eine Transaktion, der als Zeitstempel (Timestamp) bezeichnet wird
    - I.d.R. 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

- Situation:
  - Eine Transaktion T' versucht ein Objekt zu sperren, dies ist bereits von einer anderen Transaktion T gesperrt
- Wait/Die
  - Wenn TS(T') < TS(T) : T' ist älter als T und wartet
  - Wenn TS(T') > TS(T) : T' ist jünger als T und stirbt
    - T' bricht sich selbst ab
    - T' startet später mit gleichem Zeitstempel TS(T') wieder
  - 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
    - T startet später mit dem gleichen Zeitstempel TS(T) wieder
  - Wenn TS(T') > TS(T) : T' ist jünger als T und wartet
  - T behält seine Sperre nur, wenn T älter ist
- Ältere Transaktionen werden bevorzugt

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 verklemmungsfrei
  - Erzeugen aber u.U. unnötige Abbrüche und Neustarts von Transaktionen, die nie einen Deadlock verursacht hätten → schlechtere Performanz

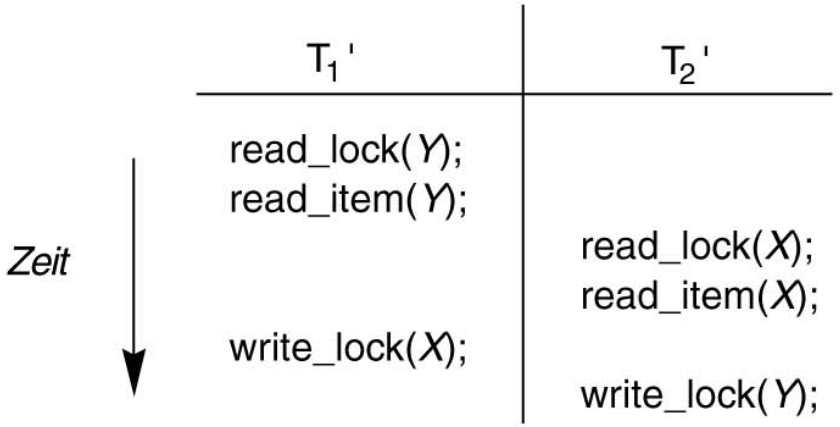
Erkennung von Deadlocks

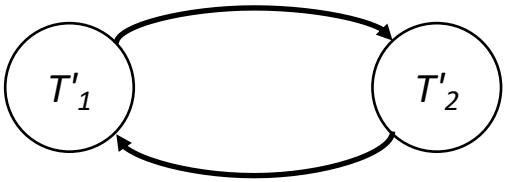
- Optimistisches Verfahren: statt Vermeidung
  nachträgliche Erkennung und Auflösung
  - Gut, wenn wenig Deadlocks zu erwarten sind
- Grundideen:
  - (Physische) Zeitbeschränkungen
  - (Logischer) Wartegraph

- Zeitbeschränkung:
  - 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
- Wartegraph …

Erkennung von Deadlocks: Wartegraph

- 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
- Weist der Wartegraph Zyklen auf, so liegt ein Deadlock vor.
  - Nicht verwechseln mit Zyklenfreiheit auf Serialisierungsgraph
- Beispiel
  - T'_1 startet mit read_lock(Y)
  - T'_2 startet mit read_lock(X)
  - T'_1 fordert write_lock(X) an
    - X gesperrt von T'_2: Kante $T'_1 \rightarrow T'_2$
  - T'_2 fordert write_lock(Y) an
    - Y gesperrt von T'_1: Kante $T'_2 \rightarrow T'_1$





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 – Definition und Beispiel

- Starvation: eine 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)
    - Eine durch das Transaktionsverarbeitungssystem abgebrochene Transaktion erhält eine höhere Priorität und 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

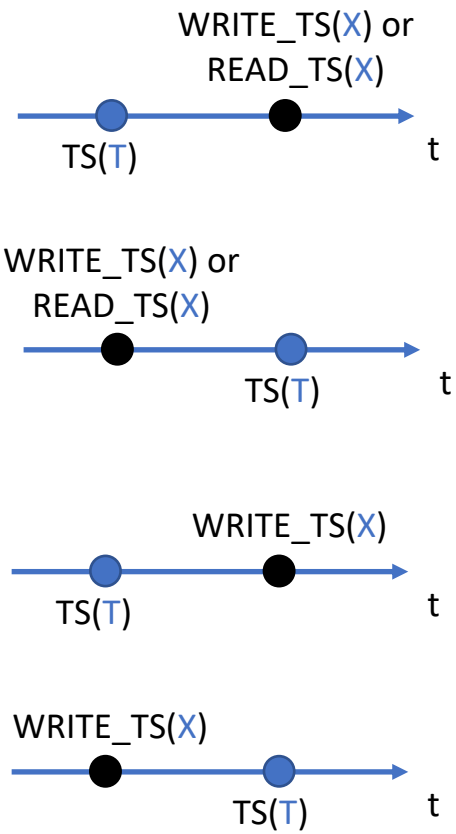
Transaktionsverarbeitung

Verfahren mit Zeitstempelung

- Zeitstempel TS(T) der Transaktion T
  - Wie vorher definiert:
    Eindeutiger Identifikator, der vom DBMS generiert wird, um eine Transaktion zu identifizieren
    - I.d.R. der Start-Zeitpunkt (Ablesen der internen Systemuhr) von T
- Zeitstempelbasierte Ansätze verwenden *keine Sperren* und zeigen *keine Verklemmungen*
- Idee: Algorithmus stellt sicher, dass der Zugriff von konfliktären Operatoren nicht die Reihenfolge (mindestens) eines äquivalenten seriellen Schedules verletzt
  - READ_TS(X): Lesezeitstempel von X
    - größter (aktuellster) TS(T) aller Transaktionen, die X erfolgreich gelesen haben
  - WRITE_TS(X): Schreibzeitstempel von X
    - TS(T) der Transaktion T, die X erfolgreich geschrieben hat

Basis-Zeitstempelordnung

- T will WRITE_ITEM(X) durchführen
  - Wenn READ_TS(X) > TS(T) oder WRITE_TS(X) > TS(T):
    - Jüngere Transaktionen haben X schon gelesen oder geschrieben
    - Operation wird abgewiesen, T wird abgebrochen („zu spät“)
  - Sonst:
    - WRITE_ITEM(X) wird durchgeführt
    - WRITE_TS(X) := TS(T)
- T will READ_ITEM(X) durchführen:
  - Wenn WRITE_TS(X) > TS(T):
    - Jüngere Transaktionen haben X schon geschrieben
    - Operation wird abgewiesen, T wird abgebrochen („zu spät“)
  - Sonst:
    - READ_ITEM(X) wird durchgeführt
    - READ_TS(X) := max (READ_TS(X) , TS(T))



