Transaktionen
und
Mehrbenutzersynchronisation

Foliensatz

DI(FH) Gerald Aistleitner, 2016/17

Transaktion - Begriffsbildung

Unter einer Transaktion versteht man die „Bündelung“ mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen als *Einheit* fehlerfrei ausgeführt werden soll. (Kemper, Eickler)

Eine Transaktion stellt also eine Folge von Datenverarbeitungsbefehlen (DML – read, update, insert, delete) dar, die die Datenbasis von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt.

Klassisches Transaktionsbeispiel - Überweisung

Folge von Befehlen, die bei einer Überweisung in einer Transaktion zusammengefasst werden:

- Lese Saldo von Konto A in Variable a: read(A,a)
- Reduziere Saldo von a um 50 Euro: a=a-50
- Schreibe den neuen Saldo von A: write(A, a)
- Lese Saldo von Konto B in Variable b: read(B,b)
- Erhöhe den Saldo um 50 Euro: b=b+50
- Schreibe den neuen Saldo von B: write(B,b)

Operationen auf Transaktions-Ebene

- ***read***
  nur lesender Zugriff, keine Veränderung von DAten
- ***write***
  Verändernde DB-Zugriffe (Insert, Update, Delete)
- ***begin of transaction (BOT)***
  Kennzeichnet den Beginn der in einer Transaktion abgebildeten Befehlsfolge
- ***commit***
  Beenden einer Transaktion. Änderungen der Datenbasis wird festgeschrieben.
- ***abort***
  Abbruch einer Transaktion. Datenbasis muss auf den Zustand vor BOT zurückgesetzt werden.
- **define savepoint (Optional)**
  Sicherungspunkt, an den sich die Transaktion zurücksetzen lässt
- **backup transaction (Optional)**
  Zurücksetzen auf (meist letzten) Sicherungspunkt

*Eigenschaften einer Transaktion - **A**CID*

A – Atomicity (Atomarität)

Transaktionen sind untrennbar.
Es werden entweder alle Änderungen einer Transaktion vollständig in der Datenbasis festgeschrieben oder gar keine.
→ **„Alles oder Nichts“** - Prinzip

Anhand des Überweisungs-Beispiels sollte schnell klar werden, warum diese Eigenschaft notwendig ist!

*Eigenschaften einer Transaktion - A**C**ID*

C – Consistency (Konsistenz)

Eine Transaktion muss **nach Beendigung** einen **konsistenten Zustand** der Datenbasis hinterlassen. Ansonsten muss sie komplett zurückgesetzt werden (siehe Atomarität).

Konsistent bedeutet, dass es keinerlei Widersprüche in der Datenbasis gibt (zB mehrfach vorhandene Datensätze mit widersprüchlichem Inhalt, Verletzung von referentieller Integrität, etc.).

Während der Ausführung einer Transaktion kann die Datenbank jedoch temporär inkonsistent sein!

Eigenschaften einer Transaktion - ACID

I – Isolation

Transaktionen dürfen sich nicht gegenseitig beeinflussen. Jede Transaktion muss (logisch gesehen) so ausgeführt werden, als wäre sie die einzige aktuell laufende Transaktion und hätte somit das ganze System für sich alleine.

Es darf also nicht passieren, dass zwei Transaktionen gleichzeitig die selben Daten bearbeiten und sich gegenseitig behindern bzw. deswegen falsche Ergebnisse liefern.

*Eigenschaften einer Transaktion - ACI**D***

D – Durability (Dauerhaftigkeit)

