

1 Striktes und Rigoroses 2PL

Welchen Vorteil bietet rigoroses Zwei-Phasen-Sperren? Wie vergleicht es sich mit anderen Formen des Zwei-Phasen-Sperrens?

Das Zwei-Phasen-Sperrrprotokoll (2PL)

- 1. Phase (Anforderungsphase, growing phase):
 - Transaktionen dürfen Locks anfordern.
 - Transaktionen dürfen keine Locks freigeben.
- 2. Phase (Freigabephase, shrinking phase):
 - Transaktionen dürfen keine Locks anfordern.
 - Transaktionen dürfen bisher erworbene Locks freigeben.

Striktes 2PL

- **Exclusive** Locks werden nicht vor dem Commit freigegeben.
- Verhindert "Dirty Reads"

Rigoroses 2PL:

- **Alle** Locks werden erst nach dem Commit freigegeben.
- Transaktionen können in der Commit-Reihenfolge serialisiert werden.

verhindert Dirty Reads

Strikt

lock_S(A)
read(A)
* unlock(A) ← erlaubt, da strikt
lock_X(B)
write(B)
commit
unlock(B) ← unlock exklusive nach commit

Rigoroses

lock_S(A)
read(A)
lock_X(B)
write(B)
commit
unlock(A)
unlock(B) } alle Locks erst nach commit abgeben

* Andere Transaktionen im Schedule könnten zu dem Zeitpunkt auf die Ressource zugreifen
=> mehr Nebenläufigkeit ← **Nachteil Rigoroses**
=> weniger Nebenläufigkeit

Vorteile Rigoroses

- keine Cascading Rollbacks:
Transaktionen dürfen keine uncommitted Daten lesen
- Nur uncommitted Sachen müssen zurückgerollt werden
- Transaktionen können in Commit-Reihenfolge serialisiert werden

Upgradebares 2PS

- Upgrade: S zu X Downgrade: X zu S
=> mehr Nebenläufigkeit & garantierte Serialisierbarkeit
- Cascading Rollbacks, wenn ein Lock vor commit freigegeben wird
↳ Dirty data

1 Striktes und Rigoroses 2PL

Welchen Vorteil bietet rigoroses Zwei-Phasen-Sperren? Wie vergleicht es sich mit anderen Formen des Zwei-Phasen-Sperrens?

2 Deadlock Detection

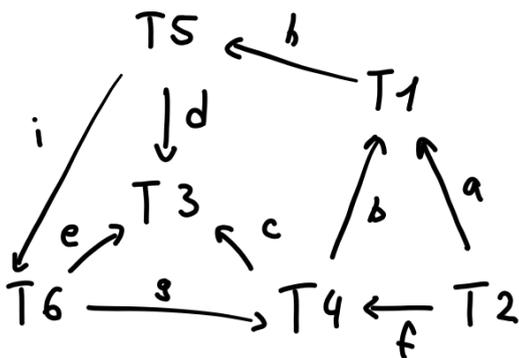
Angenommen, die Transaktionen $T_1 - T_6$ laufen gleichzeitig auf einer Datenbank. Die folgende Tabelle zeigt, welche Datenobjekte a, \dots, j derzeit von den Transaktionen gesperrt sind und auf welche Freigaben von Sperren die Transaktionen warten.

Transaktion	hält Sperre auf	Wartet auf das Freigeben der Sperre auf
→ T_1	a, b	h
T_2		a, f
T_3	c, d, e	
→ T_4	f, g	b, c
→ T_5	h	d, i
→ T_6	i	e, g

T_1, T_5, T_6, T_4, T_1

Bitte erstellen Sie den Wait-For Graph und entscheiden Sie, ob ein Deadlock vorliegt. Im Falle eines Deadlocks, lösen Sie ihn bitte auf.

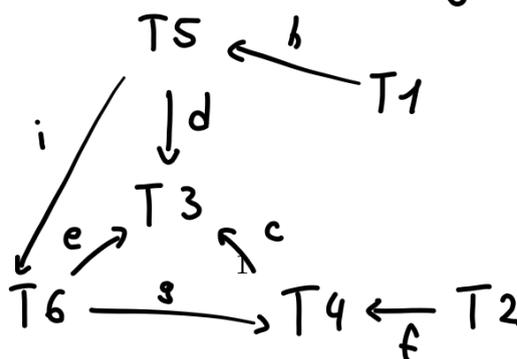
Ein Deadlock existiert, wenn der Wartegraph einen Zyklus hat.



Um den Deadlock aufzulösen muss eine Transaktion abgebrochen werden z.B. T_1 *

=> - Rollback aller Aktionen von T_1
 - Freigabe der Sperren

* Gewähltes Opfer hat max Locks
 => Maximierung der freigegebenen Ressourcen



Erst loggen, dann schreiben.

WAL: > Write Ahead Logging

WAL auf Festplatte → sichert durability auch bei Stromausfall



3 Recovery ! Bei einem Fehler Redo, wenn committed
Undo wenn nicht committed

Drei Transaktionen, T_1, T_2, T_3 , laufen gleichzeitig gemäß der folgenden Tabelle. Die Tabelle zeigt auch die geschriebenen Log-Dateien.