- Wenn T abgebrochen wird: neu starten mit neuem Zeitstempel

Beispiel

- Schedule: $r_8(X)$, $r_6(X)$, $r_9(X)$, $w_8(X)$, $w_{11}(X)$, $r_{10}(X)$
- Beachtung der Basis-Zeitstempelordnung
- READ_TS(X) = 0, WRITE_TS(X) = 0

Anfrage	**Antwort**	**Neue Zeitstempel**
$read\_item_8(X)$	OK	READ_TS(X) := 8
$read\_item_6(X)$	OK	READ_TS(X) := 8
$read\_item_9(X)$	OK	READ_TS(X) := 9
$write\_item_8(X)$	abgelehnt	T_8 abgebrochen
$write\_item_{11}(X)$	OK	WRITE_TS(X) := 11
$read\_item_{10}(X)$	abgelehnt	T_{10} abgebrochen

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
  - mit Zertifizierungssperren

Multiversionsprotokoll mit Zeitstempelordnung

- Mehrere Versionen X_1, … X_k von einem Datenobjekt X
- Für jedes X_i werden Zeitstempel gespeichert:
  - READ_TS(X_i) : der größte (aktuellste) aller Zeitstempel von Transaktionen, die diese Version gelesen haben
  - WRITE_TS(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 WRITE_ITEM(X)-Operation ausführen will und
  - Version i von X das größte WRITE_TS(X_i) aller Versionen von X hat und dabei WRITE_TS(X_i) < TS(T) gilt und
  - READ_TS(X_i) > TS(T),

  dann wird Transaktion T abgebrochen/zurückgesetzt
  - Stellt sicher, dass eine Transaktion T abgebrochen wird, wenn eine ältere Transaktion X_i geschrieben und eine jüngere als T X_i gelesen hat.
- Sonst
  - Erstelle eine neue Version X_{i+1} von X
  - Setze WRITE_TS(X_{k+1}) und READ_TS(X_{k+1}) auf TS(T)

Multiversionsprotokoll mit Zeitstempelordnung

Zwei Regeln für die Serialisierung:

2. Lesen:

- Wenn Transaktion T eine READ_ITEM(X)-Operation ausführen will und
  - Version i von X das größte WRITE_TS(X_i) aller Versionen von X hat und dabei WRITE_TS(X_i) < TS(T) gilt,

  dann wird T der Wert von X_i geliefert und READ_TS(X_i) wird auf den größeren Wert von TS(T) und READ_TS(X_i) gesetzt.
  - i.e., setze READ_TS(X_k) auf max(TS(T), READ_TS(X_k))

  → READ_ITEM(X)-Operationen können immer ausgeführt werden

Multiversionsprotokoll mit Zertifizierungssperren

- Erweitertes Sperrkonzept mit Mehrfachmodus-Sperren
  - Statt Zeitstempelordnung
- Einführung eines neuen Zustands, damit vier:

  - Lesegesperrt (S)
  - Schreibgesperrt (X)
  - Zertifizierungsgesperrt (C)
  - Entsperrt
- Idee: Auch bei Lesesperre noch Schreiben und bei Schreibsperre noch Lesen zu erlauben, wenn dies „geordnet“ (d.h. zertifiziert) geschieht.
  - Kompatibilitätsmatrix

Anforderung \ Zustand	*Keine Sperre*	S	X	C
S	✓	✓	✓, aber…	–
X	✓	✓, aber…	–	–
C	✓	–	–	–

Multiversionsprotokoll mit Zertifizierungssperren

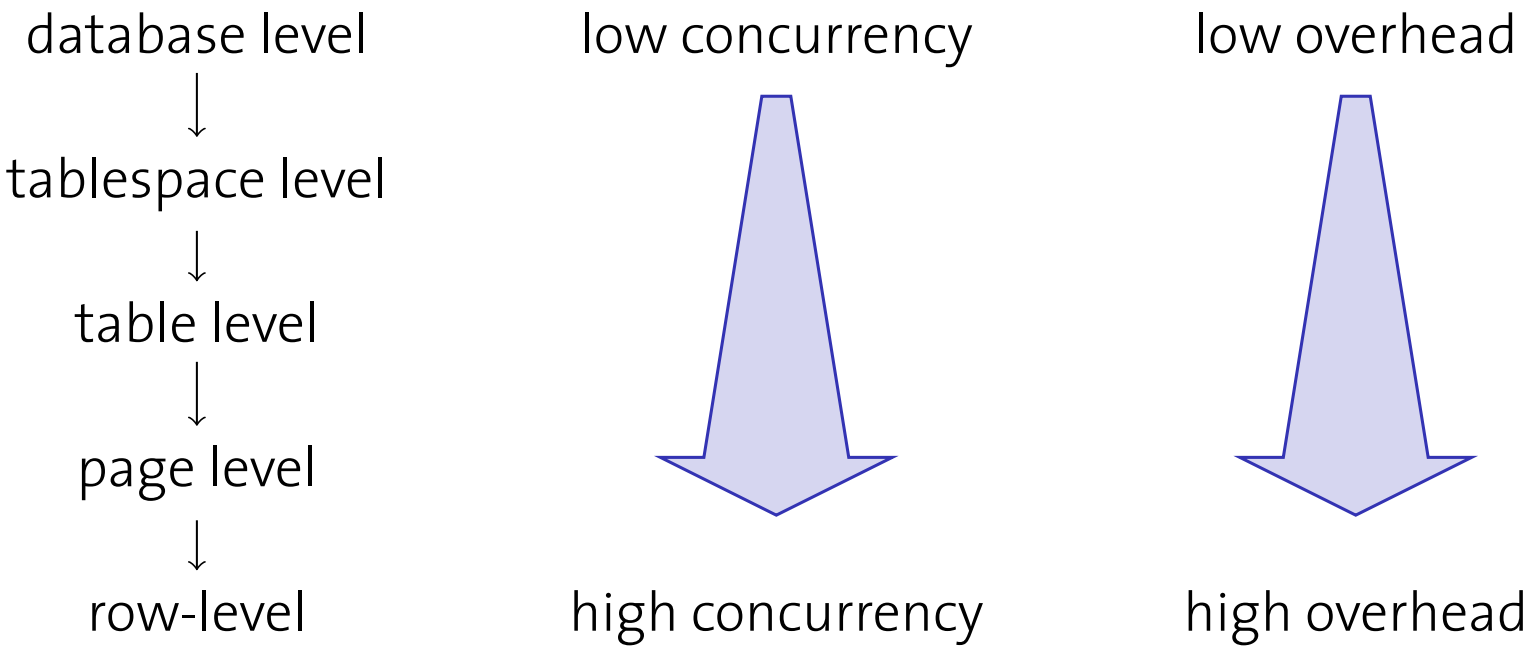
- Standardfall: wenn T eine Schreibsperre hat, können keine anderen Transaktionen auf das Objekt zugreifen
- Beim Multiversionsprotokoll mit Zertifizierungssperren ist es einer Transaktion T gestattet, ein Objekt X zu lesen, während eine Transaktion T' eine Schreibsperre auf X hält
  - Für jedes Objekt X sind zwei Versionen zugelassen:
    - Die Version X muss dabei immer von einer bestätigten Transaktion geschrieben worden sein
    - Die andere Version X' wird erzeugt, wenn eine Transaktion T' eine Schreibsperre auf X anfordert
- Andere Transaktionen können die bestätigte Version von 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

