



FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme II

Prof. Dr. Viktor Leis

Professur für Datenbanken und Informationssysteme

Tansaktionsverwaltung

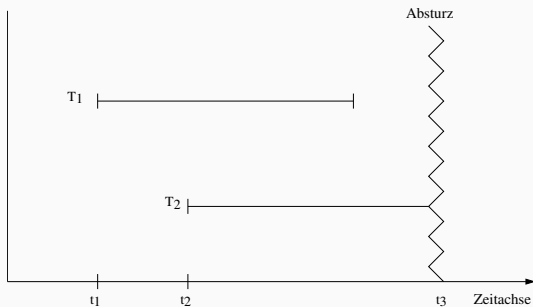
- Transaktionsverwaltung beinhaltet die folgenden zwei Teilgebiete
 - **Recovery**, d.h. die Behebung von eingetretenen, oft unvermeidbaren Fehlersituationen.
 - **Synchronisation** von mehreren gleichzeitig auf der Datenbank ablaufenden Transaktionen.

Beispiel für Transaktion (TA)

- Überweise Geld von Konto A nach Konto B:
 - Lies den Kontostand von A in die Variable a : `read(A,a);`
 - Reduziere den Kontostand um 50 EUR: $a := a - 50$;
 - Schreibe den neuen Kontostand in die Datenbasis:
`write(A,a);`
 - Lies den Kontostand von B in die Variable b : `read(B,b);`
 - Erhöhe den Kontostand um 50 EUR: $b := b + 50$;
 - Schreibe den neuen Kontostand in die Datenbasis:
`write(B,b);`

- **begin of transaction (BOT):**
 - Kennzeichnet den Beginn einer Transaktion
- **commit:**
 - Erfolgreiche Beendigung einer Transaktion
 - Dauerhafte Einbringung aller Änderungen in die Datenbasis
- **abort:**
 - Selbstabbruch der Transaktion, erfolglose Beendigung
 - Zurücksetzen der Datenbasis in den Zustand vor Beginn der Transaktion

- **define savepoint:**
 - Sicherungspunkt definieren, auf den sich die (noch aktive) Transaktion zurücksetzen lässt.
- **backup transaction:**
 - Die noch aktive Transaktion wird auf den jüngsten (zuletzt angelegten) Sicherungspunkt zurückgesetzt
 - Evtl. auch Rücksetzen auf weiter zurückliegende Sicherungspunkte möglich



- Änderungen der zum Zeitpunkt t_3 abgeschlossenen TA T_1 müssen in der Datenbasis vorhanden sein
- Änderungen der zu t_3 noch nicht abgeschlossenen TA T_2 müssen vollständig aus der Datenbasis entfernt werden (Durchführung von T_2 durch Neustart)

- **commit (work):**
 - Beende Transaktion und schreibe Änderungen fest
 - funktioniert nur, wenn keine anderen Fehler aufgetreten sind (z.B. Konsistenzverletzung durch Transaktion)
- **rollback (work):**
 - Beende Transaktion und setze alle Änderungen zurück
 - Anders als **commit** muss DBMS die "erfolgreiche" Ausführung von **rollback** immer garantieren können

- Beispiel:

```
insert into Vorlesungen  
values (5275, 'Kernphysik', 3, 2141);
```

```
insert into Professoren  
values (2141, 'Meitner', 'C4', 205);
```

`commit work`

- **commit** nach dem ersten **insert** könnte nicht erfolgreich durchgeführt werden, da zu diesem Zeitpunkt referentielle Integrität verletzt

- Transaktionen sollten ACID-Eigenschaften haben
- ACID:
 - Atomicity
 - Consistency
 - Isolation
 - Durability

Zusammenfassung (2)

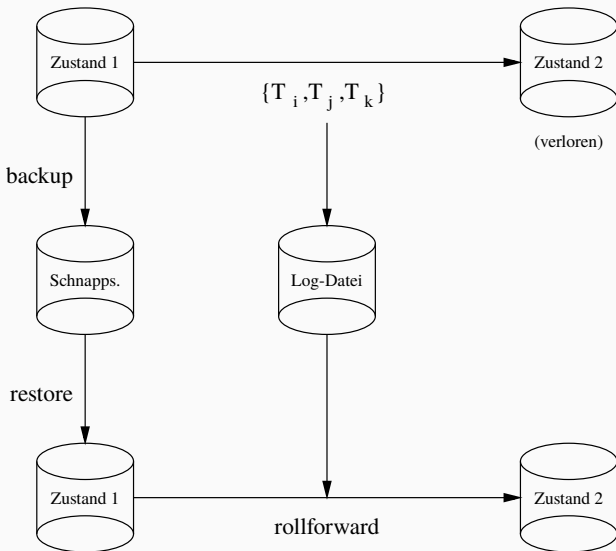
- Atomicity (Atomarität): "alles oder nichts", d.h. entweder werden all Operationen einer TA ausgeführt oder keine
- Consistency (Konsistenz): wenn eine TA auf einem konsistenten Datenbankzustand aufsetzt, ist dieser nach Beendigung der TA immer noch konsistent
- Isolation: nebenläufig ausgeführte Transaktionen dürfen keine Seiteneffekte aufeinander haben
- Durability (Dauerhaftigkeit): alle mit commit festgeschriebenen Änderungen müssen bestehen bleiben (selbst bei Systemabsturz)

Recovery

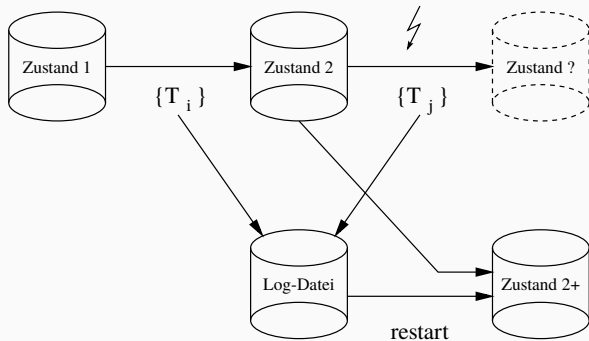
- Wichtige Aufgabe eines DBMS ist das Verhindern von Datenverlust durch Systemabstürze
- Die zwei wichtigsten Mechanismen des Recovery sind:
 - Sicherungspunkte (Backups)
 - Log-Dateien

- Ein *Sicherungspunkt* ist ein Schnappschuss des Datenbankinhalts zu einem bestimmten Zeitpunkt
- In einer *Log-Datei* werden alle Änderungen an der Datenbasis mitprotokolliert
- Offensichtlich sollten Sicherungspunkte und Log-Dateien nicht auf der gleichen Maschine gespeichert werden ...

Vollständiger Verlust



Verlust des Hauptspeichers

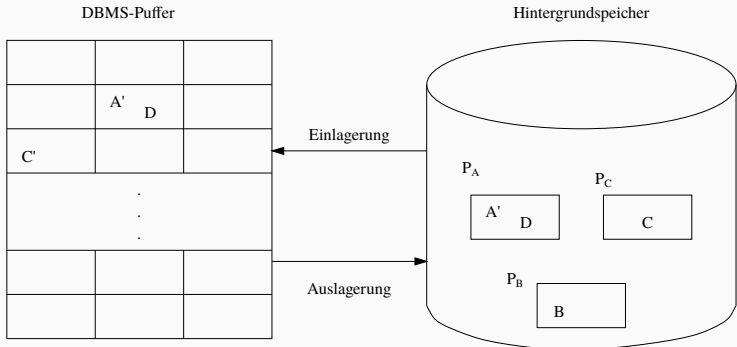


- Problem: einige TAs in $\{T_j\}$ waren noch aktiv, andere haben schon committet
- Restart stellt Zustand 2 + die Änderungen der committeten TAs in $\{T_j\}$ wieder her

