

Enforcing Serializability

יש שיטות פסימיסטיות:

1. 2PL - Two phase locking

שמניחות שהסריאליזציה הולכת להשתבש, ובהתאם לכך לא מאפשרת דברים מסויימים. מצד שני, יש שיטות אופטימיסטיות:

2. TS - Timestamps

3. Validation

שמאפשרות יותר דברים - ואם משהו משתבש אז מתקנים.

2PL - Two Phase Locking (1)

נסמן:

- נעילה (lock) - l
- שחרור (unlock) - u
- פעולה (action) - A

הכללים הם:

1. Consistency of T_i - כל טרנזקציה נועלת, מבצעת פעולה, ואז משחררת:

$$l_i(X), A_i(X), u_i(X)$$

2. Legality of S - לפני שטרנזקציה נועלת, הטרנזקציה הקודמת חייבת לשחרר את הנעילה שלה:

$$l_i(X), u_i(X), l_j(X)$$

דוגמה

T_1	T_2	A	B
		25	25
$l_1(A), r_1(A)$ $A = A + 100$ $w_1(A), u_1(A)$		125	
	$l_2(A), r_2(A)$ $A = A * 2$ $w_2(A), u_2(A)$	250	
	$l_2(B), r_2(B)$ $B = B * 2$ $w_2(B), u_2(B)$		50
$l_1(B), r_1(B)$ $B = B + 100$ $w_1(B), u_1(B)$			150

למרות שנעלנו, הסריאליזציה לא נכונה!
הבעיה היא שלא התייחסנו לחלק של Two-Phasen - הנעילה והשחרור צריכים להתבצע
בשני שלבים - נועלים הכל(שלב ראשון), מבצעים את הפעולות, ואז משחררים הכל(שלב שני).
נבצע את הסריאליזציה בשני שלבים:

T_1	T_2	A	B
		25	25
$l_1(A), r_1(A)$ $A = A + 100$ $w_1(A), l_1(B)$ $u_1(A)$		125	
	$l_2(A), r_2(A)$ $A = A * 2$ $w_2(A)$ $l_2(B)$ -Denied!	250	
$r_1(B), B = B + 100$ $w_1(B), u_1(B)$			125
	$l_2(B), w_2(A), r_2(B)$ $B = B * 2, w_2(B), u_2(B)$		250

דוגמה - בעיה!

$T_1 : l_1(A), l_2(B), u_1(A), u_1(B)$ 2-phase

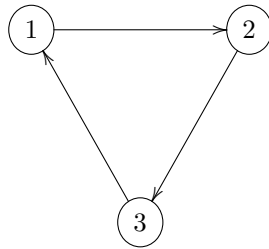
$T_2 : l_2(B), l_2(A), u_2(B), u_2(A)$ 2-phase

T_1	T_2
$l_1(A)$	$l_2(B)$
$l_1(B)$ -Denied!	$l_2(A)$ -Denied

Deadlock!

Deadlock

נוצר כאשר יש כמה טרנזקציות, שכל אחת מחכה לאחרת שתשתחרר נתון, ונועלת בעצמה נתון אחר, ונוצר מעגל:



(Starvation)Livelock

נוצר כאשר טרנזקציה T_1 נועלת את A , ו T_2 מחכה ל A . בזמן ש T_2 מחכה, T_3 מצטרפת ומבקשת את A . ברגע ש T_1 משחררת את A , T_3 מקבלת אותו במקום T_2 , וכן הלאה - עוד ועוד טרנזקציות גונבות ל T_2 את A .

פתרון לLivelock

1. במקום לבחור באופן שרירותי איזו טרנזקציה תהיה הבאה לקבל את הנתון, נבנה תור FIFO.
2. עדיפות - נשתמש בתור עדיפויות, וככל שעובר הזמן, העדיפות של טרנזקציה תעלה.

פתרון לDeadlock

נבצע תמיד את הנעילה בסדר קבוע - למשל לפי הא"ב.

דוגמה

$T_1 : l_1(A), r_1(A), l_1(B), w_1(B), u_1(A), u_1(B)$
 $T_2 : l_2(A), l_2(C), r_2(C), w_2(A), u_2(C), u_2(A)$
 $T_3 : l_3(B), r_3(B), l_3(C), w_3(C), u_3(B), u_3(C)$
 $T_4 : l_4(A), l_4(D), r_4(D), w_4(A), u_4(D), u_4(A)$

T_1	T_2	T_3	T_4
$l_1(A), r_1(A)$	$l_2(A)$ -Denied!	$l_3(B), r_3(B)$	$l_4(A)$ -Denied!
		$l_3(C), w_3(C)$ $u_3(C), u_3(D)$	
$l_1(B), w_1(B)$ $u_1(A)$			
	$l_2(A), l_2(C)$ $r_2(C), w_2(A)$ $u_2(C), u_2(A)$		
$u_1(B)$			$l_4(A), \dots u_4(A)$