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
  - Große DB, verteiltes Arbeiten

Granularität des Sperrens

- Die Granularität des Sperrens unterliegt Abwägung



- Sperren mit multipler Granularität

Sperren mit multipler Granularität

- Entscheide die Granularität von Sperren für jede Transaktion (abhängig von ihrer Charakteristik)
  - Tupel-Sperre z.B. für

```
SELECT *
FROM Kunden
WHERE Kunden_ID = 42
```
Q_1

  - und eine Tabellen-Sperre für

```
SELECT *
FROM Kunden
```
Q_2

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

Vorhabens-Sperren

- Datenbanken setzen Vorhabens-Sperren (intention locks) für verschiedene Sperrgranularitäten ein
  - Sperrmodus Intention Share (IS)
  - Sperrmodus Intention Exclusive (IX)
  - Kompatabilitätsmatrix

Anforderung \ Zustand	*Keine Sperre*	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

- Eine Sperre I□ auf einer gröberen Ebene bedeutet, dass es eine Sperre □ auf einer niederen Ebene gibt

Vorhabens-Sperren

- Protokoll für Sperren auf mehreren Ebenen:
  1. Eine Transaktion kann jede Ebene g in Modus ☐ ∈ {S, X} sperren
  2. Bevor Ebene g in Modus ☐ gesperrt werden kann, muss eine Sperre I☐ für alle gröberen Ebenen gewonnen werden
- Anfrage Q_1 (Finde Tupel in Tabelle Kunden mit Kunden_ID=42) würde
  - eine IS-Sperre für Tabelle Kunden anfordern (auch für Tablespace + DB), dann
  - eine S-Sperre auf dem Tupel mit Kunden_ID=42 akquirieren
- Anfrage Q_2 (Kopiere Tabelle Kunden) würde eine
  - S-Sperre für die Tabelle Kunden anfordern (und IS-Sperren auf dem Tablespace und der Datenbank)

Anforderung \ Zustand	*Keine Sperre*	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Entdeckung von Konflikten

- 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, folgende Anfrage ist auch noch zu bearbeiten

```
UPDATE Kunden                    Q3
SET NAME = 'John Doe'
WHERE Kunden_ID = 17
```

- Benötigte Sperren
  - IX-Sperre auf Tabelle Kunden (und ...)
  - X-Sperre auf dem Tupel mit Kunden_ID=17

→ Kompatibel mit Q_1 (kein Konflikt zw. IX und IS auf Tabellenebene)

→ Inkompatibel mit Q_2 (S-Sperre auf Tabellenebene von Q_2 steht in Konflikt mit der IX-Sperre bzgl. Q_3)

Anforderung \ Zustand	*Keine Sperre*	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Konsistenzgarantien

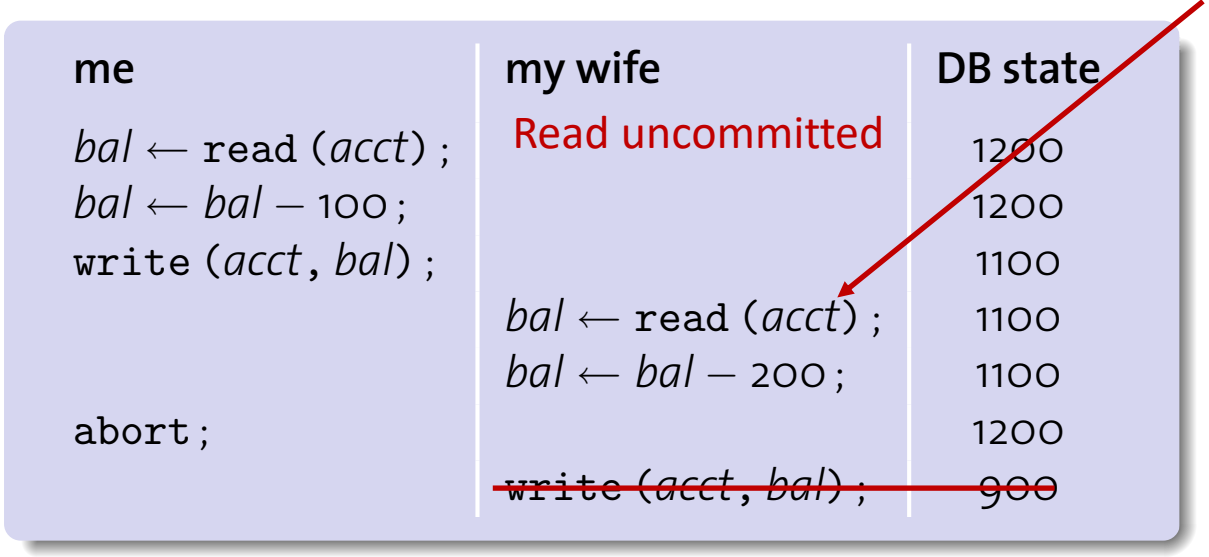
- In einigen Fällen kann man mit einigen kleinen Fehlern im Anfrageergebnis leben
  - „Fehler“ bezüglich einzelner Tupel machen sich in Aggregatfunktionen evtl. kaum bemerkbar
    - Lesen inkonsistenter Werte (inconsistent read anomaly)
- Ab SQL-92 kann man Isolations-Modi spezifizieren:
  SET ISOLATION <MODE>

```
SET ISOLATION SERIALIZABLE;
```

- Es gibt weniger strikte Modi, unter denen die Performanz höher ist (weniger Verwaltungsaufwand z.B. für Sperren)

Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach Zwei-Phasen-Sperrprotokoll)



Read uncommited nur für lesende Transaktion

Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach Zwei-Phasen-Sperrprotokoll)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
  - Schreibsperren nach Zwei-Phasen-Sperrprotokoll

T1(read committed)	T2
Read(A)	
	Write(A)
	Write(B)
	commit
Read(B)	
Read(A)	