- Log-Dateien können auch dazu verwendet werden die Änderungen einer abgebrochenen TA zurückzusetzen
- Nach diesem groben Überblick werfen wir jetzt einen Blick hinter die Kulissen

- Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion
 - Wirkung muss zurückgesetzt werden
 - R1-Recovery
- Fehler mit Hauptspeicherverlust
 - abgeschlossene TAs müssen erhalten bleiben (R2-Recovery)
 - noch nicht abgeschlossene TAs müssen zurückgesetzt werden (R3-Recovery)
- Fehler mit Hintergrundspeicherverlust
 - R4-Recovery

Speicherhierarchie



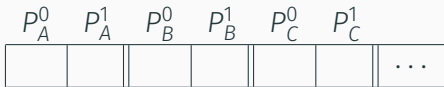
- Ersetzungsstrategien von Puffer-Seiten
 - \neg *steal*: Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ausgeschlossen
 - *steal*: Jede nicht fixierte Seite ist prinzipiell ein Kandidat für die Ersetzung, falls neue Seiten eingelagert werden müssen

- Einbringen von Änderungen abgeschlossener TAs
 - *force*-Strategie: Änderungen werden zum Transaktionsende auf den Hintergrundspeicher geschrieben
 - \neg *force*-Strategie: geänderte Seiten können im Puffer verbleiben und später zurückgeschrieben werden

Auswirkung auf Recovery

	force	\neg force
\neg steal	<ul style="list-style-type: none">• kein Redo• kein Undo	<ul style="list-style-type: none">• Redo• kein Undo
steal	<ul style="list-style-type: none">• kein Redo• Undo	<ul style="list-style-type: none">• Redo• Undo

- Update in Place
 - jede Seite hat genau eine "Heimat" auf dem Hintergrundspeicher
 - der alte Zustand der Seite wird überschrieben
- Twin-Block-Verfahren

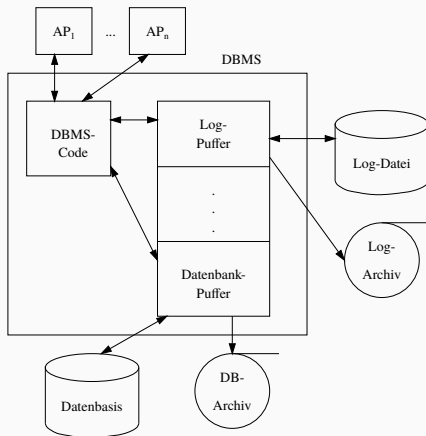


- Schattenspeicherkonzept
 - nur geänderte Seiten werden dupliziert
 - weniger Redundanz als beim Twin-Block-Verfahren

- steal
- \neg force
- update-in-place
- Kleine Sperrgranulate

- ARIES-Protokoll ist weit verbreitetes Protokoll zur Fehlerbehandlung in DBMSen
- Log-Datei enthält:
 - Redo-Information: gibt an, wie Änderungen nachvollzogen werden können
 - Undo-Information: beschreibt, wie Änderungen rückgängig gemacht werden können

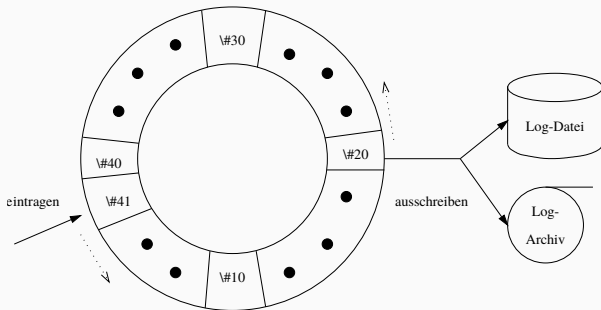
Schreiben der Log-Daten



- Die Log-Information wird zweimal geschrieben
 - Log-Datei für schnellen Zugriff: R1, R2 und R3-Recovery
 - Log-Archiv: R4-Recovery

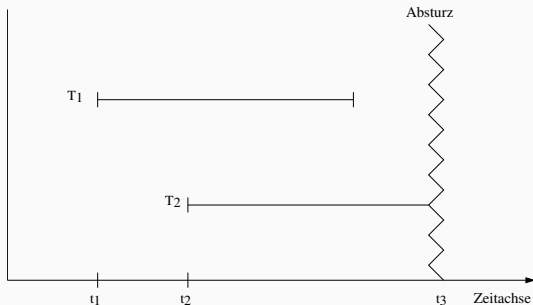
Schreiben der Log-Daten (2)

- Anordnung des Log-Ringpuffers:



- Write Ahead Log-Prinzip (WAL-Prinzip)
 - Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle "zu ihr gehörenden" Log-Einträge ausgeschrieben werden
 - Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden

Wiederanlauf nach Fehler



- TAs der Art T_1 sind *Winner*: müssen vollständig nachvollzogen werden
- TAs der Art T_2 sind *Loser*: müssen rückgängig gemacht werden

- *Analyse:*
 - Ermittlung der *Winner*-Menge von Transaktionen des Typs T_1
 - Ermittlung der *Loser*-Menge von Transaktionen der Art T_2 .
- *Wiederholung der Historie:*
 - *Alle* protokollierten Änderungen werden in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht
- *Undo der Loser:*
 - Die Änderungsoperationen der *Loser*-Transaktionen werden in umgekehrter Reihenfolge ihrer ursprünglichen Ausführung rückgängig gemacht

Phasen des Wiederanlaufs (2)



[LSN,TA,PageID,Redo,Undo,PrevLSN]

- Redo:
 - Physische Protokollierung: After-Image
 - Logische Protokollierung: Code mit dem aus dem Before-Image das After-Image erzeugt werden kann
- Undo:
 - Physische Protokollierung: Before-Image
 - Logische Protokollierung: Code mit dem aus dem After-Image das Before-Image erzeugt werden kann

Struktur der Log-Einträge (2)

- *LSN (Log Sequence Number)*,
 - eine eindeutige Kennung des Log-Eintrags
 - *LSNs* müssen monoton aufsteigend vergeben werden,
 - die chronologische Reihenfolge der Protokolleinträge kann dadurch ermittelt werden
- *TA*
 - Transaktionskennung der Transaktion, die die Änderung durchgeführt hat

Struktur der Log-Einträge (3)

- *PageID*
 - die Kennung der Seite, auf der die Änderungsoperation vollzogen wurde
 - Wenn eine Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden
- *PrevLSN*,
 - Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion
 - Diesen Eintrag benötigt man aus Effizienzgründen

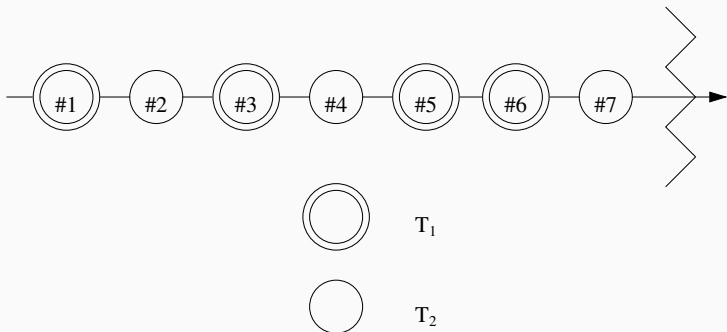
Beispiel einer Log-Datei

	T_1	T_2	Log
			[LSN,TA,PageID,Redo,Undo,PrevLSN]
1.	BOT		[#1, T_1 , BOT, 0]
2.	$r(A, a_1)$		
3.		BOT	[#2, T_2 , BOT, 0]
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A, a_1)$		[#3, T_1 , P_A , $A-=50$, $A+=50$, #1]
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, T_2 , P_C , $C+=100$, $C-=100$, #2]
9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B, b_1)$		[#5, T_1 , P_B , $B+=50$, $B-=50$, #3]
12.	commit		[#6, T_1 , commit , #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, T_2 , P_A , $A-=100$, $A+=100$, #4]
16.		commit	[#8, T_2 , commit , #7]