	T_1	T_2	T_3	Log
				[TID, DID, old, new]
1.	BOT			[T_1 start]
2.	$r(A, a_1)$			
3.		BOT		[T_2 start]
4.	$a_1 = a_1 + 4$			
5.		$r(B, b_1)$		
6.		$b_1 = b_1 \cdot 1.1$		
7.		$w(B, b_1)$		[$T_2, B, 1, 1.1$]
				point 1
8.	$w(A, a_1)$			[$T_1, A, 1, 5$]
9.		$r(A, a_2)$		
10.		$a_2 = a_2 + 10$		
11.			BOT	[T_3 start]
12.		$w(A, a_2)$		[$T_2, A, 5, 15$]
13.		commit		[T_2 commit]
14.			$r(B, b_1)$	
15.	$r(C, c_1)$			
16.	$r(B, b_2)$			
17.	$c_1 = c_1 + b_2$			
18.	$w(c_1)$			[$T_1, C, 9, 10.1$]
19.			$b_1 = b_1 \cdot 2$	
20.			$w(B, b_1)$	[$T_3, B, 1.1, 2.2$]
21.	commit			[T_1 commit]
				point 2
22.			commit	[T_3 commit]

$B = 1.1$

$A = 5$

$A = 15$

$C = 10.1$

$B = 1.1$

- Das System stürzt an Punkt 1 ab. Geben Sie an, was in jeder Phase des Wiederanlaufs geschieht.
- Diesmal stürzt das System bei Punkt 1 nicht ab, bei Punkt 2 stürzt es allerdings wieder ab. Geben Sie an, was in jeder Phase des Wiederanlaufs geschieht.

1) Da nicht committed wurde, wird nichts redone. Alles undone
Undo-Set = { T_2 } Redo-Set = {} $T_2 \quad B = 1.1 \Rightarrow B = 1$

2) T_1 wird redone und T_2 T_3 undone, da nicht committed
 $A = 1 \quad C = 9 \quad B = 1 \quad A = 5$
 $A = 5 \quad C = 10.1 \quad B = 1.1 \quad A = 15$
Redo-Set = { T_1, T_2 } $B = 2.2 \Rightarrow B = 1.1$
Undo-Set = { T_3 }

4 Deadlock Prevention vs. Detection

Unter welchen Bedingungen ist es günstiger, (i) Deadlocks zu vermeiden, als (ii) Deadlocks zuzulassen und sie dann zu erkennen?

2PL kann Deadlocks nicht verhindern

i) Konservatives 2PL aber schon (extrem vorsicht)

- > alle Locks vom Anfang an gesetzt
 - > es gibt keine zyklische Wartestände (Deadlocks), da keiner auf die Freigabe von Locks wartet
 - > Eine Transaktion beginnt erst, wenn alle Sperren verfügbar sind -> weniger Parallelität
- Gut, wenn Deadlocks ständig auftreten, da das Erkennen & Rollbacks teuer werden können
 - Gut, wenn Transaktionen sehr lang sind, da das Zurücksetzen teuer wäre
 - Für sicherheitskritische Systeme

ii) Erkennung von Deadlocks mittels Wartegraphen

- Gut, wenn Deadlocks selten auftreten, ohne konservatives 2PL hat man mehr Parallelität => effizienter (Erkennen & Rollbacks treten selten vor)
- Gut, wenn Transaktionen kurz sind, da das Zurücksetzen nicht teuer ist

5 Conflict Graph vs. Wait-For Graph

Beschreiben Sie die Unterschiede und Gemeinsamkeiten zwischen Conflict Graphs und Wait-For Graphs.

Sie können den folgenden Schedules als Grundlage für Ihre Diskussion verwenden:

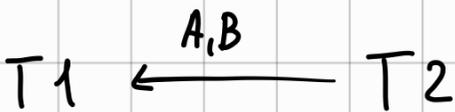
step	T_1	T_2
1	lock_X(A)	
2	lock_S(B)	
3	write(A)	
4	read(B)	
5	unlock(A)	
6	unlock(B)	
7		lock_X(B)
8		lock_S(A)
9		write(B)
10		read(A)
11		unlock(B)
12		unlock(A)
13	commit	
14		commit

Wait-For Graph

zeigt welche Transaktionen auf welche andere Transaktionen warten für die Freigabe von Ressourcen

Ziel: Deadlockerkennung
Zyklus \Rightarrow Deadlock

Richtung sagt, wer von wem abhängt

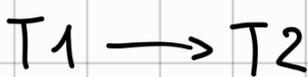


Konfliktgraphen

Setzt Kante zw. zwei Transaktionen, falls beide eine Operation auf demselben Objekt ausführen (r/w, u/r, w/w)

Sidenote: Konflikte entstehen zw. Operationen, nicht ganze Transaktionen

Ziel: Prüfen Konfliktserialisierbarkeit eines Schedules
Zyklus \Rightarrow nicht serialisierbar



Richtung sagt, wo als erstes die konfliktbehaftete Operation auftritt

Gemeinsames

Knoten: Transaktionen
Zyklus: Erkennung

Unterschiede

Kante KG: Konflikt
Kante WG: Warten auf Freigabe von Ressourcen

Zyklus KG: nicht serialisierbar

Zyklus WG: Deadlock

WG zur Laufzeit
 \rightarrow Ziel: Deadlocks fixen

CG nach Ausführung
da man die ganze Info des Schedules braucht, was der Scheduler vor der Ausführung nicht bestimmt hat

4 Deadlock Prevention vs. Detection

Unter welchen Bedingungen ist es günstiger, (i) Deadlocks zu vermeiden, als (ii) Deadlocks zuzulassen und sie dann zu erkennen?

5 Conflict Graph vs. Wait-For Graph

Beschreiben Sie die Unterschiede und Gemeinsamkeiten zwischen Conflict Graphs und Wait-For Graphs.

Sie können den folgenden Schedules als Grundlage für Ihre Diskussion verwenden:

step	T_1	T_2
1	lock_X(A)	
2	lock_S(B)	
3	write(A)	
4	read(B)	
5	unlock(A)	
6	unlock(B)	
7		lock_X(B)
8		lock_S(A)
9		write(B)
10		read(A)
11		unlock(B)
12		unlock(A)
13	commit	
14		commit