אלגוריתם לבדיקה אם יש Deadlock

DG - directed graph

nodes: T

edges: $T_i \rightarrow T_j$

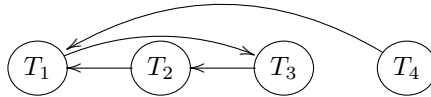
DAG \Rightarrow no deadlock

loop \Rightarrow deadlock

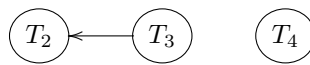
דוגמה

T_1	T_2	T_3	T_4
$l_1(A), r_1(A)$	$l_2(C), r_2(C)$	$l_3(B), r_3(B)$	$l_4(D), r_4(D)$
	$l_2(A)$ -Denied!	$l_3(C)$ -Denied!	$l_4(A)$ -Denied

כדי לבדוק אם יש Deadlock, נצייר גרף:



כדי לצאת מהDeadlock, נוציא את T_1 :



ואין מעגלים. חשוב לזכור שלהוציא טרנזקציה זה לא פעולה טריוויאלית, כי צריך לבטל את כל מה שהיא עשתה.

Locking Systems with Several Lock Modes

Shared(read)/eXclusive(write) Lock (1)

אם לטרנזקציה יש Shared Lock (sl), אז גם טרנזקציות אחרות יכולות לקבל Shared Lock. אבל רק טרנזקציה אחת יכולה להחזיק בExclusive Lock (xl) בכל רגע נתון - ובתנאי שאין Shared Lock על הנתון.

דוגמה

T_1 T_2

$sl_1(A), r_1(A)$

$sl_2(A), r_2(A)$

$sl_2(B), r_2(B)$

$xl_1(B)$ -Denied!

$u_2(A), u_2(B)$

$xl_1(B), r_1(B), w_1(B)$

$u_1(A), u_1(B)$

Upgrading Locks

במקום להשאיר את הנתון נעול הרבה זמן, קודם מבקשים Shared Lock, ורק כשרוצים לכתוב מבקשים Exclusive Lock.
יכול להיות שהטרנזקציה תצטרך לחכות לExclusive Lock (כי טרנזקציות אחרות מחזיקות Shared Lock) - אבל זה עדיף מאשר לבקש מוקדם, לחכות מוקדם, ולעכב את כולם.

דוגמה

T_1	T_2
$sl_1(A), r_1(A)$	$sl_2(A), r_2(A)$ $sl_2(B), r_2(B)$
$sl_1(B), r_1(B)$ $xl_1(B)$ -Denied!	$u_2(A), u_2(B)$
$xl_1(B), w_1(B)$ $u_1(A), u_1(B)$	

בעיית Upgrade בגלל Deadlock

T_1	T_2
$sl_1(A)$	$sl_2(A)$
$xl_1(A)$ -Denied!	$xl_2(A)$ -Denied

וקיבלנו Deadlock - על אותו נתון!
לכן נשתמש בסוג חדש של lock:

(ul)Update Lock

Update Lock יכול להיות במקביל ל Shared Lock - אבל לא במקביל ל Exclusive Lock.
אחר: וכמובן שלא במקביל ל Exclusive Lock.

Increment Lock

מכיוון שאפשר לבצע מספר פעולות חיבור ללא תלות בסדר, נגדיר פעולה "אטומית" להוספה:

$$inc_i(X, c) = r_i(X), X = X + c, w_i(X)$$

כמו כן נגדיר Increment Lock - $il_i(X)$. לא ניתן לקבל il כאשר יש sl או cl - אבל אפשר לקבל כאשר יש il אחר:

	sl	xl	il
sl	y	n	n
xl	n	n	n
il	n	n	y

דוגמה

T_1	T_2
$sl_1(A), r_1(A)$	$sl_2(A), r_2(A)$
$il_1(B), inc_1(B)$	$il_2(B), inc_2(B)$
$u_1(A), u_1(B)$	$u_2(A), u_2(B)$

Timestamping

יש אלגוריתם שעובד לפי השוואות של מספרים. לכל טרנזקציה יש מסדר שנקרא Ts - timestamp. זה נקבע לפי השעון של המחשב או לפי מספור פנימי - מה שחשוב לנו זה שהמספר יהיה קבוע. כמו כן לכל נתון יהיה RT - Read Time ו- WT - Write Time, שאומרים מתי הנתון נקרא ונכתב לאחרונה.

דוגמה

T_1	T_2	T_3	$A - RT$	$A - WT$
100	200	150	0	0
$r_1(A)$	$r_2(A)$	$r_3(A)$ $w_3(A)$	100	150
$w_1(A)$			200	
	$w_2(A)$			200

הדוגמה הזו מראה איך המספרים משתנים - היא לא מראה את האלגוריתם.

האלגוריתם

```

switch(request)
{
  case  $r_T(X)$ :
    if( $TS(T) \geq WT(X)$ )
    {
       $r_T(X)$ ;
      if( $TS(T) > RT(X)$ )
         $RT(X) = TS(T)$ ;
    }
    else rollback  $T$ ;
  case  $w_T(X)$ :
    if( $TS(T) \geq RT(X)$ )
    {
      if( $TS(T) \geq WT(X)$ )
      {
        write  $X$ ;
         $WT(X) = TS(T)$ ;
      }
    }
    else{}
}

```



```

else rollback T;
}

```

.....[צריך להשלים]

DB Recovery

בעיות אפשריות:

- במקרה שנקלענו ל-Deadlock, ונאלצנו להוציא טרנזקציה, אז צריך להחזיר את ה-DB למצב שלפני הטרנזקציה.
- אותו דבר במקרה של Timestamping
- וגם במקרה של חישוב לא חוקי - למשל חילוק באפס.

