TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

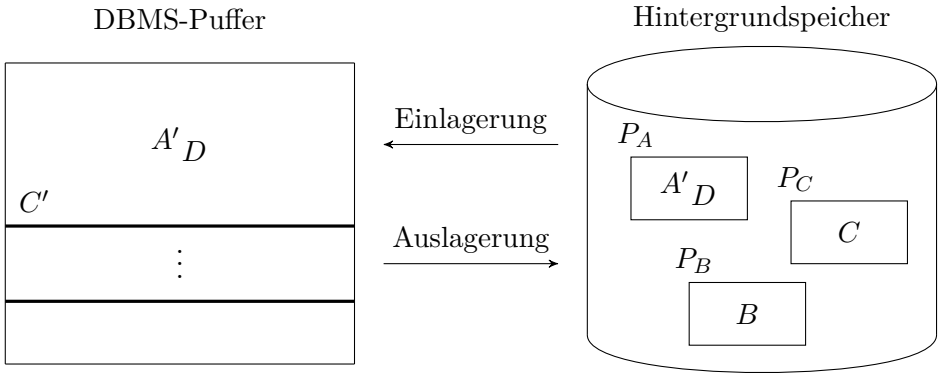
Übung zur Vorlesung *Grundlagen: Datenbanken* im WS21/22

Michael Jungmair, Josef Schmeißer, Moritz Sichert, Lukas Vogel (gdb@in.tum.de)
https://db.in.tum.de/teaching/ws2122/grundlagen/

Blatt Nr. 13

Hausaufgabe 1

Demonstrieren Sie anhand eines Beispiels, dass man die Strategien *force* und ¬*steal* nicht kombinieren kann, wenn parallele Transaktionen gleichzeitig Änderungen an Datenobjekten innerhalb einer Seite durchführen. Betrachten Sie dazu z.B. die unten dargestellte Seitenbelegung, bei der die Seite P_A die beiden Datensätze A und D enthält. Entwerfen Sie eine verzahnte Ausführung zweier Transaktionen, bei der eine Kombination aus *force* und ¬*steal* ausgeschlossen ist.



Lösung:

Folgendes Beispiel zeigt, warum man *force* und ¬*steal* nicht kombinieren kann:

Schritt	T_1	T_2
1.	**BOT**	
2.		**BOT**
3.	read(A)	
4.		read(D)
5.		write(D)
6.	write(A)	
7.	**commit**	

In Schritt 7 führt T_1 einen commit aus. Aufgrund der *force*-Strategie müssen nun alle von dieser Transaktion geänderten Seiten ausgelagert werden. Im Beispiel hat T_1 nur P_A geändert, also muss diese ausgelagert werden. Gleichzeitig existiert aber noch eine laufende Transaktion T_2, die ebenfalls die Seite P_A verändert hat. Wegen der ¬*steal*-Strategie dürfen keine Seiten ausgelagert werden, die von noch nicht beendeten Transaktionen bearbeitet

wurden. Im Beispiel muss also P_A zwingend ausgelagert werden, da T_1 einen commit ausführt, aber P_A darf nicht ausgelagert werden, da sie von der noch laufenden Transaktion T_2 verändert wurde, was einen Widerspruch darstellt.

Hausaufgabe 2

Warum ist es für die Erzielung der Idempotenz der Redo-Phase notwendig, die – und nur die – LSN einer tatsächlich durchgeführten Redo-Operation in der betreffenden Seite zu vermerken? Zeigen Sie für die folgenden Szenarien anhand von Beispielen mit logischer Protokollierung, dass die Idempotenz nicht sichergestellt werden kann.

a) LSN-Eintäge werden in der Redo-Phase nicht auf Datenseiten geschrieben.

b) LSN-Einträge von Log-Records, für die die Redo-Operation nicht ausgeführt wird, werden trotzdem in die Datenseiten übertragen.

Beantworten Sie außerdem folgende Frage:

c) Wie wird die Idempotenz der Undo-Phase sichergestellt, wenn ein Kompensationseintrag geschrieben wurde und dann noch vor der Ausführung des Undo das Datenbanksystem abstürzt?