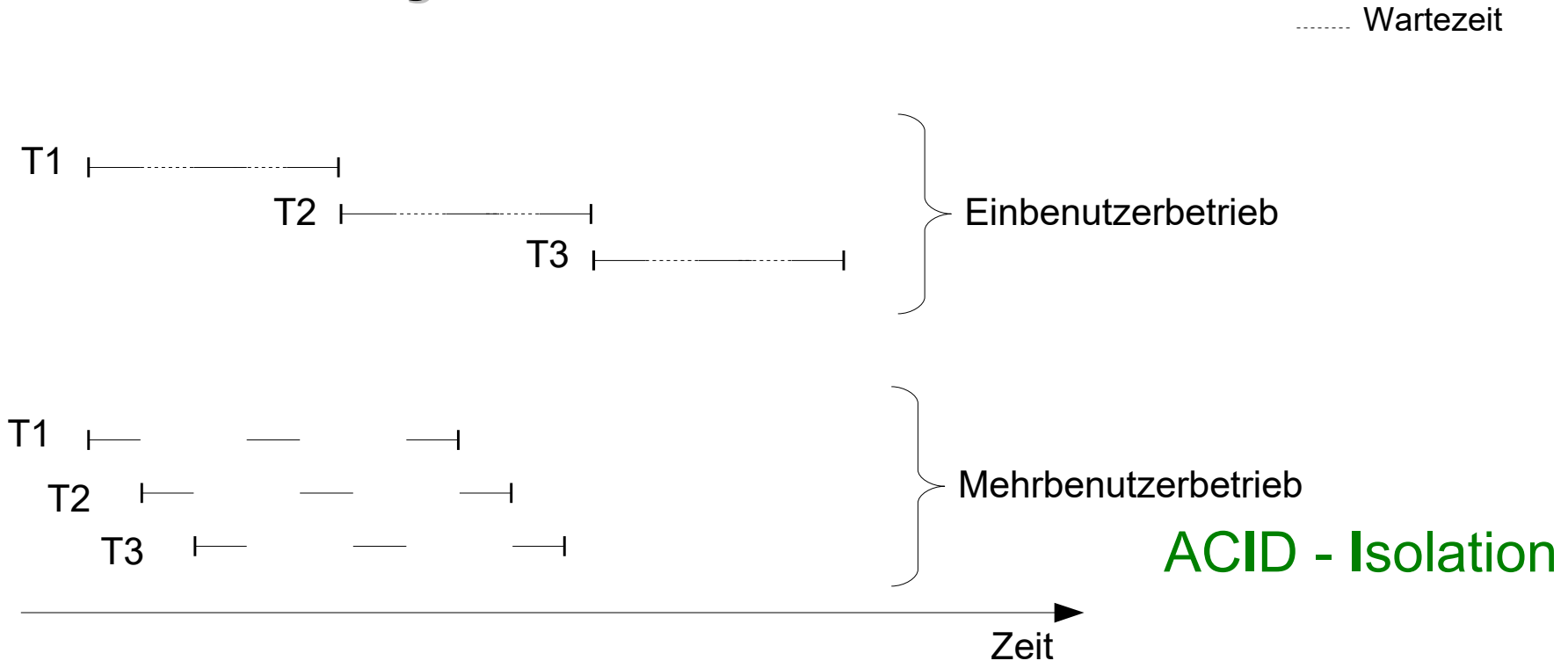
Alle Auswirkungen einer erfolgreich abgeschlossenen Transaktion müssen dauerhaft gespeichert werden.

Die Transaktionsverwaltung muss sicherstellen, dass dies auch nach einem Systemfehler gewährleistet ist. (siehe auch Recovery)

Mehrbenutzersynchronisation

Warum Mehrbenutzerbetrieb?

Häufig Wartezeiten in Transaktionen (Zugriff auf langsamere Betriebsmittel, interaktive Benutzereingaben, ...), welche von anderen Transaktionen genutzt werden können
→ Performancegewinn



ACID - **I**solation

Fehlerklassen im Mehrbenutzerbetrieb

Im unkontrollierten (und nicht synchronisierten) Mehrbenutzerbetrieb können verschiedene Arten von Fehlern auftreten, die in folgende Kategorien fallen:

- Verlorengegangene Änderungen
- Lesen nicht freigegebener Änderungen
- Phantomproblem

Verlorengegangene Änderungen

Auch als **„Lost Update“**-Problem bekannt.

T1: Transferiert 300 Euro von Konto A auf Konto B
T2: schreibt Konto A 3 % Zinsen gut

Schritt	T1	T2
1.	read(A,a_1)	
2.	a_1 -= 300	
3.		read(A,a_2)
4.		a_2 *= 1.03
5.		write(A,a_2)
6.	write(A,a_1)	
7.	read(B,b_1)	
8.	b_1 += 300	
9.	write(B,b_1)	

Verlorengegangene Änderungen

Auch als **„Lost Update“**-Problem bekannt.

T1: Transferiert 300 Euro von Konto A auf Konto B
T2: schreibt Konto A 3 % Zinsen gut

Schritt	T1	T2
1.	read(A,a_1)	
2.	a_1 -= 300	
3.		read(A,a_2)
4.		a_2 *= 1.03
5.		write(A,a_2)
6.	write(A,a_1)	
7.	read(B,b_1)	
8.	b_1 += 300	
9.	write(B,b_1)	

Zinsgutschrift geht verloren!