Fehlertoleranz Wiederanlauf: Idempotenz

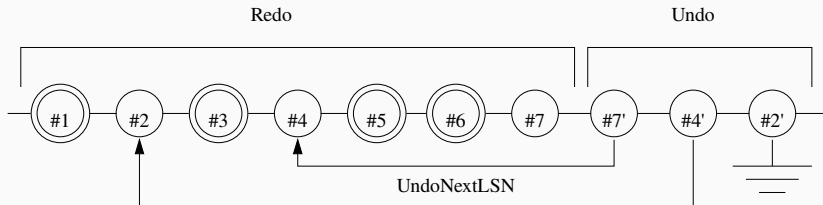
$undo(undo(\dots(undo(a))\dots)) = undo(a)$

$redo(redo(\dots(redo(a))\dots)) = redo(a)$



- Die Idempotenz der Redo-Phase wird dadurch erreicht, dass die LSN des Log-Records, für den ein Redo (tatsächlich) ausgeführt wird, in die Seite eingetragen wird.
- Dadurch wird sichergestellt, dass auch nach einem Absturz während des Wiederanlaufs ein Redo einer Operation nicht “versehentlich” auf dem After-Image der Operation ausgeführt wird.

Fehlertoleranz Wiederanlauf: Idempotenz bei Undo

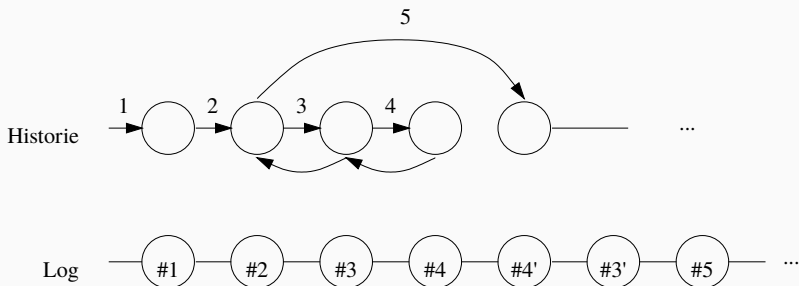


- Kompensationseinträge (CLR: compensating log record) für rückgängig gemachte Änderungen.
- #7' ist CLR für #7
- #4' ist CLR für #4

[#1, T_1 , BOT, 0]
[#2, T_2 , BOT, 0]
[#3, T_1 , P_A , $A-=50$, $A+=50$, #1]
[#4, T_2 , P_C , $C+=100$, $C-=100$, #2]
[#5, T_1 , P_B , $B+=50$, $B-=50$, #3]
[#6, T_1 , **commit**, #5]
[#7, T_2 , P_A , $A-=100$, $A+=100$, #4]
⟨#7', T_2 , P_A , $A+=100$, #7, #4⟩
⟨#4', T_2 , P_C , $C-=100$, #7', #2⟩
⟨#2', T_2 , -, -, #4', 0⟩

- CLRs sind durch spitze Klammern ⟨...⟩ gekennzeichnet

- der Aufbau eines CLR ist wie folgt
 - LSN
 - TA-Identifikator
 - betroffene Seite
 - Redo-Information
 - PrevLSN
 - UndoNxtLSN (Verweis auf die nächste rückgängig zu machende Änderung)

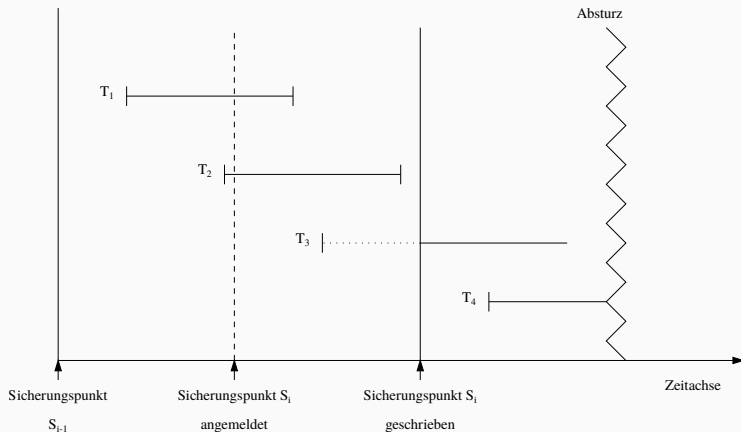


- Schritte 3 und 4 werden zurückgenommen
- notwendig für die Realisierung von Sicherungspunkten innerhalb einer TA

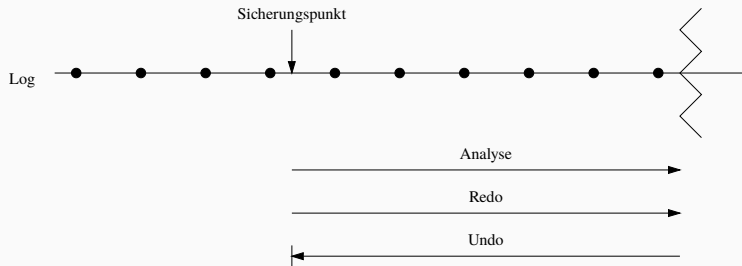
- prinzipiell ist logging ausreichend für ACID
- allerdings würden die Log-Größe und die Recovery-Zeit unbeschränkt wachsen
- deswegen will man periodisch Sicherungspunkte (“checkpoints”) anlegen, die es einem ermöglichen alte Log-Einträge zu ignorieren
- schreibt modifizierte (“dirty”) Seiten auf Platte

Transaktionskonsistenter Sicherungspunkt

- wartet bis laufenden Transaktionen beenden sind
- verzögert neue Transaktionen
- schreibt alle modifizierten Seiten

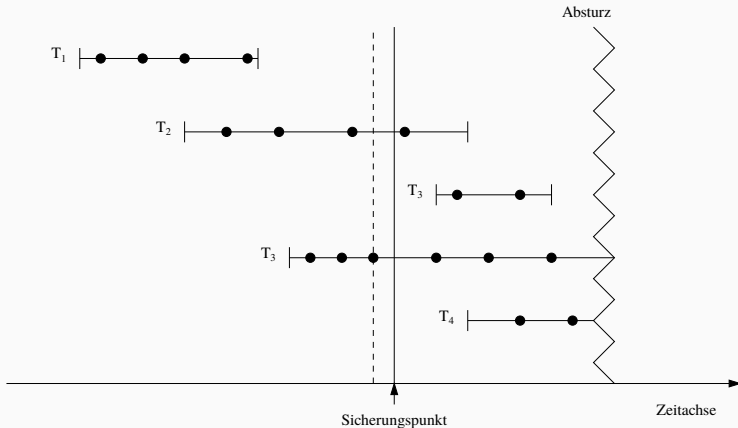


Transaktionskonsistenter Sicherungspunkt (2)



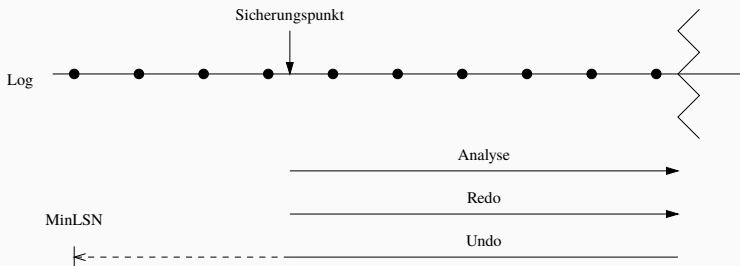
Aktionskonsistenter Sicherungspunkt

- wartet auf elementare Operationen (INSERT, UPDATE, DELETE), verzögert neue Operationen
- schreibt alle modifizierten Seiten