Lösung:

a) Als Beispiel seien folgende Logeinträge und die Seite P_1 gegeben:

$$[\#2, T_1, P_1, A \mathrel{+}= 1, A \mathrel{-}= 1, \#1]$$
$$[\#3, T_1, P_1, B \mathrel{+}= 2, B \mathrel{-}= 2, \#2]$$
$$[\#4, T_1, \textbf{commit}, \#3]$$

#1	A=10 B=20

Nun stürzt die Datenbank ab nachdem die Daten von Logeintrag #2 auf die Seite P_1 geschrieben wurden. Diese Seite sieht nach dem Absturz folgendermaßen aus:

#2	A=11 B=20

Beim Wiederanlauf muss die Transaktion T_1 wiederhergestellt werden, da ihr commit im Log enthalten ist und sie somit eine Winner-Transaktion ist. Dazu werden die Logeinträge wieder von vorne bearbeitet. Für den ersten Logeintrag wird an der Seite nichts geändert, da die LSN der Seite (#2) nicht kleiner ist als die des Logeintrags (#2). Der zweite Logeintrag hingegen muss bearbeitet werden. Dazu wird die Redo-Operation ausgeführt. Im Szenario dieser Aufgabe wird verlangt, dass in der Redo-Phase keine LSNs verändert werden. Also wird die Redo-Operation des zweiten Logeintrags ausgeführt (also $B \mathrel{+}= 2$) ohne die LSN zu ändern. Danach sieht die Seite folgendermaßen aus:

#2	A=11 B=22

Wenn die Datenbank nun wieder abstürzt, kommt es zu einer Verletzung der Idempotenz. Wie auch beim ersten Absturz, muss die Winner-Transaktion T_1 wiederhergestellt werden. Dazu werden die Logeinträge wieder nacheinander abgearbeitet wobei der erste übersprungen werden kann. Die LSN des zweiten Logeintrags (#3) ist aber größer als die, die auf der Seite gespeichert ist (#2), weswegen die Redo-Operation $B \mathrel{+}= 2$ wieder ausgeführt wird. Danach enthält die Seite folgende Daten:

#2	A=11 B=24

Man kann sehen, dass die Redo-Operation des zweiten Logeintrags fälschlicherweise mehrmals ausgeführt wurde, wenn die LSN-Einträge in der Redo-Phase nicht auf die Seite geschrieben werden.

b) Als Beispiel seien folgende Logeinträge (identisch zur letzten Teilaufgabe) und die Seite P_1 gegeben:

$$[\#2, T_1, P_1, A \mathrel{+}= 1, A \mathrel{-}= 1, \#1]$$
$$[\#3, T_1, P_1, B \mathrel{+}= 2, B \mathrel{-}= 2, \#2]$$
$$[\#4, T_1, \textbf{commit}, \#3]$$

#3	A=11 B=22

Nun stürzt die Datenbank ab. Die Transaktion T_1 muss wiederhergestellt werden, da sie eine Winner-Transaktion ist. Dazu werden die Logeinträge von vorne abgearbeitet. Die Redo-Operation des ersten Logeintrags muss nicht ausgeführt werden, da die LSN auf der Seite (#3) größer ist als die des Logeintrags (#2). In dem Szenario dieser Teilaufgabe ist vorgegeben, dass die LSN für nicht ausgeführte Redo-Operationen aber trotzdem auf die Seite geschrieben werden. Dementsprechend sieht die Seite nach der Bearbeitung des ersten Logeintrags folgendermaßen aus:

#2	A=11 B=22

Wenn die Datenbank jetzt wieder abstürzt, wird die Idempotenz verletzt. Wenn die Logeinträge durchlaufen werden, kann der erste Eintrag übersprungen werden. Beim zweiten Logeintrag ist allerdings die LSN (#3) größer als die LSN auf der Seite (#2) weswegen die Redo-Operation ausgeführt werden muss. Dies folgt zu folgendem Inhalt der Seite:

#3	A=11 B=24

Man kann sehen, dass die Redo-Operation des zweiten Logeintrags wieder fälschlicherweise doppelt ausgeführt wurde, da während des ersten Redos die LSN einer übersprungenen Redo-Operation auf die Seite geschrieben wurde.

