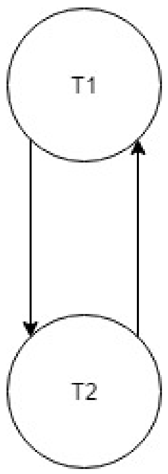
Assignment 3 **Alan Ou**
301320642

Question 1

Using R(x) to mean read, and W(z) to mean write, and assuming that the two transactions are independent:

a) S(x) R(x) U(x)
b) X(z) W(z) U(z)

Question 2



The following schedule is serializable but not conflict serializable, as shown in the dependency graph.

T1: W1(A) W1(B) R1(C)

T2: W2(A) W2(A) R2(B)

W1(A) W2(A) W2(A) R2(B) W1(B) R1(C)

The schedule is not conflict serializable because as shown in the dependency graph, there is a cycle. But the schedule is serializable because equivalent to the serial schedule:

W1(A) W1(B) R1(C) W2(A) W2(A) R2(B)

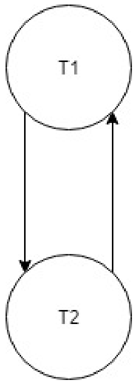
Question 3

a) No, the schedule S is not an avoid cascading abort schedule because in order to qualify to be an avoid cascading abort schedule, transactions must never read uncommitted writes, however, in schedule S, transaction 3 (R3(A)) reads the uncommitted writes of transaction 1 (W1(A)).
b) Yes, the schedule S is recoverable because as mentioned in the class notes, every "transaction commits only after each transaction from which it has read has commits." More specifically, transaction 3 commits after transaction 2, which commits after transaction 1.
c) No, the schedule S is not a strict schedule because as mentioned in the class notes, in order to qualify to be a strict schedule, "whenever transaction T writes to a data object, no other transaction can read or write to that object until transaction T has committed or aborted". However, in schedule S, transaction 1 (W1(A)) and transaction 3 (R3(A)) writes and reads respectively on object A after transaction 2 (W2(A)) writes on A, but before transaction 2 commits.

Assignment 3 **Alan Ou**
301320642

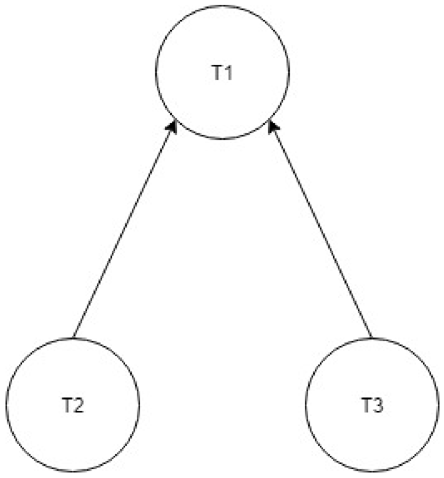
Question 4

a)



This schedule is not conflict serializable because the output of T1 (W1(B)) depends on the output of T2 (W2(B)) and also the output of T2 (W2(A)) depends on the output of T1 (W1(A)).

b)



This schedule is conflict serializable.

The equivalent serial schedule is:

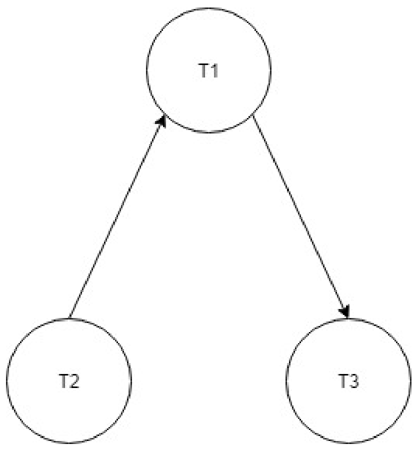
T2: W2(C)

T3: W3(B) R3(B) R3(A)

T1: R1(A) R1(C) W1(B)

W2(C) W3(B) R3(B) R3(A) R1(A) R1(C) W1(B)

c)



This schedule is conflict serializable.