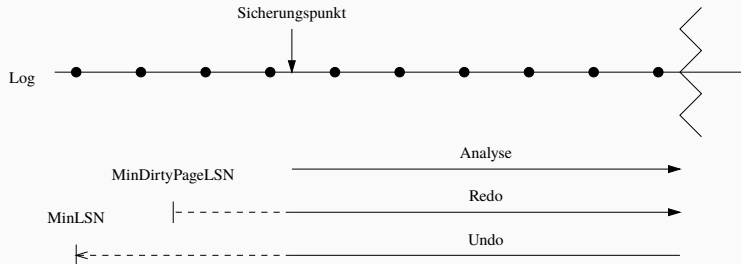
Aktionskonsistenter Sicherungspunkt (2)

- MinLSN: kleinste LSN aller aktiven Transaktionen zum Zeitpunkt des Sicherungspunktes



Unschärfer Sicherungspunkt

- behindert Ausführung normaler Transaktionen nicht



Unscharfer Sicherungspunkt (2)

- modifizierte Seiten werden nicht ausgeschrieben, nur deren Kennung
 - *Dirty Pages*=Menge der modifizierten Seiten
- *MinDirtyPageLSN*: die minimale LSN, deren Änderungen noch nicht ausgeschrieben wurde

Unscharfer Sicherungspunkt (3)

- unscharfe Sicherungspunkte erfordern nicht unbedingt, dass alle modifizierten Seiten geschrieben werden
- trotzdem ist es natürlich sinnvoll dies periodisch zu tun
- insbesondere ist es wichtig “hot-spot” Seiten, die ständig modifiziert werden und deswegen dauerhaft im Puffer verbleiben, periodisch zu schreiben

- Fehlerbehandlung (Recovery) kann Zustand der Datenbasis zum Absturzzeitpunkt wiederherstellen
- Die zwei Hauptmechanismen dazu sind:
 - Log-Dateien
 - Sicherungspunkte
- In der Praxis sind unscharfe Sicherungspunkte zu bevorzugen

Mehrbenutzersynchronisation

- Alle TAs strikt seriell (also nacheinander) auszuführen ist sicher, aber langsam
- Oft werden Systemressourcen nicht voll ausgenutzt, da eine TA auf Plattenzugriff oder Benutzereingabe wartet
- Diese TA blockiert dann alle anderen TAs
- Um Systemressourcen auszunutzen bietet sich Nebenläufigkeit an

- Nicht abgesicherte Nebenläufigkeit kann aber zu folgenden Problemen führen:
 - lost update
 - dirty read
 - non-repeatable read
 - phantom problem

Lost Update

T_1	T_2
bot	
$r_1(x)$	
\hookrightarrow	bot
	$r_2(x)$
$w_1(x)$	\leftarrow
\hookrightarrow	$w_2(x)$
commit	\leftarrow
\hookrightarrow	commit

Das Ergebnis der Transaktion T_1 ist verlorengegangen!

Dirty Read

T_1	T_2
bot	
\hookrightarrow	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	\leftarrow
$w_1(y)$	
commit	
\hookrightarrow	abort

T_1 liest einen Wert für x der so nicht gültig ist!

Non-Repeatable Read

T_1	T_2
bot	
$r_1(x)$	
\leftrightarrow	bot
	$w_2(x)$
	commit
$r_1(x)$	\leftarrow
...	

T_1 liest x zweimal mit verschiedenem Ergebnis!

Phantom Problem

T_1	T_2
bot	
select count(*)	
from R;	
↔	bot
	insert into R ...;
	commit
select count(*)	←
from R;	
...	

T_1 findet ein weiteres Tupel beim Abarbeiten der zweiten Anfrage!

- Im Idealfall sollten alle diese Probleme vermieden werden
- Man muss dabei allerdings einen Kompromiß zwischen Performanz und Genauigkeit schließen
- Je mehr Sicherheit, desto langsamer wird die Ausführung
- Über die *Isolation Levels* kann man DBMS mitteilen, welche Sicherheit erwünscht ist

- Festsetzen von Eigenschaften einer TA:
 set transaction isolation level *Stufe, Zugriffsmodus*
- Folgende Stufen für den Isolation Level sind möglich:
 - read uncommitted
 - read committed
 - repeatable read
 - serializable
- Achtung: in vielen Systemen ist read committed die Standardeinstellung
- Mögliche Zugriffsmodi:
 - read only
 - read write

Transaktionen und SQL (2)

	lost update	dirty read	nonrep. read	phant. probl.
read uncommit.	✓			
read committed	✓	✓		
repeat. read	✓	✓	✓	
serializable	✓	✓	✓	✓

- “read only” sagt dem DBMS, dass eine TA nur Leseoperationen enthält
- Das hat Auswirkungen auf die Performanz
- Nebenläufiges Ausführen von TAs die nur lesen ist unkritisch, d.h. beliebig vieler solcher TAs können völlig uneingeschränkt parallel laufen
- Erst wenn eine TA dazukommt, die auch schreibt müssen Vorkehrungen getroffen werden

- Befehl zum Markieren eines Transaktionsbeginns
`start transaction;` oder `begin;`
- Befehl zur erfolgreichen Beendigung
`commit [work];`
- Befehl zum Abbruch
`rollback [work];`

Bonusanomalie: Write Skew (“Serialization Anomaly”)

T_1	T_2
bot	
\hookrightarrow	bot
	$p := r_2(x)$
	$w_2(y, p)$
$q := r_1(y)$	\leftarrow
$w_1(x, q)$	
commit	
\hookrightarrow	commit?

Bei einer beliebigen seriellen Ausführung ist $x = y$.

Write Skew und Snapshot Isolation

- Viele Systeme (z.B. Oracle) lassen obige Ausführung auch im **serializable** Modus zu
- Ergebnis ist, dass die Werte x und y vertauscht werden
- Solche Systeme implementieren “Snapshot Isolation”, bei dem Leser keine Sperren halten
- Es ist nicht klar, ob dieses Verhalten dem SQL Standard entspricht
- Seit Version 9.5 machte PostgreSQL es “richtig”, d.h., eine der beiden Transaktionen wird abgebrochen

- Um Lösungsansätze besser verstehen zu können, wird das Problem zunächst etwas formaler betrachtet
- Danach werden Lösungen vorgestellt, die in DBMS eingesetzt werden

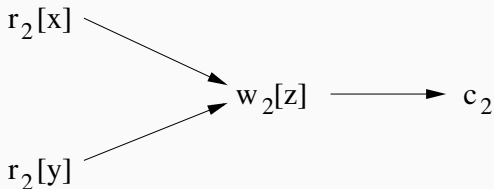
- Operationen einer TA T_i
 - $r_i(A)$: Lesen des Datenobjekts A
 - $w_i(A)$: Schreiben des Datenobjekts A
 - a_i : Abbruch
 - c_i : erfolgreiche Beendigung

- *bot*: begin of transaction (implizit)

Formale Definition einer TA (2)

- Eine TA T_i ist eine partielle Ordnung von Operationen mit der Ordnungsrelation $<_i$ so dass:
 - $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ ist ein Datenobjekt}\} \cup \{a_i, c_i\}$
 - $a_i \in T_i$, gdw. $c_i \notin T_i$
 - Sei t gleich a_i oder c_i , dann gilt für jede andere Operation p_i : $p_i <_i t$
 - Falls $r_i[x]$ und $w_i[x] \in T_i$, dann gilt entweder $r_i[x] <_i w_i[x]$ oder $w_i[x] <_i r_i[x]$

- Transaktionen werden oft als gerichtete azyklische Graphen (DAGs) dargestellt:



- $r_2[x] <_2 w_2[z]$, $w_2[z] <_2 c_2$, $r_2[x] <_2 c_2$, $r_2[y] <_2 w_2[z]$,
 $r_2[y] <_2 c_2$
- Transitive Beziehungen sind im Graph implizit enthalten