Anhand der beiden Szenarien kann man sehen, dass nur die LSN einer tatsächlich durchgeführten Redo-Operation auf die Seite geschrieben werden darf, damit die Idempotenz sichergestellt werden kann.

c) Wie alle anderen Logeinträge auch erhalten CLRs eine LSN. Diese kann bei einem Wiederanlauf mit der LSN auf einer Seite verglichen werden, um zu entscheiden, ob die Undo-Operation, die durch den CLR repräsentiert wird, ausgeführt werden muss. Werden die LSNs für CLRs nur dann auf die Seite geschrieben, wenn das Undo ausgeführt wurde, ist somit auch die Idempotenz der Undo-Phase garantiert.

Hausaufgabe 3

Sie verwenden ein Datenbanksystem mit Write-Ahead-Logging und der Strategie $\neg$*force* und *steal*. Die Datenbank verwaltet lediglich zwei Datenobjekte, X mit dem Anfangswert 10 und Y mit dem Anfangswert 100.

Sie starten die 3 Transaktionen T_1, T_2 und T_3 zum gleichen Zeitpunkt:

T_1	T_2	T_3
BOT	**BOT**	**BOT**
$r(X, x_1)$	$r(Y, y_2)$	$r(X, x_3)$
$x_1 := x_1 + 1$	$r(X, x_2)$	$x_3 := x_3 \cdot 10$
$w(X, x_1)$	$y_2 := y_2 \cdot 2$	$w(X, x_3)$
COMMIT	$x_2 := x_2 + 5$	**COMMIT**
	$w(Y, y_2)$	
	$w(X, x_2)$	
	COMMIT	

Während der Ausführung stürzt Ihre Datenbank ab. Sie wissen nicht, ob - und wenn ja, welche - Transaktionen festgeschrieben wurden. Sie wissen nur, dass die Datenbank ausschließlich *serielle Historien* erzeugt, also dass Transaktionen immer atomar ausgeführt werden und somit keine Verzahnung möglich ist. Bevor Sie die Datenbank neu starten, durchsuchen Sie die Festplatte und stellen fest, dass Y dort den Wert 200 hat. Nachdem die Datenbank neu gestartet wurde und der Recovery-Prozess abgeschlossen ist, liefert sie für X den Wert 110.

Sie wollen nun dem Fehler auf den Grund gehen:

a) Finden Sie zunächst anhand der Zwischenwerte für X und Y heraus, welche Transaktionen *winner* sind, und welche *loser*.

b) Geben Sie das Log an, wie es zum Zeitpunkt des Absturzes auf der Platte stand (verwenden Sie logische Protokollierung).

c) Geben Sie das Log nach Beendigung des Recovery-Prozesses an.

Lösung:

a) Zum Zeitpunkt des Absturzes hatte Y auf der Festplatte den Wert 200. Da Y zu Beginn den Wert 100 hatte, muss Transaktion T_2 zu diesem Zeipunkt schon gestartet und mindestens bis zur Aktion $w(Y, y_2)$ ausgeführt worden sein.

Betrachten wir nun den Wert von X nach Abschluss der Recovery. Aus $X = 110$ folgt, dass zuerst T_1 (X hat nun den Zwischenwert 11) und dann T_3 (X hat nun den Endwert 110) ausgeführt wurde, T_2 aber nicht ausgeführt wurde! T_2 muss also eine Losertransaktion sein, welche während des Recovery-Prozesses wieder rückgängig gemacht wurde.

Das Datenbanksystem hat die Transaktionen also in der logischen Reihenfolge T_1, T_3, T_2 ausgeführt, wobei T_1 und T_3 *winner* sind und T_2 *loser* ist.

b) Hier ein mögliches Log. Bitte beachten: auf die Festplatte wird nur das Log selbst (also die „Log“-Spalte) geschrieben.