Nur commited gelesen

Aber:
Unrepeatable Read
nicht ausgeschlossen

Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach Zwei-Phasen-Sperrprotokoll)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
  - Schreibsperren nach Zwei-Phasen-Sperrprotokoll
- Repeatable read (auch ‘read stability‘)
  - Lese- und Schreibsperren nach Zwei-Phasen-Sperrprotokoll
  - Problem: keine sog. range locks
    - Anfrage Q_1

      ```
      SELECT SUM(Gehalt)
      FROM Mitarbeiter
      WHERE AbtNr=5
      ```

    - Anfrage Q_2

      ```
      INSERT INTO Mitarbeiter
                  (SVN, NName, VName, Adresse, Gehalt, GebDatum, AbtNr, VorgesSVN)
      VALUES ('90123456G789 ', 'Marini', 'Richard',
                  '98 Oak Forest, Katy, TX', 37000, '30.12.1962‘, 5, '67890123D456‘)
      ```

  → Phantom Read

Isolations-Modi

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

Resultierende Konsistenzgarantien

Isolationsmodus	dirty read	non-repeatable read	phantom read
read uncommitted	möglich	möglich	möglich
read committed	–	möglich	möglich
repeatable read	–	–	möglich
serializable	–	–	–

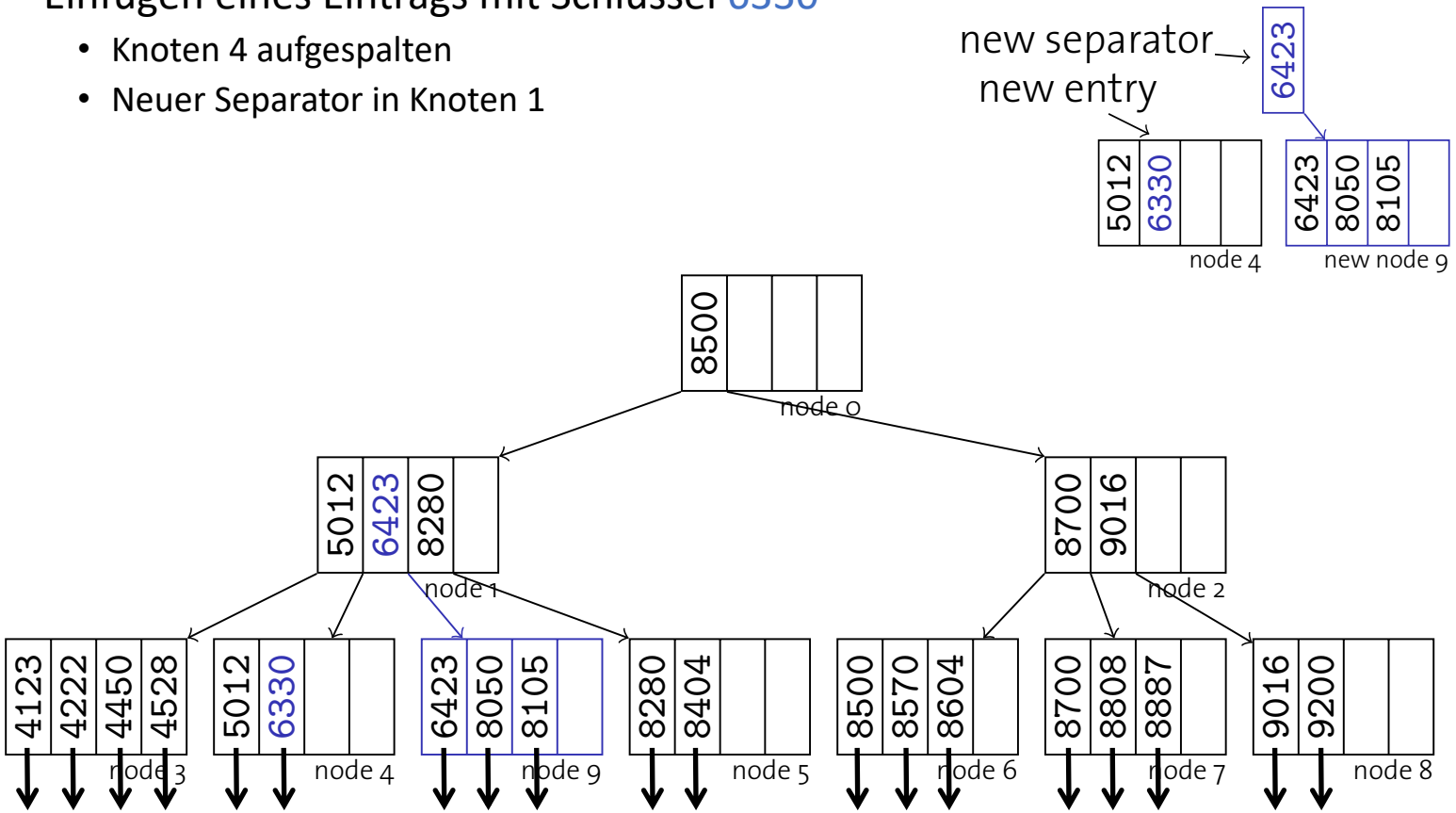
- Einige Implementierungen unterstützen mehr, weniger oder andere Isolationsmodi
- Nur wenige Anwendungen benötigen (volle) Serialisierbarkeit