- Mehrere TAs können nebenläufig ausgeführt werden
- Dies wird durch eine *Historie* (Schedule) beschrieben
- Eine Historie gibt an, wie Operationen aus verschiedenen TAs relativ zueinander ausgeführt werden
- Da verschiedene Operationen parallel ausgeführt werden können, ist eine Historie eine partielle Ordnung

Konfliktoperationen

- Operationen, die in Konflikt miteinander stehen, dürfen nicht parallel ausgeführt werden
- Zwei Operationen stehen in Konflikt miteinander, wenn beide auf dem gleichen Datenobjekt arbeiten und mindestens eine davon eine Schreiboperation ist

	T_i	
T_j	$r_i[x]$	$w_i[x]$
$r_j[x]$		\neg
$w_j[x]$	\neg	\neg

- Sei $T = \{T_1, T_2, \dots, T_n\}$ eine Menge von Transaktionen
- Eine Historie H über T ist eine partielle Ordnung mit der Ordnungsrelation $<_H$, so dass
 - $H = \bigcup_{i=1}^n T_i$
 - $<_H \supseteq \bigcup_{i=1}^n <_i$
 - Für zwei beliebige Operationen $p, q \in H$ die in Konflikt miteinander stehen gilt: entweder $p <_H q$ oder $q <_H p$

Beispiel einer Historie

$$H = \begin{array}{ccccccc} & & r_2[X] \rightarrow & w_2[Y] \rightarrow & w_2[Z] \rightarrow & C_2 & \\ & & \uparrow & \uparrow & \uparrow & & \\ r_3[Y] \rightarrow & w_3[X] \rightarrow & w_3[Y] \rightarrow & w_3[Z] \rightarrow & C_3 & & \\ & \uparrow & & & & & \\ r_1[X] \rightarrow & w_1[X] \rightarrow & C_1 & & & & \end{array}$$

(Konflikt-)Äquivalenz

- Zwei Historien H und H' sind (*konflikt-äquivalent*) ($H \equiv H'$), wenn:
 - Sie enthalten die gleichen Mengen von TAs (samt allen dazugehörigen Operationen)
 - Sie ordnen die Konfliktoperationen der nicht abgebrochenen TAs in der gleichen Art und Weise an
- Die Idee dabei ist, das berechnete Endergebnis nicht zu verändern

$$\begin{aligned} & r_1[X] \rightarrow w_1[Y] \rightarrow r_2[Z] \rightarrow c_1 \rightarrow w_2[Y] \rightarrow c_2 \\ \equiv & r_1[X] \rightarrow r_2[Z] \rightarrow w_1[Y] \rightarrow c_1 \rightarrow w_2[Y] \rightarrow c_2 \\ \equiv & r_2[Z] \rightarrow r_1[X] \rightarrow w_1[Y] \rightarrow w_2[Y] \rightarrow c_2 \rightarrow c_1 \\ \neq & r_2[Z] \rightarrow r_1[X] \rightarrow w_2[Y] \rightarrow w_1[Y] \rightarrow c_2 \rightarrow c_1 \end{aligned}$$

- Da serielle Historien sicher sind, ist es wünschenswert Historien mit ähnlichen Eigenschaften zu haben
- Insbesondere möchte man eine Historie haben die äquivalent zu einer seriellen Historie ist
- Eine solche Historie nennt man *serialisierbar*

- Präzise Definition:
 - Die *abgeschlossene Projektion* $C(H)$ einer Historie H enthält nur die erfolgreich abgeschlossenen TAs
 - Eine Historie H ist serialisierbar, wenn $C(H)$ äquivalent zu einer seriellen Historie H_s ist

- Wie überprüft man die Serialisierbarkeit?
- Eine Historie ist genau dann serialisierbar, wenn ihr *Serialisierbarkeitsgraph* $SG(H)$ azyklisch ist

- Der Serialisierbarkeitsgraph $SG(H)$ einer Historie $H = \{T_1, \dots, T_n\}$ ist ein gerichteter Graph mit folgenden Eigenschaften:
 - Die Knoten sind die erfolgreich abgeschlossenen TAs aus H
 - Eine Kante zwischen zwei TAs T_i und T_j wird eingetragen, wenn es zwei Konfliktoperationen p_i und q_j gibt und $p_i <_H q_j$

Beispiel

- Historie H

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$$

- $SG(H)$

$$SG(H) = T_1 \begin{array}{l} \nearrow T_2 \\ \searrow T_3 \end{array}$$

Beispiel (2)

- H ist serialisierbar
- Mögliche Ordnungen

$$H_S^1 = T_1 \mid T_2 \mid T_3$$

$$H_S^2 = T_1 \mid T_3 \mid T_2$$

$$H \equiv H_S^1 \equiv H_S^2$$

Beispiel (3)

$$H = \begin{array}{ccccccc} r_1[X] & \rightarrow & w_1[X] & \rightarrow & w_1[Y] & \rightarrow & C_1 \\ & & \uparrow & & \uparrow & & \\ & \swarrow & r_2[X] & \rightarrow & w_2[Y] & \rightarrow & C_2 \\ & & \downarrow & & & & \\ r_3[X] & \rightarrow & w_3[X] & \rightarrow & & & C_3 \end{array}$$

$$SG(H) = \begin{array}{ccc} & & T_3 \\ & \nearrow & \\ T_2 & & \uparrow \\ & \searrow & \\ & & T_1 \end{array}$$

Beispiel (4)

- H ist serialisierbar
- Mögliche Ordnung

$$H_S^1 = T_2 \mid T_1 \mid T_3$$
$$H \equiv H_S^1$$

Beispiel (5)

$$H = \begin{array}{ccccc} w_1[x] & \rightarrow & w_1[y] & \rightarrow & C_1 \\ \uparrow & & \downarrow & & \\ r_2[x] & \rightarrow & w_2[y] & \rightarrow & C_2 \end{array}$$

$$SG(H) = T_1 \not\leftrightarrow T_2$$

- H ist nicht serialisierbar

- Für TAs sind weitere Eigenschaften wünschenswert:
 - Rücksetzbarkeit (Recoverability)
 - Vermeidung kaskadierenden Rücksetzens (avoiding cascading aborts: ACA)
 - Striktheit (strictness)

Weitere Eigenschaften (2)

- Zuerst müssen wir Schreib-/Leseabhängigkeiten (reads-from relationship) definieren
- Eine TA T_i liest (Datenobjekt x) von TA T_j , wenn
 - $w_j[x] < r_i[x]$
 - $a_j \not< r_i[x]$
 - Falls ein $w_k[x]$ existiert mit $w_j[x] < w_k[x] < r_i[x]$, dann $a_k < r_i[x]$
- Eine TA kann auch von sich selbst lesen

- Eine Historie ist *rücksetzbar*, wenn folgendes gilt
 - Immer wenn eine TA T_i von einer anderen TA T_j liest ($i \neq j$) und $c_i \in H$, dann $c_j < c_i$
- Die TAs müssen eine bestimmte Commit-Reihenfolge einhalten
- Bei nicht rücksetzbaren Historien können Probleme mit dem C und D der ACID-Eigenschaften auftreten

$$H = w_1[x] r_2[x] w_2[y] c_2 a_1$$

- H ist nicht rücksetzbar
- Die Konsequenzen sind:
 - Wenn Ergebnis von T_2 so stehen bleibt, dann haben wir inkonsistente Daten (T_2 hat Daten von einer abgebrochenen TA gelesen)
 - Wenn wir T_2 zurücksetzen, dann nehmen wir Änderungen einer fest zugesicherten TA zurück

Kaskadierendes Rücksetzen

Schritt	T_1	T_2	T_3	T_4	T_5
0.	...				
1.	$w_1[x]$				
2.		$r_2[x]$			
3.		$w_2[y]$			
4.			$r_3[y]$		
5.			$w_3[z]$		
6.				$r_4[z]$	
7.				$w_4[v]$	
8.					$r_5[v]$
9.	a_1 (abort)				