The equivalent serial schedule is:

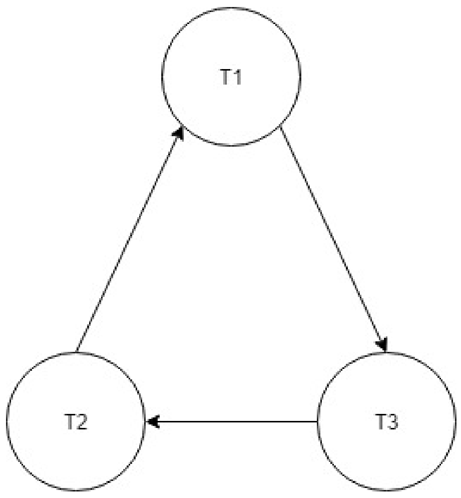
T2: W2(B) R2(A)

T1: R1(A) W1(B) W1(C) W1(A)

T3: W3(B) R3(B)

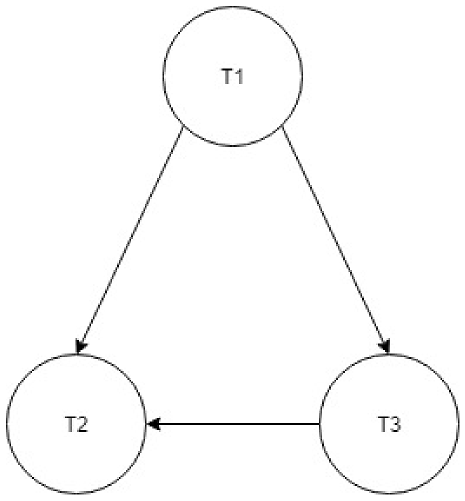
W2(B) R2(A) R1(A) W1(B) W1(C) W1(A) W3(B) R3(B)

d)



This schedule is not conflict serializable because the output of T3 (W3(A)) depends on the output of T1 (R1(A)), the output of T2 (W2(B)) depends on the output of T3 (R3(B)), and the output of T1 (W1(C)) depends on the output of T2 (R2(C)).

e)



This schedule is conflict serializable.

The equivalent serial schedule is:

T1: R1(A) R1(B) R1(C) W1(C)

T3: W3(A) W3(B) W3(B)

T2: R2(A) W2(C) R2(C)

R1(A) R1(B) R1(C) W1(C) W3(A) W3(B) W3(B) R2(A) W2(C) R2(C)

Assignment 3 **Alan Ou**
301320642

Question 5

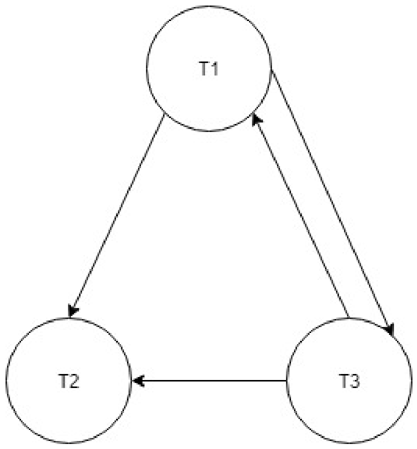
a) **Table 1**

X(D) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
S(A) at **time t3** is marked as **granted**
S(B) at **time t4** is marked as **granted**
X(A) at **time t5** is marked as **blocked**
X(C) at **time t6** is marked as **granted**
S(C) at **time t7** is marked as **blocked**
S(D) at **time t7** is marked as **blocked**
S(E) at **time t8** is marked as **granted**
X(E) at **time t9** is marked as **granted**
S(E) at **time t10** is marked as **granted**

Table 2

S(A) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
X(B) at **time t2** is marked as **granted**
S(B) at **time t3** is marked as **blocked**
X(A) at **time t4** is marked as **granted**
S(C) at **time t5** is marked as **granted**
X(C) at **time t6** is marked as **blocked**

b) **Table 1**



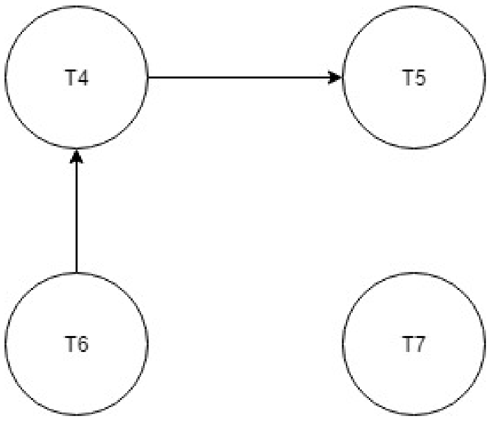
There exists a deadlock because in the wait-for graph, we can see a cycle between transaction 1 and transaction 3.

At time t1, transaction 1 puts X(D). At time t7, transaction 3 attempts to put S(D), therefore the first edge.

At time t3, transaction 3 puts S(A). At time t5, transaction 1 attempts to put X(A), therefore the second edge.

The two edges create a cycle in the wait-for graph, therefore shows that a deadlock exists.