Lesen nicht freigegebener Änderungen

Auch als „dirty read“ bezeichnet, da ein Wert gelesen wird, der so niemals in einem gültigen Zustand der Datenbasis vorkommt.

Schritt	T1	T2
1.	read(A,a_1)	
2.	a_1 -= 300	
3.	write(A,a_1)	
4.		read(A,a_2)
5.		a_2 *= 1.03
6.		write(A,a_2)
7.	read(B,b_1)	
8.	…	
9.	**abort**	

Lesen nicht freigegebener Änderungen

Auch als „dirty read“ bezeichnet, da ein Wert gelesen wird, der so niemals in einem gültigen Zustand der Datenbasis vorkommt.

Schritt	T1	T2
1.	read(A,a_1)	
2.	a_1 -= 300	
3.	write(A,a_1)	
4.		read(A,a_2)
5.		a_2 *= 1.03
6.		write(A,a_2)
7.	read(B,b_1)	
8.	…	
9.	**abort**	

T1 rollback

300 Euro wurden nicht verzinst!

Phantomproblem / Unrepeatable Read

Während der Abarbeitung einer Transaktion generiert eine weitere Transaktion ein Datum, das mitberück-sichtigt werden sollte.

T1	T2
	select sum(Kontostand)
	from Konten
insert into Konten	
values (C, 1000, …)	
	select sum(Kontostand)
	from Konten

Serialisierbarkeit

Serielle Ausführung bedeutet meist schlechte Performance (verglichen mit Nebenläufigkeit, siehe Folie 10), vermeidet dafür die gerade aufgezählten Fehler.

Serialisierbare Ausführung einer Menge von Transaktionen **entspricht einer kontrollierten, nebenläufigen, verzahnten Ausführung**, wobei die Wirkung der nebenläufigen Ausführung einer möglichen **seriellen Ausführung** der Transaktion entspricht. (gesteuert über Kontrollkomponente)

Unter einer **Historie** versteht man die zeitliche Anordnung der elementaren Operationen von einer Menge von Transaktionen.

Beispiel

- T1: Überweisung von A nach B
- T2: Überweisung von C nach A

Schritt	T1	T2
1.	BOT	
2.	read(A)	
3.		BOT
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	commit	
10.		read(A)
11.		write(A)
12.		commit

Serialisierbare Historie

Schritt	T1	T2
1.	BOT	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	commit	
7.		BOT
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		commit

Äquivalente serielle Ausführung

Beispiel

Schritt	T1	T3
1.	BOT	
2.	read(A)	
3.	write(A)	
4.		BOT
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		commit
10.	read(B)	
11.	write(B)	
12.	commit	

Nicht serialisierbare Historie

Datenobjekt A: T1 kommt vor T3
Datenobjekt B: T3 kommt vor T1

→ Historie **nicht äquivalent** zu einer der beiden möglichen seriellen Ausführungen von T1 und T3 (Lost Update)!

Rücksetzbare Historien

- Minimalanforderung, dass noch aktive Transaktionen abgebrochen werden können, ohne dass andere bereits mit commit abgeschlossene Transaktionen in Mitleidenschaft gezogen werden.

- Eine Transaktion heißt rücksetzbar, falls immer die schreibende Transaktion vor der lesenden Transaktion ihr commit durchführt.

Ansonsten könnte man die schreibende TA evtl. nicht zurücksetzen, da die lesende Transaktion dann mit einem offiziell nie existenten Wert ihre Berechnung committed hätte.

Kaskadierendes Rücksetzen

Schritt	T1	T2	T3	T4	T5
0.	…				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1 (**abort**)				

- Das Rücksetzen einer Transaktion kann eine Menge weiterer Rollbacks in Gang setzen.
  → Beeinträchtigung der Performance!