- Eine Historie *vermeidet kaskadierendes Rücksetzen*, wenn folgendes gilt
 - Immer wenn eine TA T_i von einer anderen TA T_j liest ($i \neq j$), dann $c_j < r_i[x]$
- Es darf nur von bereits erfolgreich abgeschlossenen TAs gelesen werden

- Eine Historie ist *strikt*, wenn folgendes gilt
 - Bei zwei Operationen $w_j[x] < o_i[x]$ (mit $o_i[x] = r_i[x]$) oder $w_i[x]$ gilt entweder $a_j < o_i[x]$ oder $c_j < o_i[x]$
- Nur von bereits erfolgreich abgeschlossenen TAs darf gelesen oder dürfen Datenobjekte überschrieben werden

Striktheit (2)

- Nur bei strikten Historien darf physische Protokollierung beim Recovery angewendet werden

$$x = 0$$

$w_1[x, 1]$ before image von T_1 : 0

$$x = 1$$

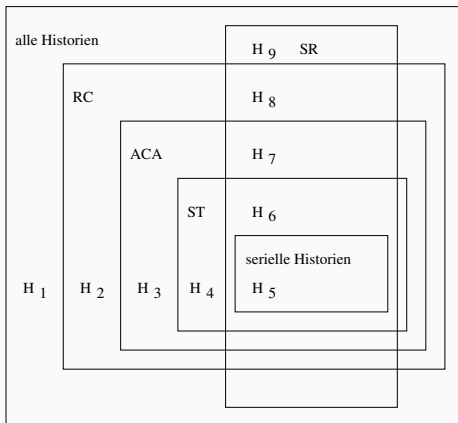
$w_2[x, 2]$ before image von T_2 : 1

$$x = 2$$

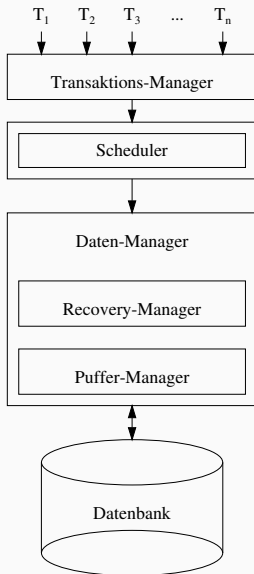
a_1

c_2

Bei Abbruch von T_1 wird x fälschlicherweise auf 0 gesetzt



SR: serialisierbar, RC: rücksetzbar, ACA: vermeidet kaskadierendes Rücksetzen, ST: strikt



- Ein *Scheduler* ist ein Programm, das die eingehenden Operationen ordnet und für eine serialisierbare und rücksetzbare Historie sorgt
- Mehrere Möglichkeiten nach Entgegennahme einer Operation:
 - (Sofort) ausführen
 - Zurückweisen
 - Verzögern

- Es existieren zwei grobe Strategien:
 - Pessimistisch
 - Optimistisch

- Scheduler verzögert entgegengenommene Operationen
- Wenn mehrere Operationen da sind, legt Scheduler möglichst geschickte Reihenfolge fest
- Wichtigster Vertreter: Sperrbasierter Scheduler (in der Praxis weit verbreitet)

- Scheduler schickt entgegenkommene Operationen möglichst schnell zur Ausführung,
- Muss später eventuell "Schaden" reparieren

- Hauptidee relativ einfach:
 - Jedes Datenobjekt hat eine zugehörige Sperre
 - Bevor eine TA T_i zugreifen darf, muss sie Sperre anfordern
 - Falls eine andere TA T_j Sperre hält, bekommt T_i die Sperre nicht und muss warten, bis T_j die Sperre freigegeben hat
 - Nur eine TA kann Sperre halten und auf Datenobjekt zugreifen
- Wie garantiert man Serialisierbarkeit?

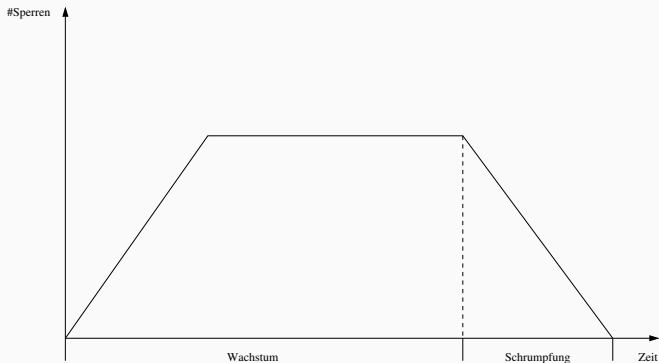
Zwei-Phasen-Sperrprotokoll

- Abgekürzt durch 2PL
- Zwei Sperrmodi:
 - S (shared, read lock, Lesesperre)
 - X (exclusive, write lock, Schreibsperre)
 - Verträglichkeitsmatrix (auch Kompatibilitätsmatrix genannt):

angeford. Sp.	gehaltene Sperre		
	keine	S	X
S	✓	✓	-
X	✓	-	-

- Jedes Objekt, das von einer TA benutzt werden soll, muss vorher entsprechend gesperrt werden
- Eine TA kann eine Sperre die sie hält nicht noch einmal anfordern
- Wenn eine Sperre nicht gewährt werden kann (nach Matrix), wird TA in Warteschlange eingereiht
- Eine TA darf nach der ersten Freigabe einer Sperre keine weitere anfordern (es gibt zwei Phasen)
- Bei Transaktionsende muss eine TA alle Sperren zurückgeben

Zwei Phasen



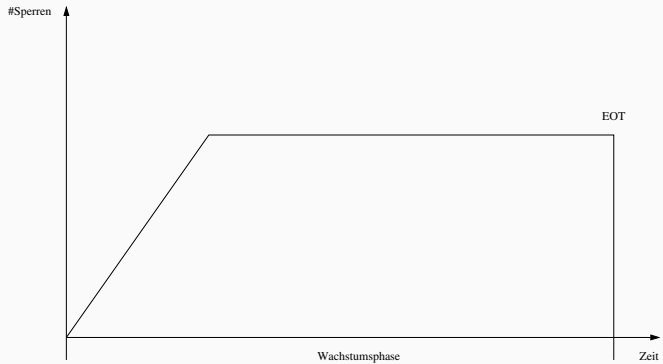
- Wachstumsphase: es werden Sperren angefordert, aber keine freigegeben
- Schrumpfungsphase: es werden Sperren freigegeben, aber keine angefordert

Verzahnung nach 2PL

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX[x]		
3.	r[x]		
4.	w[x]		
5.		BOT	
6.		lockS[x]	T_2 muss warten
7.	lockX[y]		
8.	r[y]		
9.	unlockX[x]		T_2 wecken
10.		r[x]	
11.		lockS[y]	T_2 muss warten
12.	w[y]		
13.	unlockX[y]		T_2 wecken
14.		r[y]	
15.	commit		
16.		unlockS[x]	
17.		unlockS[y]	
18.		commit	

- 2PL schließt kaskadierendes Rücksetzen nicht aus und lässt sogar nicht-rücksetzbare Historien zu
- Erweiterung zum *strengen* 2PL:
 - alle Sperren werden bis zum Ende der Transaktion gehalten
 - damit ist kaskadierendes Rücksetzen ausgeschlossen (die erzeugten Schedules sind sogar strikt)

Strenges 2PL (2)



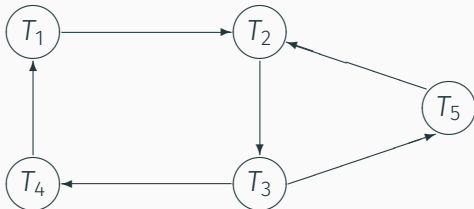
Verklemmungen (Deadlocks)

- Beispiel für ein Deadlock:

T_1	T_2
bot	
lockX ₁ (a)	
w ₁ (a)	
↔	bot
	lockS ₂ (b)
	r ₂ (b)
lockX ₁ (b)	↔
↔	lockS ₂ (a)