Table 2



There does not exist a deadlock in the lock requests for table 2 because in the wait-for graph, there is no cycle.

c) **Table 1**

X(D) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
S(A) at **time t3** is marked as **granted**
S(B) at **time t4** is marked as **granted**
X(A) at **time t5** is marked as **blocked**
X(C) at **time t6** is marked as **granted**
S(C) at **time t7** is marked as **blocked**
S(D) at **time t7** is marked as **aborted**
S(E) at **time t8** is marked as **granted**
X(E) at **time t9** is marked as **granted**
S(E) at **time t10** is marked as **aborted**

After removing all locks from transactions of aborted actions, the resulting granted and blocked locks are:

X(D) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
S(B) at **time t4** is marked as **granted**
X(A) at **time t5** is marked as **blocked**
X(C) at **time t6** is marked as **granted**
S(C) at **time t7** is marked as **blocked**
S(E) at **time t8** is marked as **granted**
X(E) at **time t9** is marked as **granted**

Table 2

S(A) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
X(B) at **time t2** is marked as **granted**
S(B) at **time t3** is marked as **aborted**
X(A) at **time t4** is marked as **granted**
S(C) at **time t5** is marked as **granted**
X(C) at **time t6** is marked as **blocked**

After removing all locks from transactions of aborted actions, the resulting granted and blocked locks are:

S(A) at **time t2** is marked as **granted**
X(B) at **time t2** is marked as **granted**
X(A) at **time t4** is marked as **granted**
S(C) at **time t5** is marked as **granted**
X(C) at **time t6** is marked as **blocked**

d) **Table 1**

X(D) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted - aborted**
S(A) at **time t3** is marked as **granted - aborted**
S(B) at **time t4** is marked as **granted**
X(A) at **time t5** is marked as **granted**
X(C) at **time t6** is marked as **granted - aborted**
S(C) at **time t7** is marked as **granted**
S(D) at **time t7** is marked as **- transaction aborted**
S(E) at **time t8** is marked as **- aborted**
X(E) at **time t9** is marked as **- aborted**
S(E) at **time t10** is marked as **granted – transaction aborted**

After removing all locks from transactions of aborted actions, the resulting granted and blocked locks are:

X(D) at **time t1** is marked as **granted**
S(B) at **time t4** is marked as **granted**
X(A) at **time t5** is marked as **granted**
S(C) at **time t7** is marked as **granted**

Table 2

S(A) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
X(B) at **time t2** is marked as **granted**
S(B) at **time t3** is marked as **blocked**
X(A) at **time t4** is marked as **granted**
S(C) at **time t5** is marked as **granted - aborted**
X(C) at **time t6** is marked as **granted**

After removing all locks from transactions of aborted actions, the resulting granted and blocked locks are:

S(A) at **time t1** is marked as **granted**
S(A) at **time t2** is marked as **granted**
X(B) at **time t2** is marked as **granted**
S(B) at **time t3** is marked as **blocked**
X(A) at **time t4** is marked as **granted**
X(C) at **time t6** is marked as **granted**

Question 6

Assuming four levels of granularity, the following abbreviations will be used: database level (abbreviated D), table level with the Registration table (abbreviated R) and Students table (abbreviated S).

a) IS(D) IS(S) IS(S1-S300) S(S1:1 - S300:200) **OR** simply putting IS(D)
b) SIX(D) SIX(S) SIX(S12-S56) S(S12:1 – S56:200) X(S18:5)
c) IX(D) IX(R) IX(R1-R400) X(R1)
d) IX(D) IX(R) IX(R1-R400) X(R1:1 – R400:200)
e) IX(D) X(S) X(R) X(S1-S300) X(R1-R400) X(S1:1-S300:200) X(R1:1 – R400:200)

Question 7

```
a) S-lock A
   S-lock B
           U-lock A
   S-lock E
           U-lock B
   S-lock K
           U-lock E
   Read 17*
           U-lock K
```

b) X-lock A
X-lock B
    U-lock A
X-lock D
    U-lock B
X-lock I
Insert 7*
Split I
Update D
    U-lock D
    U-lock I

c) X-lock A
X-lock C
X-lock F
X-lock M
    U-lock A
    U-lock C
    U-lock F
Delete 28*
    U-lock M

d) X-lock A
X-lock C
    U-lock A
X-lock G
X-lock P
Insert 51*
Split P
Update G
Split G
Update C
    U-lock C
U-lock G
U-lock P

Question 8

As copied from the class notes, the three test conditions are:

Test 1: For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins.