- Ziel ist Vermeidung von kaskad. Rücksetzen:
  Änderungen werden erst nach dem commit freigegeben

Strikte Historien

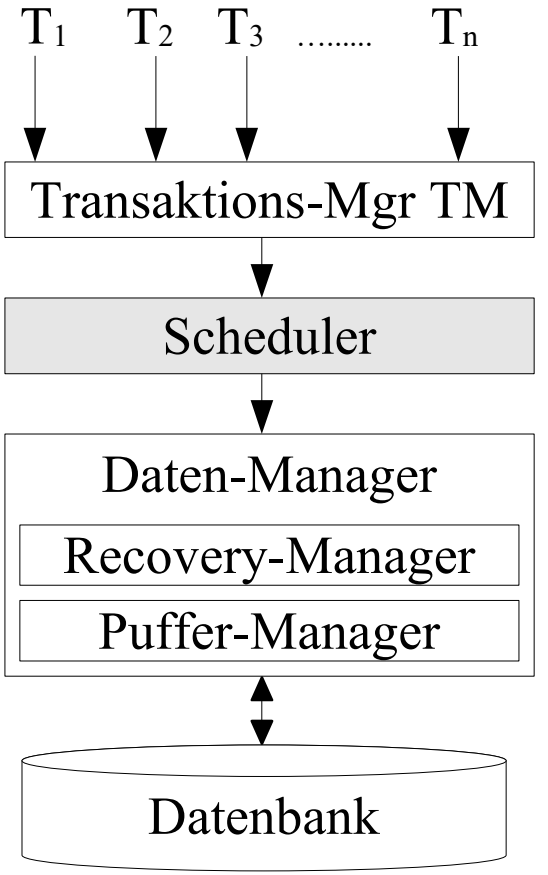
- Bei strikten Historien dürfen auch veränderte Daten einer noch laufenden Transaktion nicht überschrieben werden.

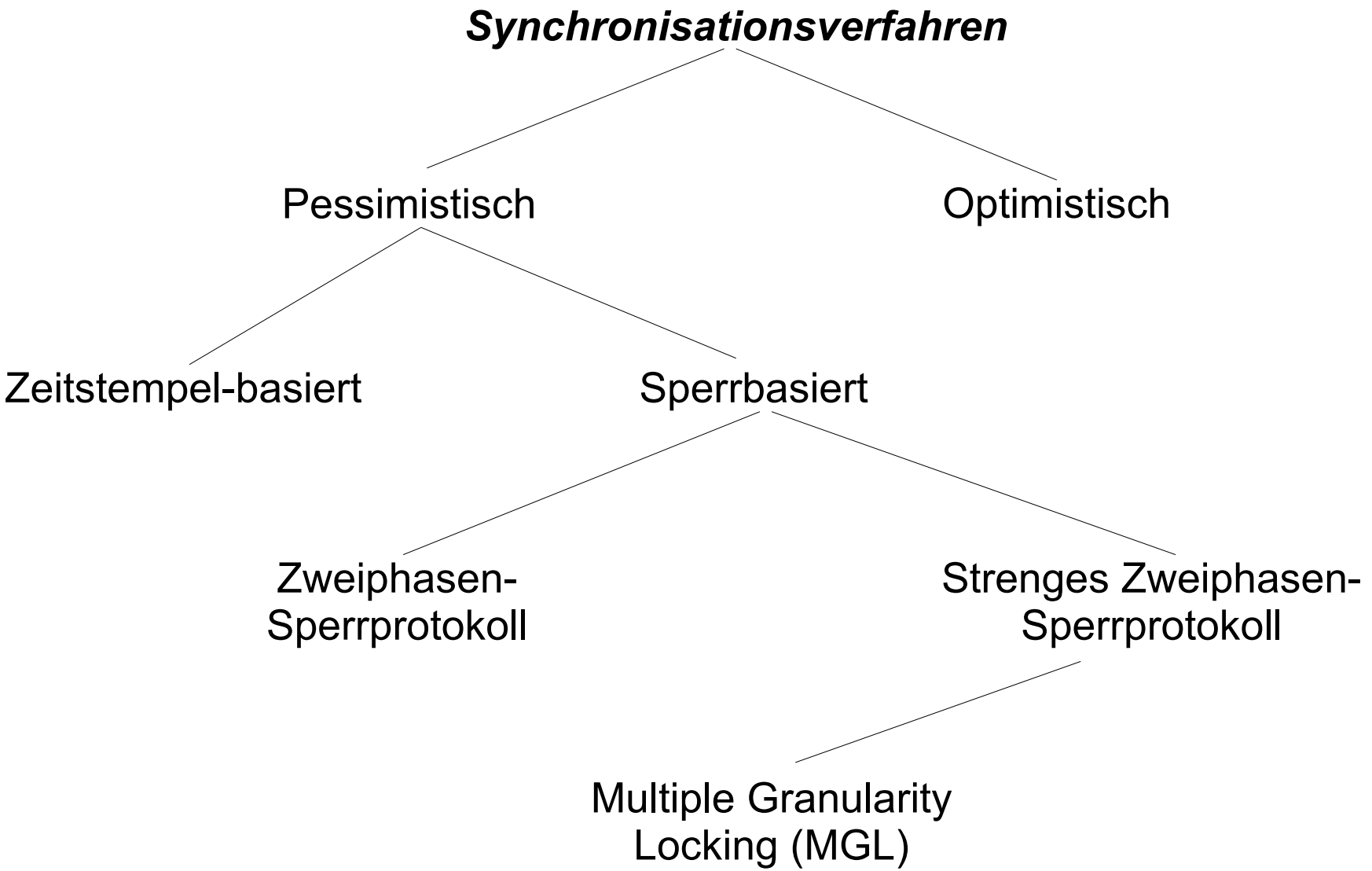
  Falls ein Datum A von einer Transaktion geschrieben wurde, dürfen andere Transaktionen erst nach der Beendigung dieser Transaktion (durch **commit oder abort)** auf A lesend oder schreibend zugreifen.