- Keine TA soll "ewig" auf eine Sperre warten
- Eine Strategie zum Erkennen von Deadlocks ist Time-Out
 - Richtige Zeitdauer zu finden ist problematisch
- Präzise Methode benutzt Wartegraphen
 - Knoten sind TAs, Kanten sind Wartet-auf-Beziehungen
 - Wenn Graph Zyklen aufweist, liegt ein Deadlock vor

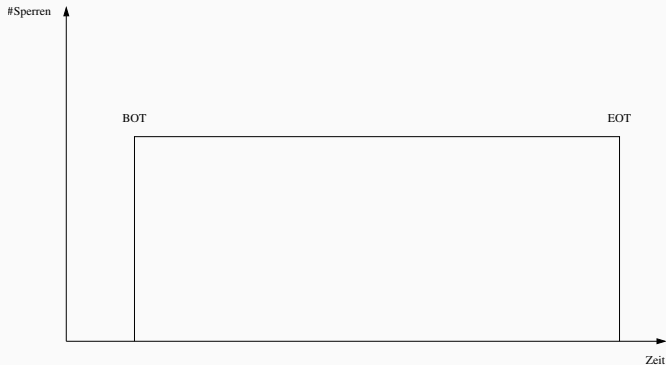
- Beispiel



- Wartegraph hat Zyklen, d.h. Deadlock liegt vor
- Zyklen können hier durch Zurücksetzen von T_2 oder T_3 aufgelöst werden

Deadlock-Vermeidung

- Deadlocks können durch *Preclaiming* vermieden werden
- Preclaiming bedeutet, dass alle Sperrern zu Beginn einer TA angefordert werden
- In der Praxis unrealistisch



Deadlock-Vermeidung (2)

- Time-Out-Verfahren ist oft zu vorsichtig, z.T. werden TAs abgebrochen, wenn nur der Verdacht auf ein Deadlock besteht
- Eine andere Methode besteht darin zum Zeitpunkt wenn T_i eine Sperre anfordert, die von T_j gehalten wird, zu entscheiden
 - TAs bekommen Prioritäten zugewiesen
 - Wenn Priorität von T_i höher ist, darf T_i warten
 - Wenn Priorität von T_i kleiner ist, wird T_i abgebrochen

Deadlock-Vermeidung (3)

- Durch Prioritäten wird vermieden, dass durch ein Warten ein Deadlock entstehen kann
- Prioritätenvergabe muss umsichtig erfolgen
 - Wenn eine abgebrochene TA T_i beim Neustart ständig niedrige Prioritäten erhält, können sich immer TAs mit höheren Prioritäten "vordrängeln"
 - T_i kommt nie zum Zug, wir haben kein Deadlock, aber ein *Livelock*

Deadlock-Vermeidung (4)

- Vermeidung von Deadlocks und Livelocks: Verwendung von Zeitstempeln als Prioritäten
- Zeitstempel sind eindeutig und wachsen monoton mit der Zeit
- Eine TA bekommt beim ersten Aufruf einen Zeitstempel t_s zugewiesen, beim Neustart behält sie den alten Zeitstempel
- Je älter der Zeitstempel, desto höher die Priorität
- Irgendwann hat eine immer wieder abgebrochene TA den ältesten Zeitstempel, Livelocks werden verhindert

Deadlock-Vermeidung (5)

- Angenommen T_j hält Sperre, T_i fordert sie an
- Im Zeitstempelfverfahren kann ein Scheduler nun zwei verschiedene Strategien fahren
 - Wait-Die:
Falls $ts(T_i) < ts(T_j)$,
dann wartet T_i , sonst bricht T_i ab
 - Wound-Wait:
Falls $ts(T_i) < ts(T_j)$,
dann bricht T_j ab, sonst wartet T_i

Phantom-Problem

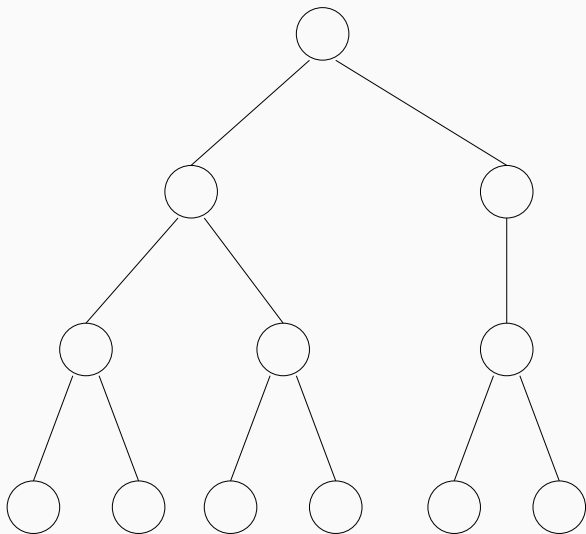
- Mit strengem 2PL haben wir alle am Anfang des Kapitels angesprochenen Probleme gelöst, außer des Phantom-Problems
- Phantom-Problem lässt sich mit Sperren auf Datenobjekten nicht lösen, da keine Sperren auf nichtexistenten Datenobjekten angefordert werden können
- Außerdem kann die Anzahl der Sperren bei großen Transaktionen (z.B. kompletter Scan einer Tabelle) sehr groß sein
- Lösung der Probleme durch *hierarchische Sperrgranulate* (multi-granularity locking: MGL)

Datenbasis

Segmente

Seiten

Sätze



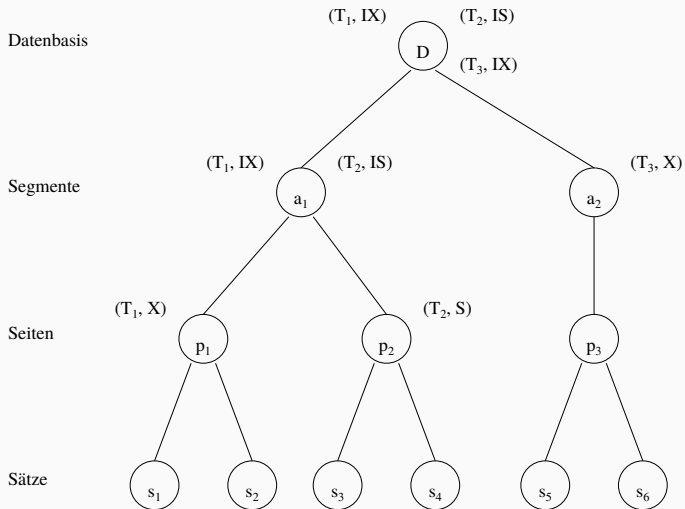
- *S* (shared): für Leser
- *X* (exclusive): für Schreiber
- *IS* (intention share): für beabsichtigtes Lesen weiter unten in der Hierarchie
- *IX* (intention exclusive): für beabsichtigtes Schreiben weiter unten in der Hierarchie

Kompatibilitätsmatrix

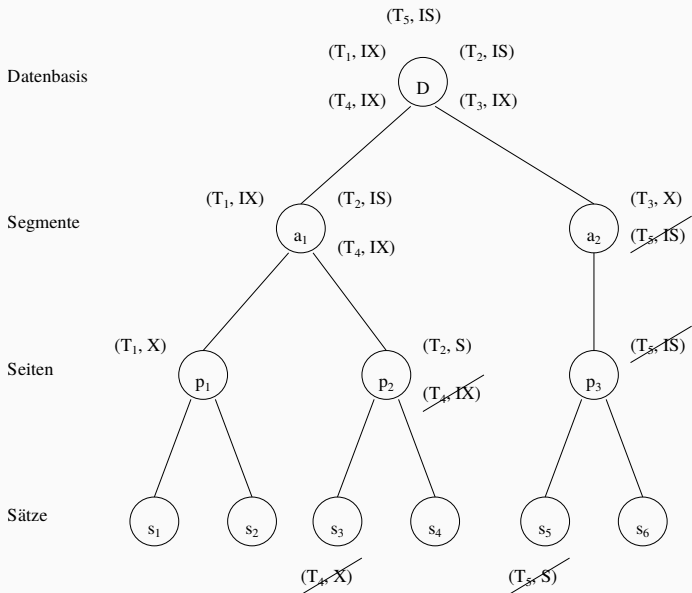
angef. Sp.	gehaltene Sperre				
	keine	S	X	IS	IX
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