Sperren in Index-Strukturen

Transaktionsverarbeitung

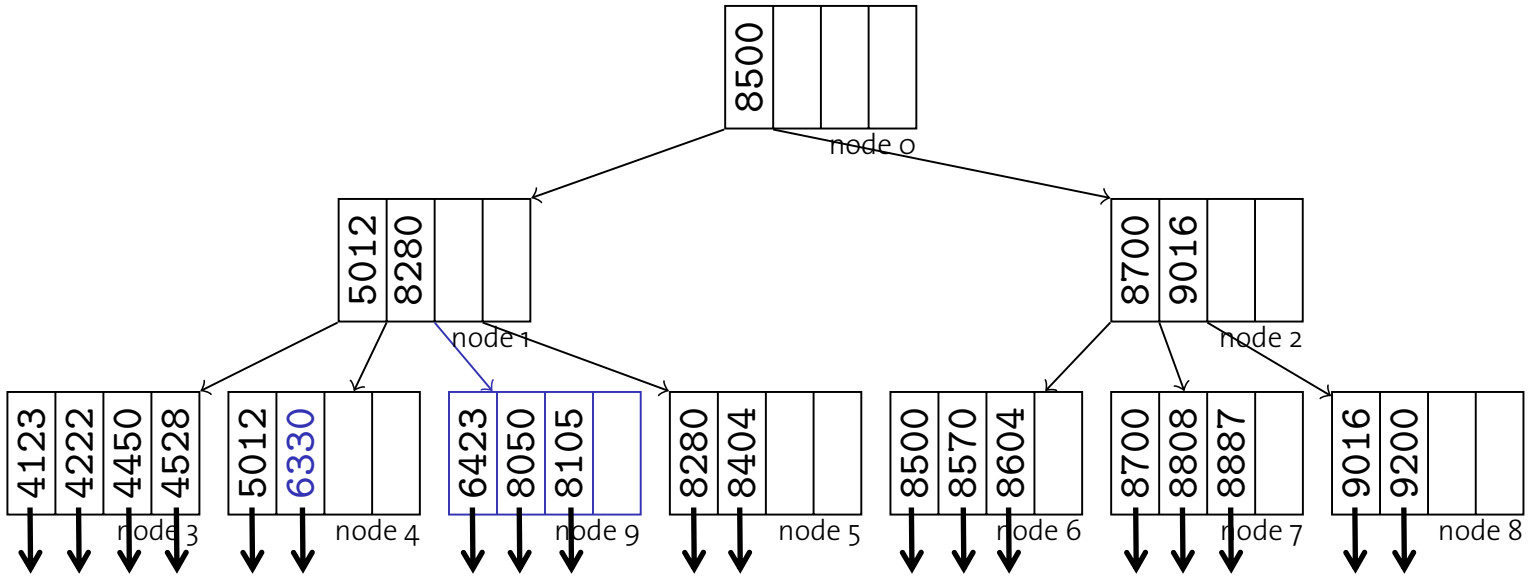
Nebenläufigkeit beim Indexzugriff

- Betrachten wir eine Transaktion T_w, die etwas in einen B⁺-Baum einführt, was zu einer Splitoperation führt
  - Einfügen eines Eintrags mit Schlüssel 6330
    - Knoten 4 aufgespalten
    - Neuer Separator in Knoten 1



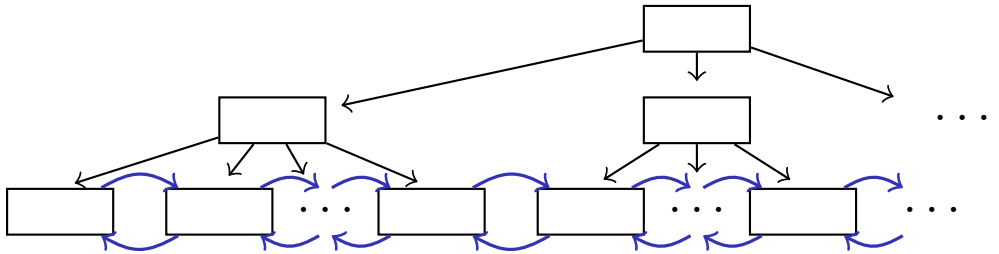
Nebenläufigkeit beim Indexzugriff

- Angenommen, die Aufspaltung ist gerade erfolgt, aber der neue Separator 6423 ist noch nicht etabliert
- Weiterhin: nebenläufiges Lesen in Transaktion T_r sucht nach 8050
  - Verzeigerung weist auf Knoten $node_4$
  - Knoten $node_4$ enthält 8050 nicht mehr, entsprechender Datensatz wird nicht gefunden
- Auch in B⁺-Bäumen muss beim Umbau mit Sperren gearbeitet werden!



Sperren und B⁺-Baum-Indexe

- B⁺-Baum-Operationen
  - Für die Suche erfolgt ein Top-Down-Zugriff
  - Für Aktualisierungen ...
    - erfolgt erst eine Suche,
    - dann werden Daten ggf. in ein Blatt eingetragen und
    - ggf. werden Aufspaltungen von Knoten nach oben propagiert.



- Nach dem Zwei-Phasen-Sperrprotokoll ...
  - müssen Lese/Schreib-Sperren auf dem Weg nach unten akquiriert werden (Konversion provoziert ggf. Verklemmungen)
  - 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

- Zwei-Phasen-Sperrprotokoll nicht angemessen für B⁺-Bäume

Sperrprotokoll für B^+-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 (Zwei-Phasen-Sperrprotok.)

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

Aufspaltungssicherheit

- Wir müssen auf dem Weg nach unten in den B-Baum Schreibsperren 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)

readers

```
1 place S lock on root ;
2 current ← root ;
3 while current is not a leaf node do
4     place S lock on appropriate son of current ;
5     release S lock on current ;
6     current ← son of current ;
```

writers

```
1 place X lock on root ;
2 current ← root ;
3 while current is not a leaf node do
4     place X lock on appropriate son of current ;
5     current ← son of current ;
6     if current is safe then
7         release all locks held on ancestors of current ;
```

Erhöhung der Nebenläufigkeit

- Auch mit Sperrkopplung werden eine beträchtliche Anzahl von Sperren für innere Knoten benötigt (wodurch die Nebenläufigkeit gemindert wird)
- 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 (inkl. einer Schreibsperre für das betreffende Blatt)
  - Wenn die Annahme falsch ist, traversiere Indexbaum erneut unter Verwendung korrekter Schreibsperren

Sperrkopplungsprotokoll (Variante 2)

Modifikationen nur für Schreibvorgänge

```
1  place S lock on root ;
2  current ← root ;
3  while current is not a leaf node do
4      son ← appropriate son of current ;
5      if son is a leaf then
6          place X lock on son ;
7      else
8          place S lock on son ;
9      release lock on current ;
10     current ← son ;
11 if current is unsafe then
12     release all locks and repeat with protocol Variant 1 ;
```

Zusammenfassung

- 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

Zwischen-Rückblick

- Protokolle zur Sicherung der ACID-Eigenschaften im Mehrbenutzerbetrieb
  - Binäre Sperren
    - Nur zwei Zustände: Datenobjekt ist gesperrt oder nicht
    - Zu 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

Übersicht

- Transaktionen
- Schedules
- Sperren
- Wiederherstellungsverwaltung
  - Fehlerszenarien und die Aufgabe des Transaktionsverwalters
  - Logging
    - Log-Information
    - Log-Files: Inhalt und Aufbau
    - Nutzung bei Transaktionsverarbeitung und in Fehlersituationen
  - Recovery
    - Restart eines DBMS bzw. einer DB

Fehlerszenarien

Wiederherstellungsverwaltung

TRANSAKTIONSFEHLER (1/3)

- Verschiedene Gründe, weshalb eine DB-Transaktion fehlschlagen kann
- Sieben typische Fehlerfälle
  - Mit in der Literatur uneinheitliche 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