Der Datenbank-Scheduler

Der Scheduler hat die Aufgabe, die Einzeloperationen verschiedener Transaktionen so auszuführen, dass die Historie definierte Anforderungen erfüllt:

- Serialisierbarkeit (Mindestanforderung!)
- Ohne kaskadierendes Rollback rücksetzbar (i.A. ebenfalls gefordert)




Synchronisationsverfahren
Pessimistisch
Optimistisch
Zeitstempel-basiert
Sperrbasiert
Zweiphasen-
Sperrprotokoll
Strenges Zweiphasen-
Sperrprotokoll
Multiple Granularity
Locking (MGL)

Sperrbasierte Synchronisation

- Während des laufenden Betriebs wird sichergestellt, dass die resultierende Historie serialisierbar bleibt.
- Wird erreicht, indem eine Transaktion erst nach Erhalt einer entsprechenden Sperre auf ein Datum zugreifen kann.

Sperrmodi

- **S-Lock** (shared, read lock, Lesesperre):
  Wenn eine Transaktion eine S-Sperre besitzt, kann diese **read**-Operationen ausführen. Mehrere Transaktionen können gleichzeit eine S-Sperre auf das gleiche Objekt besitzen

- **X-Lock** (exclusive, write lock, Schreibsperre):
  write darf nur ausgeführt werden, wenn die TA ein X-Lock auf das Objekt besitzt

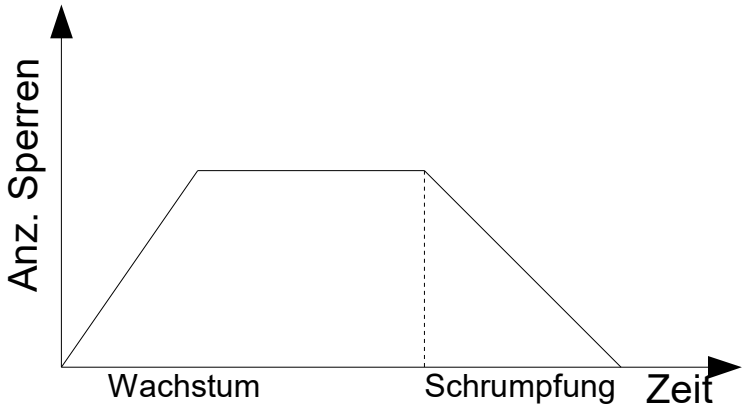
- Verträglichkeitsmatrix

	NL	S	X
S	✓	✓	-
X	✓	-	-

Zwei-Phasen-Sperrprotokoll

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. Eine TA muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Ist die Sperre nicht möglich, wird die Transaktion in eine Warteschlange eingereiht, bis die Sperre möglich ist.
4. Jede Transaktion durchläuft 2 Phasen:
    1. Eine **Wachstumsphase**:
       Sperren anfordern, aber nicht freigeben
    2. Eine **Schrumpfungsphase**
       Freigabe der Sperren, keine weiteren anfordern
5. Bei EOT muss eine Transaktion alle Sperren zurückgeben

Zwei-Phasen-Sperrprotokoll



2PL-Schedules sind immer serialisierbar!