Test 2: For all i and j such that $T_i < T_j$, check that

- T_i completes before T_j begins its Write phase +
- WriteSet(T_i) ∩ ReadSet(T_j) is empty

Test 3: For all i and j such that $T_i < T_j$, check that

- Ti completes Read phase before Tj completes its read +
- WriteSet(Ti) ∩ ReadSet(Tj) is empty +
- WriteSet(Ti) ∩ WriteSet(Tj) is empty

Transaction T1 – Does not validate

Transaction T1 does not validate successfully because it fails all three test conditions. Testing the first condition, T1 fails because right after T1 initiates Read phase, T2 initiates its Read phase, therefore T1 does not complete before T2 begins.

Testing the second condition, WriteSet(T_1) ∩ ReadSet(T_2) is not empty because the object F appears in the WriteSet of T1 and also appears in the ReadSet of T2. The third test builds on top of test 2, therefore will fail it as well based on the same reasons as test 2.

Transaction T2 – Does not validate

Transaction T2 does not validate successfully because it fails all three test conditions. Testing the first condition, T2 fails because shortly after T2 initiates validation phase, T3 initiates its Read phase, therefore T2 does not complete before T3 begins.

Testing the second condition, WriteSet(T_2) ∩ ReadSet(T_3) is not empty because the object A appears in the WriteSet of T2 and also appears in the ReadSet of T3. The third test builds on top of test 2, therefore will fail it as well based on the same reasons as test 2.

Transaction T3 – Does not validate

Transaction T3 does not validate successfully because it fails all three test conditions. Testing the first condition, T3 fails because shortly after T3 initiates Read phase, T4 initiates its Read phase, therefore T3 does not complete before T4 begins.

Testing the second condition, WriteSet(T_3) ∩ ReadSet(T_4) is not empty because the object C appears in the WriteSet of T3 and also appears in the ReadSet of T4. The third test builds on top of test 2, therefore will fail it as well based on the same reasons as test 2.

Transaction T4 – Validates

Transaction T4 validates successfully. Testing the first condition, T4 fails because right after T4 initiates Read phase, T5 initiates its Read phase, therefore T4 does not complete before T5 begins.

Before testing the second test condition, lets first state some assumptions. Using the information that the "write phase is completed immediately after the completion of the validation phase", I assume that the write phase also starts immediately after the completion of the validation phase. Now, testing the second test condition. At event number 13, T4 completes Validation phase; this also means that immediately after, the write phase is also completed, completing the entire transaction. At event number 15, T5 completes Validation phase, meaning that only at event number 15, that T5 starts its write phase. Using this information, T4 satisfies part one of the second condition, that T4 completes before T5 starts its Write phase. T4 also satisfies the second part of the second test condition, which states that "WriteSet(Ti) ∩ ReadSet(Tj) is empty". Therefore, T4 satisfies the second test condition, which means that the transaction passes the validation stage.

Transaction T5 – Validates

Transaction T5 validates successfully because it is the last transaction, therefore does not meet the condition $T_i < T_j$.

Question 9

S: R1(A); R2(A); W2(B); W3(B); W1(B); C1; C2; C3

a) Using timestamp-based concurrency control protocols, each transaction will be provided a timestamp (TS(T)) when it begins; for example, in schedule S, transaction 1 will be provided a smaller timestamp (TS(T1) = 1) than transaction 2, which is provided a smaller timestamp (TS(T2) = 2) than transaction 3 (TS(T3) = 3), using the assumption that **transaction Ti has a timestamp i.**

Then, "each object is given a read-timestamp (RTS(O)) and a write-timestamp (WTS(O))"; these timestamps represent the largest timestamp of any transaction that successfully executed a read(O) and write(O) respectively.

Then, through comparing timestamps, a violation may occur, leading to the action being aborted, or the action will be allowed to do the read or write it needs to. For reads, if TS(T) < WTS(O), abort and restart transaction, otherwise allow. For writes, abort and restart transaction if TS(T) < RTS(O) or TS(T) < WTS(O), otherwise allow.

Assuming objects A and B initially have RTS = 0 and WTS = 0, and transactions have timestamps TS(T1) = 1, TS(T2) = 2, TS(T3) = 3.

Now, the first action. Transaction 1 (R1(A)) is allowed to read A since it is the first read on A, and TS(T1) = 1 > RTS(A) = 0. The RTS(A) is set to 1 because as stated in the protocol, we set RTS(A) to be the Max(RTS(A), TS(T)).

Similarly, transaction 2 (R2(A)) is allowed to read A since TS(T2) = 2 > RTS(A) = 1. The RTS(A) is set to 2.