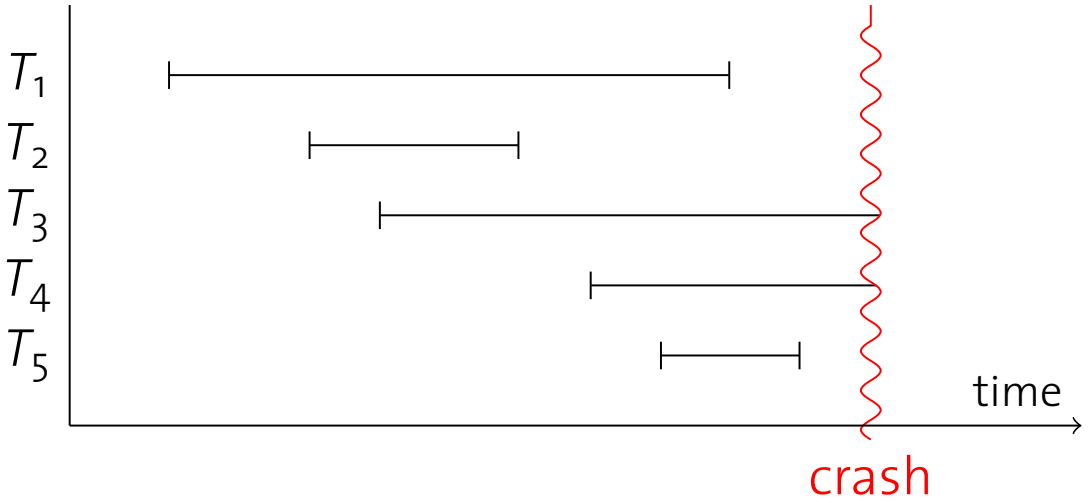
   → Lese-/Schreibfehler bei der Transaktionsausführung
5. „Katastrophen“: [Medienfehler]
   - Subsumierung verschiedener Ausnahmesituationen, z.B. physischer Verlust eines Datenträgers durch Brand oder Diebstahl
   - Ausnahmesituationen, die sich aus einer fehlerhaften Bedienung des Systems ergeben, wobei 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]
   - Die Komponente zur Überwachung der nebenläufigen Transaktionsausführung veranlasst die Unterbrechung, nicht den Abbruch einer Transaktion, um Interferenzen mit anderen Transaktionen zu vermeiden.
   - Die in diesem Zusammenhang gestoppte Transaktion wird später wieder gestartet.

- Wichtig: 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 und 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 und T_4 wurden noch nicht beendet
  → Atomarität: Alle Effekte müssen rückgängig gemacht werden (undo)

Wiederherstellungsverwalter – 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

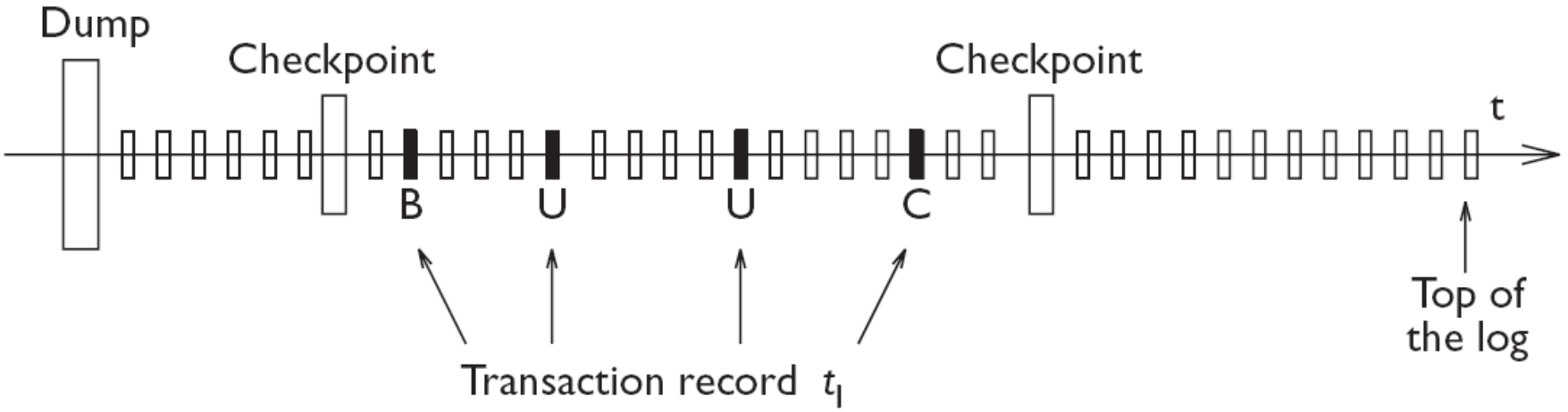
Organisation des Log-File

- Sequentielle Struktur
- Speichert DB-Änderungen chronologisch
- Verwaltet durch den Wiederherstellungsverwalter
- Liegt auf persistentem Speicher
- Zwei Typen von Log-Einträgen:
  1. Transaktionsbezogene Einträge:
     - 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 (seltener Eintrag, da vollständige DB-Kopie angelegt wird)
     - CHECKPOINT (häufigerer Eintrag, da nur partielle Aktualisierung erfolgt)

T = ID der Transaktion
X = ID des Datenobjekts
AS = Wert des Datenobjekts nach der Operation (after state)
BS = Wert des Datenobjekts vor der Operation (before state)

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

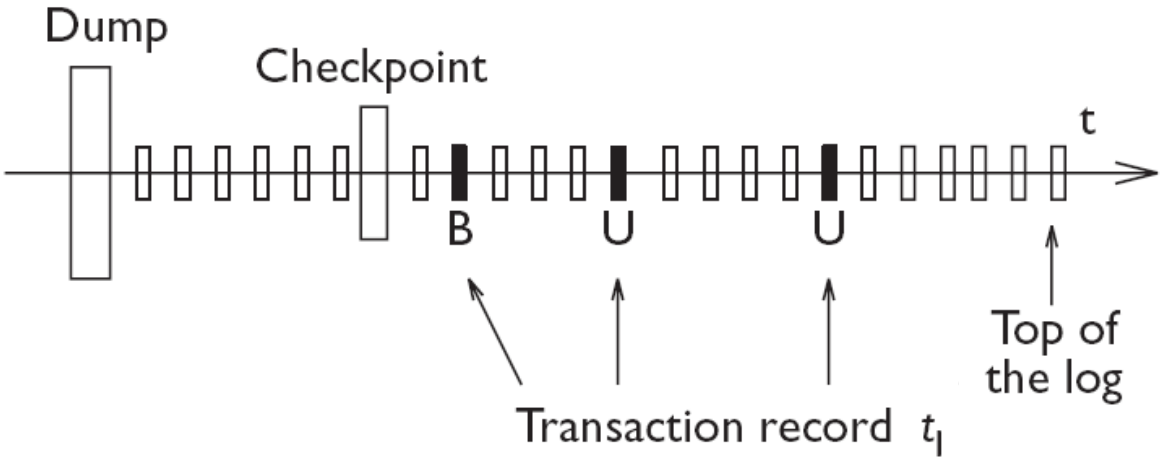
- Im Fehlerfall
  - Transaktionen T mit BEGIN_TRANSACTION und ohne COMMIT werden – soweit nötig – rückgängig gemacht