Schritt	T_1	T_2	T_3	Log
1.	**BOT**			$[\#1, T_1, \textbf{BOT}, 0]$
2.	$r(X, x_1)$			
3.	$x_1 := x_1 + 1$			
4.	$w(X, x_1)$			$[\#2, T_1, P_X, X \mathrel{+}= 1, X \mathrel{-}= 1, \#1]$
5.	**commit**			$[\#3, T_1, \textbf{commit}, \#2]$
6.			**BOT**	$[\#4, T_3, \textbf{BOT}, 0]$
7.			$r(X, x_3)$	
8.			$x_3 := x_3 \cdot 10$	
9.			$w(x, x_3)$	$[\#5, T_3, P_X, X \mathrel{\cdot}= 10, X \mathrel{/}= 10, \#4]$
10.			**commit**	$[\#6, T_3, \textbf{commit}, \#5]$
11.		**BOT**		$[\#7, T_2, \textbf{BOT}, 0]$
12.		$r(Y, y_2)$		
13.		$r(X, x_2)$		
14.		$y_2 := y_2 \cdot 2$		
15.		$x_2 := x_2 + 5$		
16.		$w(Y, y_2)$		$[\#8, T_2, P_Y, Y \mathrel{\cdot}= 2, Y \mathrel{/}= 2, \#7]$
17.		$w(X, x_2)$		$[\#9, T_2, P_X, X \mathrel{+}= 5, X \mathrel{-}= 5, \#8]$

Ob Schritt 17 tatsächlich ausgeführt wurde, können wir nicht wissen. Der Vollständigkeit zuliebe gehen wir hier vom *worst case* aus. Die Datenbank stürzt direkt vor dem *commit* von T_2 ab.

c) Da T_2 eine *loser*-Transaktion ist, wird die Datenbank während der Recovery in der *Undo-Phase Compensation Log Records* schreiben, um die partiell durchgeführte Transaktion rückgängig zu machen. Das endgültige Log sieht dann wie folgt aus:

$$[\#1, T_1, \textbf{BOT}, 0]$$
$$[\#2, T_1, P_X, X \mathrel{+}= 1, X \mathrel{-}= 1, \#1]$$
$$[\#3, T_1, \textbf{commit}, \#2]$$
$$[\#4, T_3, \textbf{BOT}, 0]$$
$$[\#5, T_3, P_X, X \mathrel{\cdot}= 10, X \mathrel{/}= 10, \#4]$$
$$[\#6, T_3, \textbf{commit}, \#5]$$
$$[\#7, T_2, \textbf{BOT}, 0]$$
$$[\#8, T_2, P_Y, Y \mathrel{\cdot}= 2, Y \mathrel{/}= 2, \#7]$$
$$[\#9, T_2, P_X, X \mathrel{+}= 5, X \mathrel{-}= 5, \#8]$$
$$\langle \#9', T_2, P_X, X \mathrel{-}= 5, \#9, \#8 \rangle$$
$$\langle \#8', T_2, P_Y, Y \mathrel{/}= 2, \#9', \#7 \rangle$$
$$\langle \#7', T_2, -, -, \#8', 0 \rangle$$

Hausaufgabe 4

Wofür stehen die vier Buchstaben ACID? Erklären Sie für jeden der vier Konzepte, warum es für eine Datenbank wichtig ist. Geben Sie dazu jeweils ein Beispiel an, was passieren könnte, wenn dieses Konzept nicht gelten würde.

Lösung:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Wenn eine Datenbank keine Atomarität garantieren kann, kann es zu inkonsistenten Daten kommen (unabhängig von der Konsistenzeigenschaft/Consistency). Hierzu dient eine Überweisung in einer Bank als Beispiel: Wenn eine Transaktion daraus besteht, einen Kontostand zu verringern und einen andern zu erhöhen, entsteht ein inkonsistenter Zustand, wenn nur eine der beiden Operationen tatsächlich gespeichert wird.

Bei einer Datenbank, die nicht konsistent ist, können weitreichende Probleme entstehen. Wenn z.B. garantiert werden muss, dass Kontonummern eindeutig sind, die Datenbank aber inkonsistente Daten zulässt, kann nicht mehr garantiert werden, dass Überweisungen tatsächlich beim richtigen Kontobesitzer ankommen.

Bei dem Beispiel einer Überweisung kann es auch zu Problemen kommen, wenn die Datenbank Transaktionen nicht korrekt voneinander isoliert. Zwei parallele Überweisungen können gleichzeitig Kontostände ändern, was zu inkorrekten Kontoständen nach der Ausführung beider Transaktionen führen kann.

Wenn eine Datenbank eingesetzt wird, die keine Dauerhaftigkeit garantieren kann, kann nie darauf vertraut werden, dass ein commit eine Transaktion tatsächlich festschreibt. Das ist aber z.B. bei einem Geldautomaten notwendig, der erst Geld ausgeben sollte, sobald die Datenbank garantieren kann, dass die Abhebetransaktion verbucht ist.