דוגמה

A, B נתונים ב-DB. t משתנה לוקלי לצורכי חישובים

$$A = 8 \quad B = 8$$

Action	t	Mem A	Mem B	Disk A	Disk B
$r(A, t)$	8	8		8	8
$t = t * 2$	16	8		8	8
$w(A, t)$	16	16		8	8
$r(B, t)$	8	16	8	8	8
$t = t * 2$	16	16	8	8	8
$w(B, t)$	16	16	16	8	8
output (A)	16	16	16	16	8
output (B)	16	16	16	16	16

מה קורה אם צריך לבטל את זה?
נתבונן בכמה מנגנונים

Undo Recovery

כדי להתאושש, אנו צריכים גיבוי(copy) של בסיס הנתונים, log שבו רשומים הפעולות שביצענו.

- כשטרנזקציה מתחילה, נכתב בלוג $\langle \text{start } T \rangle$

- כשהטרנזציה מגיעה לנקודה שאחרי כל החישובים, ממש לפני שהשינויים נכתבים, נכתב בלוג $\langle \text{commit } T \rangle$.
- commit היא פקודה בתוך התוכנית, ובאחריות המתכנת לכתוב אותה.
- במידה והטרנזקציה נכשלת, נכתב בלוג $\langle \text{abort } T \rangle$.
- בכל כתיבה, נכתב בלוג $\langle T, X, v \rangle$, כאשר T - הטרנזקציה, X - הנתון שנכתב, ו- v - הערך הישן.

כדי לעבוד עם שיטת Undo, צריכים לשמור על כללים:

(1) סדר כתיבה

1. מכינים את הכתיבה ללוג
2. כותבים ל-DB
3. מבצעים את הכתיבה ללוג

דוגמה

	Action	t	Mem A	Mem B	Disk A	Disk B	LOG(UNDOO)
1							
2	$r(A, t)$	8	8		8	8	$\langle \text{start } T \rangle$
3	$t = t * 2$	16	8		8	8	
4	$w(A, t)$	16	16		8	8	$\langle T, A, 8 \rangle$
5	$r(B, t)$	8	16	8	8	8	
6	$t = t * 2$	16	16	8	8	8	
7	$w(B, t)$	16	16	16	8	8	$\langle T, B, 8 \rangle$
8	flush log						
9	output (A)	16	16	16	16	8	
10	output (B)	16	16	16	16	16	
12	flush log						$\langle \text{commit} \rangle$

אלגוריתם UNDOO

- 1)↓ committed T
active T
- 2)↑ incomplete - UNDO
- 3) incomplete \Rightarrow abort
- 4) flush log

- את 1 מבצעים במידה ונכשלים אחרי 12 - במידה וכבר ביצענו את הטרוזקציה אז לא צריך recovery
- את 2 מבצעים במידה ונכשלים ב10-11 - מבצעים undo - מחזירים את A, B לערכים המקוריים שלהם.
- את 3 מבצעים במידה ונכשלים ב8-10 - מבצעים כמו 2
- את 4 מבצעים במידה ונכשלים ב1-8 - מבצעים כמו 2

REDO

כאן הסדר של הכתיבה בלוג הוא

1. start
2. commit
3. abort
4. $\langle T, X, v \rangle$ - כאשר הפעם v הוא הערך החדש

דוגמה

T	t	M.A	M.B	D.A	D.B	Log(REDO)
1						$\langle \text{start } T \rangle$
2	$r(A, t)$	8	8	8	8	
3	$t = t * 2$	16				
4	$w(A, t)$		16			$\langle T, A, 16 \rangle$
5	$r(B, t)$	8	8			
6	$t = t * 2$	16				
7	$w(B, t)$		16			$\langle T, B, 16 \rangle$
8						$\langle \text{commit} \rangle$
9	flush log					
10	output (A)			16		
11	output (B)				16	

מה צריך לעשות

- 1) ↓ log: commit action
- 2) ↓ imcomplete - nothing
committed - REDO

- 3) incomplete \Rightarrow abort
- 4) flush log

Dirty Data

T_1	T_2	T_3
\vdots		
$w_1(A)$		
\vdots	$r_2(A)$	
\vdots	\vdots	
\vdots	$w_2(A)$	
\vdots	commit	
\vdots	\vdots	$r_3(A)$
abort	\vdots	\vdots

T_1 עשתה abort לפני שהיא סיימה - אבל אחרי שהיא כתבה את A . זה גורם לכך ש T_2 ו T_3 כבר השתמשו ב A - בערך לא נכון שלא!
 המצב הזה נקרא Dirty Data, וצריך לעשות cascading rollback.
 צריך לרשום check point, כדי שיהיה אפשר לעשות את זה.