    → 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

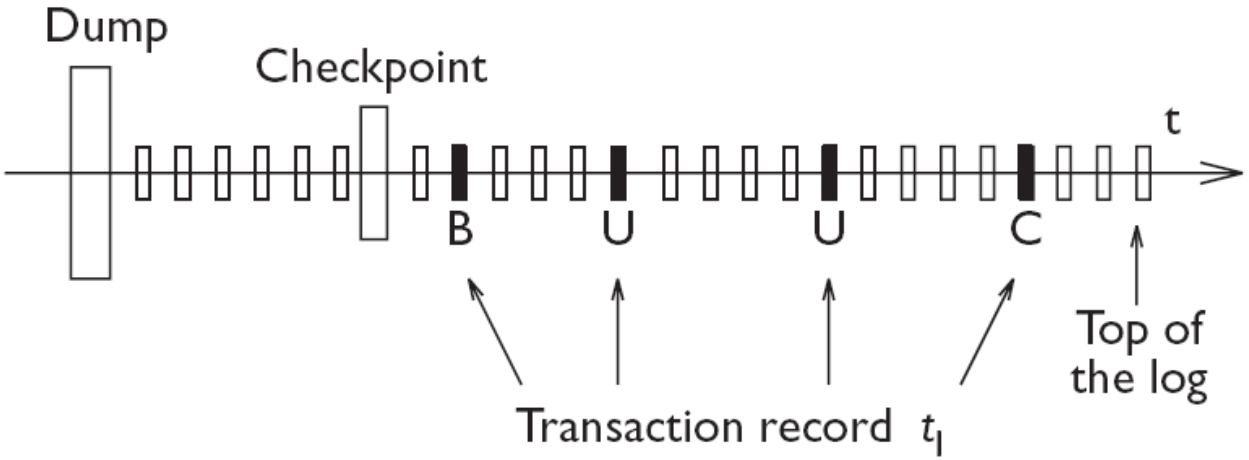
UNDO

- 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

- 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 bezeichnet in diesem Zusammenhang die Idee, …
  - … dass eine Transaktion T erst dann ihren COMMIT-Point erreichen kann, 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 wird.
  - Damit werden die DB-Veränderungen abgeschlossener Transaktionen dauerhaft in die DB eingefügt
- Checkpoint-Eintrag:
  - $CK(T_1, T_2, T_3, \ldots, T_n)$, enthält die Identifier T_i der aktiven Transaktionen
- Bei einem Recovery-Vorgang muss dann nur bis zum letzten Checkpoint-Eintrag zurückgegangen werden und von dort an die aktiven Transaktionen abhandeln (UNDO/REDO)

DUMP

- Vollständige Kopie der DB
  - „Backup“
- Wird i.d.R. erzeugt, wenn DB nicht operativ ist
  - (z.B. einmal pro Nacht)
- Wird im Allg. im persistenten Speicher abgelegt
- Nach Erzeugung des DUMP entsprechender Eintrag im Log-File

Grundregeln für Log-Einträge

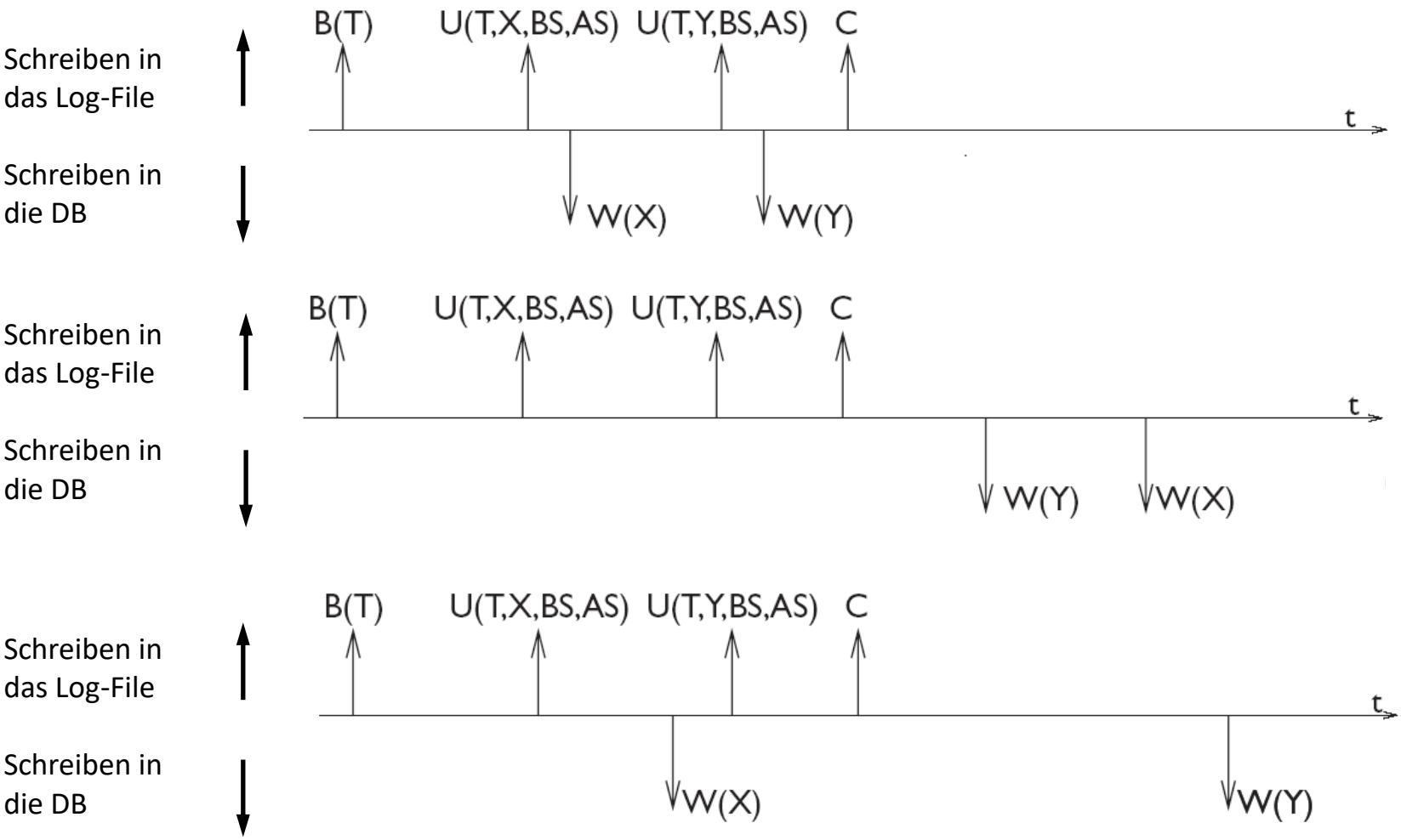
- 2 Grundregeln:
  - Write-Ahead Log (WAL)
    BS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die entsprechende Operation auf der DB ausgeführt wird
  - Commit-Precedence (CP)
    AS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die Transaktion abgeschlossen 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