Again, transaction 2 (W2(B)) is allowed to write B since it is the first write on B, and TS(T2) = 2 > WTS(B) = 0. The WTS(B) is set to 2 because as stated in the protocol, we set WTS(B) to be the Max(WTS(B), TS(T)).

Transaction 3 (R3(B)) is allowed to write B since TS(T3) = 3 > WTS(B) = 2. The WTS(B) is set to 3.

Transaction 1 (W1(B)) will be aborted because TS(T1) = 1 is not > WTS(B) = 3, therefore violates the protocol and therefore needs to be aborted and transaction will be restarted. This will cause the entire transaction one to be rolled back, which means that this and the first action will be removed.

b) The way timestamp-based concurrency control protocols with Thomas' Write Rule handles schedule S is exactly the same as the timestamp-based concurrency control protocols without Thomas' Write Rule except that when we use Thomas’ Write Rule, we ignore outdated writes when TS(T) < WTS(O), instead of restarting T.

Handling schedule S, assuming objects A and B initially have RTS = 0 and WTS = 0, and transactions have timestamps TS(T1) = 1, TS(T2) = 2, TS(T3) = 3.

The first action, transaction 1 (R1(A)) is allowed to read A because of the same reasons as mentioned above, in part A without Thomas' Write Rule. Furthermore, this first action R1(A) will not be rolled back because of the reason stated below. Transaction 2 (R2(A)), transaction 2 (W2(B)), transaction 3 (R3(B)) will all be allowed because of the reasons as stated above in part A.

Transaction 1 (W1(B)) will be ignored because TS(T1) = 1 is not > WTS(B) = 3, therefore violates the protocol, however Thomas’ Write Rules says that we can ignore writes that have TS(T) < WTS(O), instead of restarting T1. Therefore, the first action stays because of this rule.

Question 10

Assuming no previous checkpointing, therefore both the transaction table and dirty page table are both empty. The transaction table records will consist of <transID, lastLSN, status>, while the dirty page table records will consist of <pageID, recLSN>.

a) Analysis begins by finding the most recent begin_checkpoint log record, which is LSN 00, and starting at the end_checkpoint record, which is LSN 10.

At log record 20 (LSN 20): Add (T1, 20, active) to transaction table. Add (P5, 20) to dirty page table

At log record 30 (LSN 30): Add (T2, 30, active) to transaction table. Add (P3, 30) to dirty page table

At log record 40 (LSN 40): Change status of T2 from “active” to “committed”

At log record 50 (LSN 50): Remove T2 from transaction table

At log record 60 (LSN 60): Add (T3, 60, active) to transaction table. P3 entry already exists in dirty page table, therefore is not changed.

At log record 70 (LSN 70): Change (T1, 20, active) to (T1, 70, aborted)

The analysis phase is now complete. The final transaction table and dirty page table are as follows.

Transaction Table

transID	lastLSN	status
T1	70	aborted
T3	60	active

Dirty Page Table

pageID	recLSN
P5	20
P3	30

b) Redo starts at the "smallest recLSN in dirty page table after analysis", which in this case would be LSN 20.

At log record 20 (LSN 20): Redo changes to P5

At log record 30 (LSN 30): Retrieve P3 and check pageLSN. If pageLSN on disk >= LSN (30), then there is no need to redo anything because the changes have already been written to disk, otherwise redo changes to P5.

At log record 40 (LSN 40): No change to dirty pages, therefore no need to redo anything

At log record 50 (LSN 50): No change to dirty pages, therefore no need to redo anything

At log record 60 (LSN 60): Redo changes to P3

At log record 70 (LSN 70): No change to dirty pages, therefore no need to redo anything

c) Undo scans backwards from the end of the log, therefore using the transaction table constructed during the analysis stage, the LSN of the most recent log record is chosen as the start. The most recent log record is LSN 70.

Initial loser transactions: LSN 70, LSN 60

At log record 70 (LSN 70): (T1, 70, aborted) is undone to (T1, 20, active) and loser transactions now becomes: Loser transactions: LSN 60, LSN 20

At log record 60 (LSN 60): Changes on P3 are undone and a CLR is written

At log record 20 (LSN 20): Changes on P5 are undone and a CLR is written

Assignment 3 **Alan Ou**
301320642

Question 11

a) **Transaction 3 and transaction 4** will likely be undone because they have yet commit before the crash.

b) The values of data objects A, B, C and D on disk are as follows:

A: 20 B: 40 C: 50 D: 50

c) Assuming the numbers on the left of the actions are LSN, the contents of the log after recovery completion is as follows:

10 <CLR: Undo T4 LSN 8>
11 <T4 end>
12 <CLR: Undo T3 LSN 7>
13 <T3 end>