COMP207 Tutorial Exercises Solutions
Week 5 ($2^{nd}/4^{th}$ November)

The exercises below provide the opportunity to practice the concepts and methods discussed during previous week's videos/reading material. If you haven't done so, it is worthwhile to spend some time on making yourself familiar with these concepts and methods. Don't worry if you cannot solve all the exercises during the tutorial session, but try to tackle at least one or two of them. If at some point you do not know how to proceed, you could review the relevant material from the videos/reading material and return to the exercise later.

Recoverable, Cascadeless, and Strict Schedules

Exercise 1 (Exercise 20.24 in [1])**.** For each of the following schedules, determine if the schedule is (A) recoverable, (B) cascadeless, (C) strict, (D) non-recoverable. Try to determine the strictest recoverability condition that each schedule satisfies.

(a) $S_1: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); w_2(Y); c_2$

(b) $S_2: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y); c_1; c_2; c_3$

(c) $S_3: r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1; w_2(Z); w_3(Y); w_2(Y); c_3; c_2$

Solutions:

Schedule	Recoverable?	Cascadeless?	Strict?	Strictest Condition
S_1	yes	yes	yes	strict
S_2	no	no	no	non-recoverable
S_3	yes	yes	no	cascadeless

Exercise 2 (Exercise 19.1.1 in [2])**.** What are all the ways to insert lock operations (of the simple lock type only), unlock operations, and commit operations into

$$r_1(X); r_1(Y); w_1(X); w_1(Y)$$

so that the transaction T_1 is:

(a) Two-phase locked, and strict two-phase locked.

(b) Two phase locked, but not strict two-phase locked.

Solutions:

(a) The locking operation for X must occur before $r_1(X)$, the commit operation must come immediately after $w_1(Y)$, followed by two unlock operations for X and Y (which can come in any order). The locking operation for Y can be inserted anywhere before $r_1(Y)$. This leads to the six combinations:

- $l_1(Y);\ l_1(X);\ r_1(X);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(X);\ u_1(Y)$
- $l_1(X);\ l_1(Y);\ r_1(X);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(X);\ u_1(Y)$
- $l_1(X);\ r_1(X);\ l_1(Y);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(X);\ u_1(Y)$
- $l_1(Y);\ l_1(X);\ r_1(X);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(Y);\ u_1(X)$
- $l_1(X);\ l_1(Y);\ r_1(X);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(Y);\ u_1(X)$
- $l_1(X);\ r_1(X);\ l_1(Y);\ r_1(Y);\ w_1(X);\ w_1(Y);\ c_1;\ u_1(Y);\ u_1(X)$

(b) As in (a), the locking operation for X must occur before $r_1(X)$, and the locking operation for Y can be inserted anywhere before $r_1(Y)$. The unlocking operation for X must come after $w_1(X)$, and the unlocking operation for Y must come after $w_1(Y)$. To be not strict 2PL, one of the unlocking operations must come before the commit operation.

Timestamp-based deadlock detection

As pointed out in the video "No cascading-rollbacks!", deadlocks may arise even if transactions are scheduled using strict two-phase locking (strict 2PL). The video "Detecting deadlocks" covered several techniques for deadlock detection, among them two techniques based on timestamps: *wait-die* and *wound-wait*.

Exercise 3. Consider the following schedules:

- $S_1: xl_1(X);\ r_1(X);\ sl_2(Y);\ r_2(Y);\ xl_2(X);\ w_2(X);\ u_2(X);\ u_2(Y);\ w_1(X);\ u_1(X)$
- $S_2: sl_1(X);\ r_1(X);\ xl_2(Y);\ r_2(Y);\ xl_1(Y);\ r_1(Y);\ w_2(Y);\ u_2(Y);\ w_1(Y);\ u_1(X);\ u_1(Y)$

For each of the schedules, decide if a lock request is denied, and if so give the first lock request that is denied and say what happens in this case under the

(a) wait-die scheme;

(b) wound-wait scheme.

Assume that T_1 arrives earlier than T_2.

Solutions

- S_1: The request for an exclusive lock on X for T_2 is denied, because T_1 holds an exclusive lock on X ("T_2 waits for T_1 to unlock X").
  - **(a)** Under the wait-die scheme, T_2 will be aborted and restarted, because T_2 is younger than T_1.
  - **(b)** Under the wound-wait scheme, T_2 is allowed to wait.
- S_2: The request for an exclusive lock on Y for T_1 is denied, because T_2 holds an exclusive lock on Y ("T_1 waits for T_2 to unlock Y").
  - **(a)** Under the wait-die scheme, T_1 is allowed to wait further, because it is older than T_2.
  - **(b)** Under the wound-wait scheme, T_2 is aborted and restarted.

Timestamp-based scheduling

Exercise 4 (Exercise 18.8.1 in [2]). Below are several sequences of start events and read/write operations (here, st_i means that transaction T_i starts):

(a) $st_1;\ st_2;\ r_1(X);\ r_2(Y);\ w_2(X);\ w_1(Y)$

(b) $st_1;\ r_1(X);\ st_2;\ w_2(Y);\ r_2(X);\ w_1(Y)$

(c) $st_1;\ st_2;\ st_3;\ r_1(X);\ r_2(Y);\ w_1(Z);\ r_3(Y);\ r_3(Z);\ w_2(Y);\ w_3(X)$

(d) $st_1;\ st_3;\ st_2;\ r_1(X);\ r_2(Y);\ w_1(Z);\ r_3(Y);\ r_3(Z);\ w_2(Y);\ w_3(X)$

Tell what happens as each of the sequences executes under a (basic) timestamp-based scheduler. Assume that the read and write times of all items are 0 at the beginning of the sequence.