Das Beispiel rechts entspricht der seriellen Ausführung von T_1 vor T_2.

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX(A)		
3.	read(A)		
4.	write(A)		
5.		BOT	
6.		lockS(A)	T_2 muß warten
7.	lockX(B)		
8.	read(B)		
9.	unlockX(A)		T_2 wecken
10.		read(A)	
11.		lockS(B)	T_2 muß warten
12.	write(B)		
13.	unlockX(B)		T_2 wecken
14.		read(B)	
15.	commit		
16.		unlockS(A)	
17.		unlockS(B)	
18.		commit	

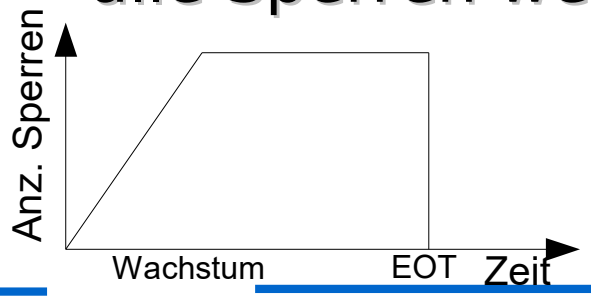
Zwei-Phasen-Sperrprotokoll / Kaskad. Rücksetzen

- 2-Phasen-Sperrprotokoll garantiert die Serialisierbarkeit

- Es vermeidet aber nicht das kaskadierende Rollback siehe vorheriges Beispiel: bei Abbruch vor Schritt 15 müsste auch die Transaktion T_2 zurückgesetzt werden.

Strenges 2PL-Protokoll

- Anforderung 1-5 wie bisher
- jedoch keine Schrumpfungsphase mehr, sondern alle Sperren werden erst zum EOT freigegeben.



Verklemmungen / Deadlocks

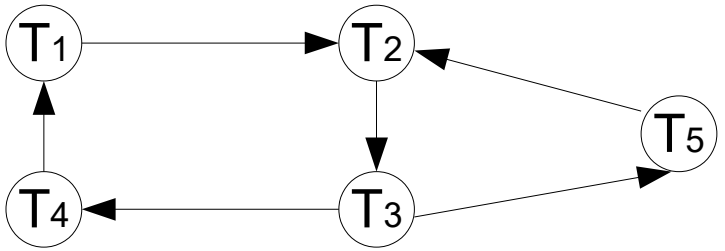
Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX(A)		
3.		BOT	
4.		lockS(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	lockX(B)		T_1 wartet auf T_2
9.		lockS(A)	T_2 wartet auf T_1
10.	….	….	->Deadlock

Preclaiming zur Vermeidung von Verklemmungen (Konservatives 2-Phasen-Sperrprotokoll):
Transaktion wird erst begonnen, wenn alle ihre Sperranforderungen schon bei Beginn erfüllt werden können.
→ Einschränkung der Parallelität, bedingte Ausführung, ..

Erkennen von Deadlocks

- **Time-out Strategie:**
  Falls eine TA in einer definierten Zeit (zB 1 Sek.) keinen Fortschritt erzielt, wird von einem Deadlock ausgegangen und die TA zurückgesetzt.

- **Wartegraphen:**
  Knoten=aktive TA,
  Kanten=gerichtet (T_i wartet auf T_j: $T_i \rightarrow T_j$)
  Verklemmung nur dann, wenn Zyklus vorkommt!



Auswahl der rückzusetzenden Transaktion

Kriterien:

- **Minimierung des Rücksetzungsaufwands**
  jüngste Transaktion oder diejenige mit den wenigsten Sperren
- **Maximierung der freigegebenen Ressourcen**
  TA mit den meisten Sperren wählen, um die Gefahr eines nochmaligen Deadlocks zu verkleinern
- **Vermeidung von Starvation (Verhungern)**
  Verhindern, dass immer wieder die gleiche Transaktion zurückgesetzt wird
- **Mehrfache Zyklen**
  Oft sind TA in mehreren Zyklen beteiligt. Werden diese zurückgesetzt können mehrere Verklemmungen auf einmal gelöst werden (siehe Bsp. Vorher, T_2)

Deadlock-Vermeidung durch Zeitstempel

Jeder Transaktion wird ein Zeitstempel zugeordnet (ältere TA hat kleineren Zeitstempel).
Transaktionen warten nicht mehr bedingungslos auf die Freigabe einer Sperre durch eine andere TA, wodurch Deadlocks verhindert werden.