- Sperren werden in der Hierarchie von oben nach unten angefordert
 - Für eine *S* oder *IS* Sperre müssen alle Vorgänger in der Hierarchie im *IS* oder *IX* Modus gesperrt sein
 - Für eine *X* oder *IX* Sperre müssen alle Vorgänger in der Hierarchie im *IX* Modus gehalten werden
- Sperren werden von unten nach oben wieder freigegeben (Sperre wird nur freigegeben, wenn auf keinem Nachfolger des Knotens noch eine Sperre gehalten wird)

Beispiel



Beispiel (2)



Beispiel (3)

- TAs T_4 und T_5 sind blockiert
- Hier noch kein Deadlock, aber im einfachen MGL-Protokoll sind auch Deadlocks möglich

- Neben den sperrbasierten Protokollen gibt es noch eine weitere große Klasse an Protokollen, die zeitstempelbasierte Synchronisation
- Transaktionsmanager weist jeder TA einen eindeutigen Zeitstempel zu
- Jede Operation der TA bekommt diesen Zeitstempel

- Ein Scheduler benutzt die Zeitstempel um in Konflikt stehende Operationen zu ordnen:
 - Angenommen $p_i[x]$ und $q_j[x]$ stehen in Konflikt miteinander
 - $p_i[x]$ wird vor $q_j[x]$ ausgeführt, gdw. der Zeitstempel von T_i älter als der Zeitstempel von T_j ist

Zeitstempel (2)

- Scheduler speichert zu jedem Datenobjekt x den Zeitstempel der letzten auf x ausgeführten Operation
- Das wird für jeden Operationstypen q gemacht:
 $\text{max-}q\text{-scheduled}(x)$
- Wenn Scheduler eine Operation p bekommt, wird ihr Zeitstempel mit allen $\text{max-}q\text{-scheduled}(x)$ verglichen, mit denen p in Konflikt steht
- Wenn der Zeitstempel von p älter als ein $\text{max-}q\text{-scheduled}(x)$ ist, wird p zurückgewiesen (und TA abgebrochen)
- Ansonsten wird p ausgeführt und $\text{max-}p\text{-scheduled}(x)$ aktualisiert

- Einfaches Zeitstempelverfahren erzeugt u.U. nicht rücksetzbare Schedules
- Rücksetzbarkeit kann dadurch garantiert werden, dass TAs in Zeitstempelreihenfolge committen
- Solange noch TAs laufen, von denen eine TA T_i gelesen hat, wird ein commit von T_i verzögert

- Zeitstempelbasierte Synchronisation wird in der Praxis kaum eingesetzt
- Phantom-Problem wird nicht gelöst
- Jede Operation wird praktisch zur Schreiboperation, da immer die $\text{max-q-scheduled}(x)$ -Felder aktualisiert werden müssen

- Transaktion wird “einfach mal ausgeführt” und im Nachhinein wird entschieden, ob es einen Konflikt gab
- Annahme: Konflikte sind selten

1. Lesephase: Transaktion wird ausgeführt, Lesezugriffe werden protokolliert ("read set"), Änderungen werden in lokalen Variablen zwischengespeichert ("write set")
2. Validierungsphase: Es wird geprüft, ob es einen Konflikt gab
3. Schreibphase: Änderungen werden in die Datenbank eingebracht

- Zu Beginn der Validierungsphase einer wird der Transaktion T_j ein Zeitstempel $TS(T_j)$ zugeteilt
- Für alle Transaktionen T_a , die älter sind ($TS(T_a) < TS(T_j)$), und noch nicht vollständig abgeschlossen sind (inklusive Schreibphase) muss gelten:
 $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
- Validierungs- und Schreibphase müssen “atomar” durchgeführt werden

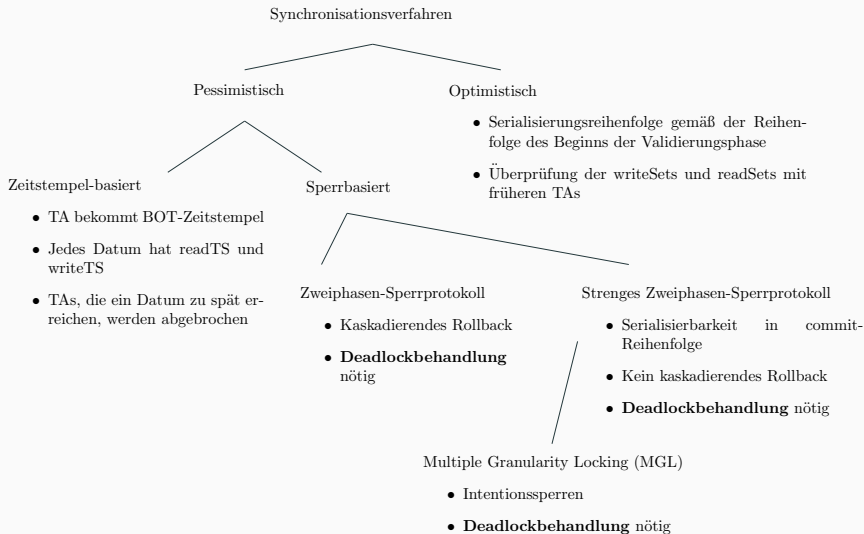
- Kann unnötige Verzögerung anderer Transaktionen vermeiden
- Wenn Konflikte recht häufig sind, kann es zu häufigen Abbrüchen kommen

Multi-Version Concurrency Control (MVCC)

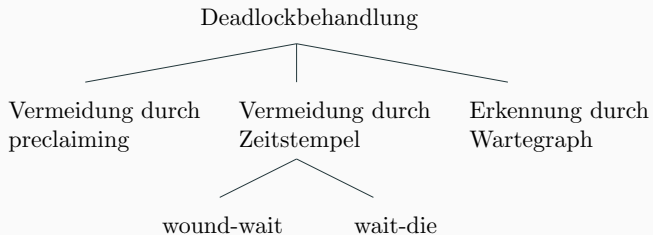
- Alte Versionen von Tupeln können genutzt werden um Mehrbenutzersynchronisation zu verbessern (“Multi-Version Concurrency Control”)
- Entweder werden alte Versionen aus der Log-Datei wiederhergestellt oder Änderung erzeugen immer neue Tupel-Versionen (kein update in place, jedes Tupel hat ein Versionsfeld)

- Transaktionen laufen logisch auf einem Snapshot der Daten, indem sie nur Tupel-Versionen zum Startzeitpunkt ansehen
- Vorteil: Leser müssen nicht sperren, Schreibsperrern sind ausreichend
- $WriteSet(T_a) \cap WriteSet(T_j) = \emptyset$
- Write Skew Anomalie möglich
- Alter Versionen müssen irgendwann aufgeräumt werden (“Garbage Collection”)

Übersicht Synchronisationsverfahren



Übersicht Deadlockbehandlung



- Mehrbenutzersynchronisation gehört mit zu den wichtigsten Funktionen eines DBMS
- Normalerweise bleibt dies den Benutzern verborgen, aber über die Einstellung der Isolation-Levels kann in die Qualität dieser Synchronisation eingegriffen werden
- Bekannte Verfahren:
 - Sperrbasierte Synchronisation
 - Zeitstempelbasierte Synchronisation
 - Optimistische Synchronisation
 - MVCC