Solutions

(a) The following table shows what happens when the sequence is executed under a timestamp-based scheduler:

Time	Operation	RT(X)	WT(X)	RT(Y)	WT(Y)	Other Action
0		0	0	0	0	
1	st_1	0	0	0	0	new timestamp for T_1: $\mathrm{TS}(T_1) = t_1$
2	st_2	0	0	0	0	new timestamp for T_2: $\mathrm{TS}(T_2) = t_2 > t_1$
3	$r_1(X)$	t_1	0	0	0	granted
4	$r_2(Y)$	t_1	0	t_2	0	granted
5	$w_2(X)$	t_1	t_2	t_2	0	granted
6	$w_1(Y)$					T_1 aborts

(b) Execution of the sequence under a timestamp-based scheduler:

Time	Operation	RT(X)	WT(X)	RT(Y)	WT(Y)	Other Action
0		0	0	0	0	
1	st_1	0	0	0	0	new timestamp for T_1: $\mathrm{TS}(T_1) = t_1$
2	$r_1(X)$	t_1	0	0	0	granted
3	st_2	t_1	0	0	0	new timestamp for T_2: $\mathrm{TS}(T_2) = t_2 > t_1$
4	$w_2(Y)$	t_1	0	0	t_2	granted
5	$r_2(X)$	t_2	0	0	t_2	granted
6	$w_1(Y)$					T_1 aborts

(c) Execution of the sequence under a timestamp-based scheduler:

		X		Y		Z		
Time	**Operation**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**Other Action**
0		0	0	0	0	0	0	
1	st_1	0	0	0	0	0	0	new timestamp for T_1: $\text{TS}(T_1) = t_1$
2	st_2	0	0	0	0	0	0	new timestamp for T_2: $\text{TS}(T_2) = t_2 > t_1$
3	st_3	0	0	0	0	0	0	new timestamp for T_3: $\text{TS}(T_3) = t_3 > t_2$
4	$r_1(X)$	t_1	0	0	0	0	0	granted
5	$r_2(Y)$	t_1	0	t_2	0	0	0	granted
6	$w_1(Z)$	t_1	0	t_2	0	0	t_1	granted
7	$r_3(Y)$	t_1	0	t_3	0	0	t_1	granted
8	$r_3(Z)$	t_1	0	t_3	0	t_3	t_1	granted
9	$w_2(Y)$							T_2 aborts, because $\text{RT}(Y) = t_3 > t_2 = \text{TS}(T_2)$
10	$w_3(X)$							

(d) Execution of the sequence under a timestamp-based scheduler:

		X		Y		Z		
Time	**Operation**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**Other Action**
0		0	0	0	0	0	0	
1	st_1	0	0	0	0	0	0	new timestamp for T_1: $\text{TS}(T_1) = t_1$
2	st_3	0	0	0	0	0	0	new timestamp for T_3: $\text{TS}(T_3) = t_3 > t_1$
3	st_2	0	0	0	0	0	0	new timestamp for T_2: $\text{TS}(T_2) = t_2 > t_3$
4	$r_1(X)$	t_1	0	0	0	0	0	granted
5	$r_2(Y)$	t_1	0	t_2	0	0	0	granted
6	$w_1(Z)$	t_1	0	t_2	0	0	t_1	granted
7	$r_3(Y)$	t_1	0	t_2	0	0	t_1	granted
8	$r_3(Z)$	t_1	0	t_2	0	t_3	t_1	granted
9	$w_2(Y)$	t_1	0	t_2	t_2	t_3	t_1	granted
10	$w_3(X)$	t_1	t_3	t_2	t_2	t_3	t_1	granted

Exercise 5 (Exercise 18.8.2 in [2])**.** Tell what happens during the following sequences of events if a multiversion, timestamp scheduler is used. What happens instead, if the scheduler does not maintain multiple versions?

(a) $st_1; st_2; st_3; st_4; w_1(A); w_2(A); w_3(A); r_2(A); r_4(A);$

(b) $st_1; st_2; st_3; st_4; w_1(A); w_3(A); r_4(A); r_2(A);$

(c) $st_1; st_2; st_3; st_4; w_1(A); w_4(A); r_3(A); w_2(A);$

Solutions

(a) MVCC version: The three writes create three versions of A, each with their own timestamp. The operation $r_2(A)$ ends up reading what it wrote itself. The operation $r_4(A)$ ends up reading what was written by T_3.

Basic version: The three writes succeed and overwrites the old value. Then, $r_2(A)$ makes T_2 restart and then either $r_4(A)$ happens, followed by $w_2(A)$ and $r_2(A)$ (which all succeed, since T_2 has a new, larger timestamp) or $w_2(A)$ happens and then some order of $r_2(A)$ and two $r_4(A)$ (the first aborts, but succeeds the second time).

(b) MVCC version: The two writes create two versions of A, each with their own timestamp. The operation $r_4(A)$ ends up reading what was written by T_3 and $r_2(A)$ reads what was written by T_1.

Basic version: The two writes succeed and overwrites the old value. Then, $r_4(A)$ succeeds, but $r_2(A)$ aborts the first time and succeeds the second.

(c) MVCC version: The first two writes create two different versions of the item, then, $r_3(A)$ reads what was written by T_1 and $w_2(A)$ aborts and succeed in creating a new version the second time.

Basic version: The two first writes succeed, $r_3(A)$ then aborts and then, in some order, we get $r_3(A)$ and two $w_2(X)$ (the first aborts, the second succeed).

References

[1] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson Education, 7th edition, 2016.

[2] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Pearson Education, 2nd edition, 2009.