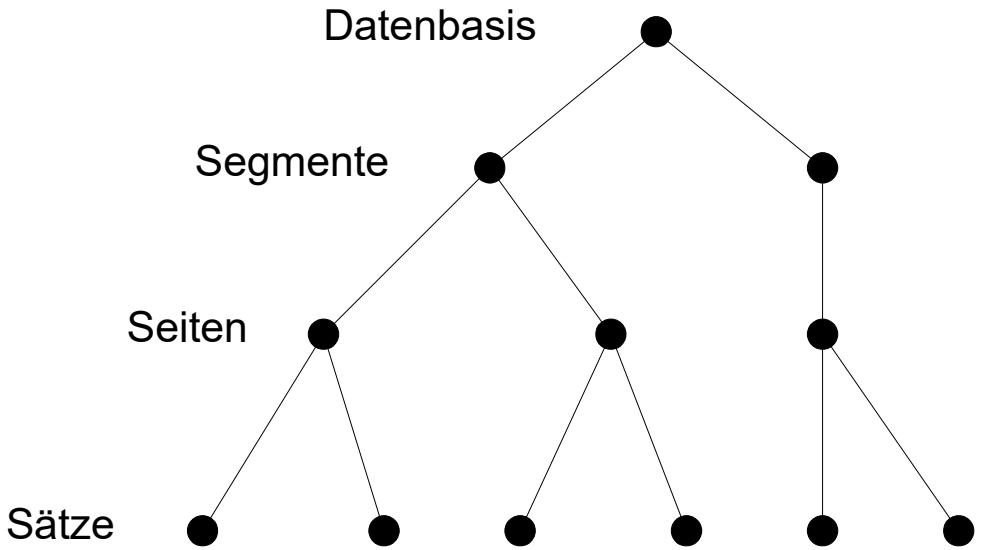
- **wound-wait:**
  Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen und zurückgesetzt, so dass T_1 weiterlaufen kann. Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.

- **wait-die:**
  Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre. Sonst wird T_1 abgebrochen und zurückgesetzt.

Hierarchische Sperrgranulate

- Ausgangsbasis bisher war eine Sperre auf gleicher Ebene (in der Regel der Datensatz)

- Möglichkeit der Sperre auf anderer Ebene
  → kleine Granularität: viele Sperren notwendig, Last
  → große Granularität: Parallelitätsgrad eingeschr.



Hierarchische Sperrgranulate

- Lösung durch Einführung neuer Sperrmodi:
  NL: keine Sperrung (No Lock)
  S: Sperrung durch Leser
  X: Sperrung durch Schreiber
  IS: Intention share: Weiter unten in der Hierarchie ist Lesesperre beabsichtigt
  IX: Intention exclusive: Weiter unten ist eine Schreibsperre beabsichtigt

	NL	S	X	IS	IX
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

Multiple-granularity locking (MGL)

Hierarchische Sperrgranulate

Bei einer Sperrung eines Objekts werden erst geeignete Sperren in allen übergeordneten Knoten erworben (Sperre top-down, Freigabe bottom-up).

Regeln:

1. Bevor ein Knoten mit S od. IS gesperrt wird, müssen alle Vorgänger in der Hierarchie von der TA im IX oder IS-Modus gehalten werden.
2. Bevor ein Knoten mit X oder IX gesperrt wird, müssen alle Vorgänger von der TA im IX-Modus gehalten werden.
3. Die Sperren werden von unten nach oben freigegeben, so dass bei keinem Knoten die Sperre freigegeben wird, wenn die betreffende TA noch Nachfolger dieses Knotens gesperrt hat.

MGL / Zusätzlicher Sperrmodus SIX

SIX: S(hared) + IX (Intention eXclusive)

Kann verwendet werden, wenn (annähernd) alle Sätze eines Satztyps gelesen, aber nur einige geändert werden sollen. X-Sperre wäre hier zu restriktiv und IX würde das Sperren jedes Satzes verlangen.