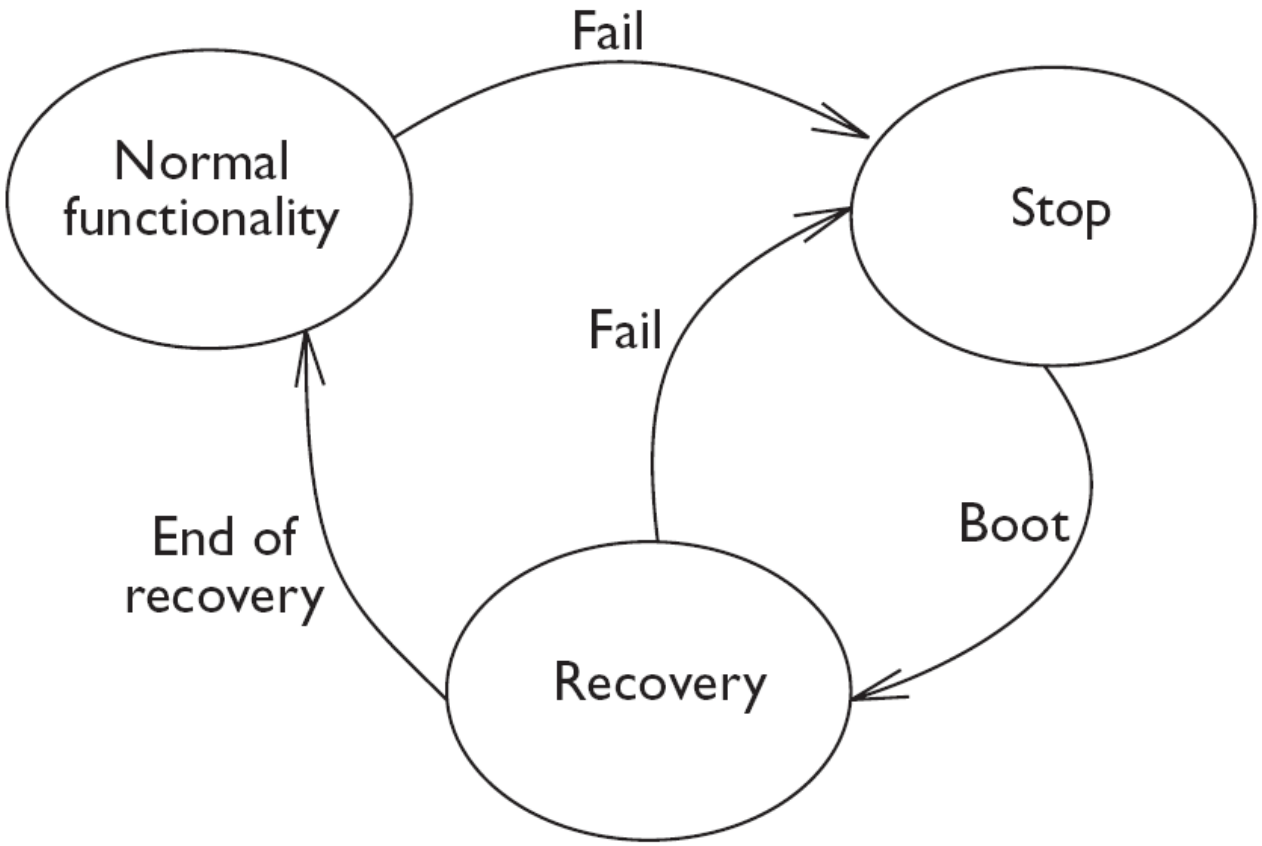
Beispiele


Schreiben in das Log-File
Schreiben in die DB
B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C
t
W(X)
W(Y)
Schreiben in das Log-File
Schreiben in die DB
B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C
t
W(Y)
W(X)
Schreiben in das Log-File
Schreiben in die DB
B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C
t
W(X)
W(Y)

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-Menge (Transaktionen, die verworfen werden müssen)
  - Bilde REDO-Menge (Transaktionen, die reproduziert werden können)
- Phase 3
  - 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
  - Durchlaufe das Log-File vorwärts und führe REDO für alle Operationen der Transaktionen in der REDO-Menge aus

- Dieses Vorgehen sichert Atomarität und Dauerhaftigkeit

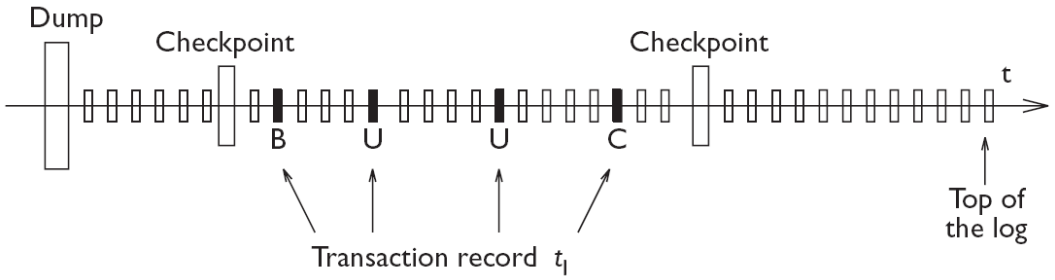
Cold Restart

- 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

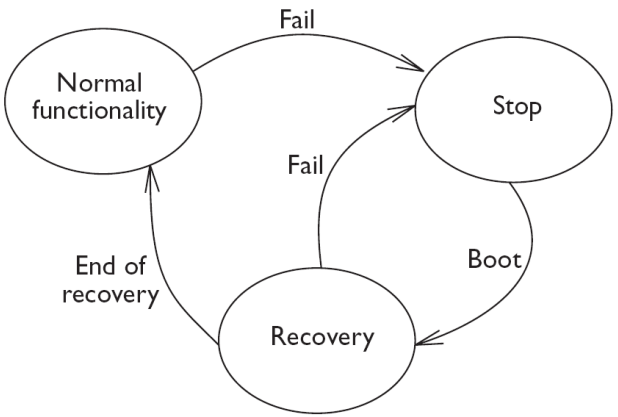
- Auch dieses Vorgehen sichert Atomarität und Dauerhaftigkeit

Zusammenfassung

- 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



Das Gesamtbild der Architektur