	NL	S	X	IS	IX	SIX
S	✓	✓	-	✓	-	-
X	✓	-	-	-	-	-
IS	✓	✓	-	✓	✓	✓
IX	✓	-	-	✓	✓	-
SIX	✓	-	-	✓	-	-

Vorteile MGL:
TA mit wenig Änderungen:
→ Sperre auf niedriger Ebene
→ hohe Parallelität

TA mit hohem Datenvolumen:
→ Sperre auf höherer Ebene
→ Reduktion des Sperraufwandes

Zeitstempel-basierende Synchronisation

Ist Verfahren zur Synchronisation das keine Sperren verwendet, sondern auf Basis von Zeitstempel-vergleichen durchgeführt wird.

- Jeder Transaktion wird zu Beginn ein Zeitstempel TS zugewiesen (ältere TA haben kleineren TS).

- Jedes Datum A in der Datenbasis erhält 2 Marken:
  - readTS(A): Zeitstempelwert der jüngsten Transaktion, die dieses Datum A gelesen hat
  - writeTS(A): Zeitstempel der jüngsten Transaktion, die das Datum A geschrieben hat.

Zeitstempel-basierende Synchronisation

Ablauf:

T_i will A **lesen**, also $r_i(A)$:

- Falls $TS(T_i) < writeTS(A)$:
  T_i ist älter als eine andere Transaktion, die A schon geschrieben hat → T_i muss zurückgesetzt werden.

- $TS(T_i) >= writeTS(A)$:
  T_i kann ihre Leseoperation durchführen und readTS(A) wird auf $max(TS(T_i), ReadTS(A))$ gesetzt.

Zeitstempel-basierende Synchronisation

T_i will A **schreiben**, also $w_i(A)$:

- Falls $TS(T_i) < readTS(A)$:
  Eine jüngere Lesetransaktion, die den neuen Wert von A (den T_i gerade schreiben möchte) hätte lesen müssen → T_i muss zurückgesetzt werden.

- $TS(T_i) < writeTS(A)$:
  T_i will einen Wert einer jüngeren Transaktion überschreiben, was nicht passieren darf → T_i msus zurückgesetzt werden.

- Sonst:
  T_i darf das Datum A schreiben und die Marke writeTS(A) wird auf $TS(T_i)$ gesetzt.

Pessimistische / Optimistische Synchronisation

Bisher beschriebene Methoden waren pessimistische Verfahren. Dabei wird von auftretenden Konflikten ausgegangen, die mit unterschiedlichen Methoden verhindert werden sollen. Dies erfolgt oft auf Kosten der Parallelität.

Optimistische Synchronisation geht davon aus, dass Konflikte selten auftreten und deshalb Transkationen einfach ausgeführt werden sollten. Im Nachhinein wird dann entschieden, ob ein Mehrbenutzerkonflikt aufgetreten ist oder nicht.
Der Schedular ist hier eine Art Beobachter.
Ist ein Konflikt aufgetreten wir die Transaktion zurückgesetzt – also nachdem schon alle Schritte erledigt gewesen wären.
Sinnvoll wenn hauptsächlich Lesevorgänge stattfinden.

Optimistische Synchronisation

Ein Beispiel für Optimistische Synchronisation:

Eine TA wird in 3 Phasen unterteilt:

- *Lesephase*:
  Hier werden alle Operationen der TA ausgeführt (auch Änderungen). Gegenüber der Datenbasis tritt die TA nur als Leser auf, da alle gelesenen Daten in lokalen Variablen der TA gespeichert und auch verändert (Schreiboperationen) werden.

- *Validierungsphase*:
  Hier wird unterschieden, ob die TA Konflikte mit anderen TA hatte (Zeitstempel).

- *Schreibphase*:
  Die Änderungen mit positiver Validierung werden in die Datenbank geschrieben.

Optimistische Synchronisation

TA mit gescheiterter Validierung werden zurückgesetzt. Da die Transaktion noch keine Schreiboperationen durchgeführt hat (nur lokale Variablen), können die veränderten Werte noch von keiner anderen TA gelesen worden sein → kein kaskadierendes Rücksetzen nötig!

Validierung:
Es müssen alle Transaktionen Ta betrachten werden, die älter sind als Tj: TS(Ta) < TS(Tj)
Das Alter bezieht sich nicht auf den Transaktionsstart, sondern auf den Eintritt in die Validierungsphase.
Eine der folgenden Bedingungen müssen erfüllt sein:

- Ta war zum Beginn von Tj schon abgeschlossen (inkl. Schreibphase)
- Tj hat keine Datenelemente gelesen, die von Ta geschrieben wurden

Validierung und Schreiben wird ununterbrechbar ausgeführt!