1. DB Anomalies:

a) r3(x); r1(x); r2(x); r3(z); r2(y); w3(z); w2(y); r2(z), r1(z); a1; c2; c3
On x: r3; r1; r2; a1; c2; c3
On y: r2; w2; c2
On z: r3; w3; r2; r1; a1; c2; c3
No anomaly

b) r1(x); r2(z); r3(x); w3(x); r3(z); r1(y); r2(x); w3(z); r2(y); w2(y); r1(y); c3; c1; c2
On x: r1; r3; w3; r2; c3; c1; c2
On y: **r1**; r2; **w2**; **r1**; c1; c2 **/ unrepeatable read**
On z: r2; r3; w3; c3; c2

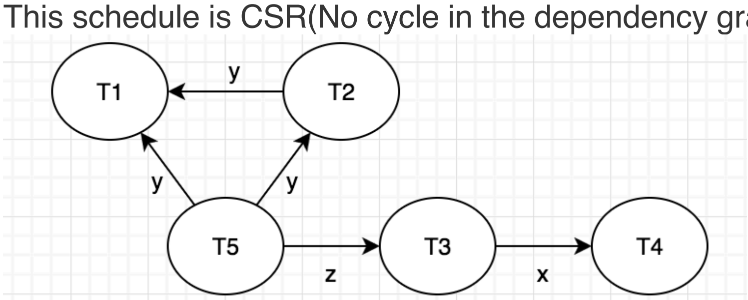
c) r1(x); w1(x); r4(x); r3(x); r2(y); r2(x); r1(y); w2(x); r3(y); c1; w4(x); w3(y); c2; c4; c3
On x: r1; w1; r4; r3; r2; **w2**; c1; **w4**; c2; c4; c3 **/update loss**
On y: r2; r1; r3; c1; w3; c2; c3

d) r1(x); r2(x); w2(x); r1(x); w1(x); w3(y); r3(x); r2(y); w3(y); r3(y); a1; c3; c2
On x: r1; r2; w2; r1; w1; **r3**; a1; c3; c2 **/ r3(x) dirty read.**
On y: w3; r2; w3; r3; c3; c2

2. Serializability

a) w5(y); r5(z); w3(x); r2(y); w1(y); w3(z); w4(x) In case of serializability, for possible equivalent serial schedule(s), which transaction could be the first transaction?

This schedule is CSR(No cycle in the dependency graph), and therefore it is VSR.

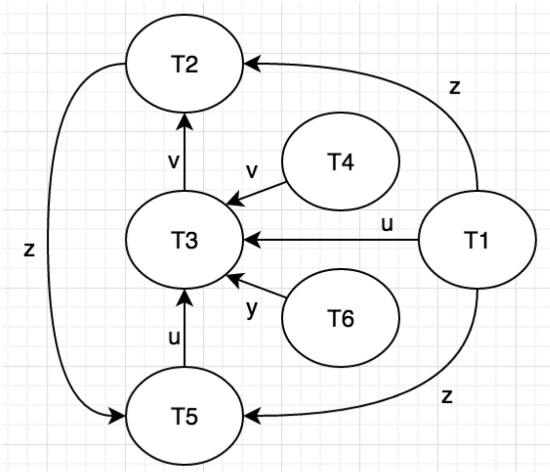


On x: w3; w4
On y: w5; r2; w1
On z: r5; w3

There exist many possible equivalent serial schedules, two of them are
w5(y); r5(z); r2(y); w1(y); w3(x); w3(z); w4(x) (T5 -> T2 -> T1 -> T3 -> T4), or
w5(y); r5(z); w3(x); w3(z); w4(x); r2(y); w1(y) (T5 -> T3 -> T4 -> T2 -> T1)
Base on the dependency graph, transaction 5 could be the first transaction for possible equivalent serial schedules.

b) r1(u); r4(v); r5(x); r6(y); w1(z); w2(z); r5(u); w5(x); w3(y); w3(v); r2(v); w3(u); w5(z) In case the schedule is CSR and/or VSR, write down one possible equivalent serial schedule.

This schedule is Non-CSR. There is a cycle in the dependency graph:



Cycle: u(T5->T3),v(T3->T2),z(T2->T5)

On x: r5; w5
On y: r6; w3
On z: w1; w2; w5
On u: r1; r5; w3
On v: r4; w3; r2

This schedule is Non-VSR:

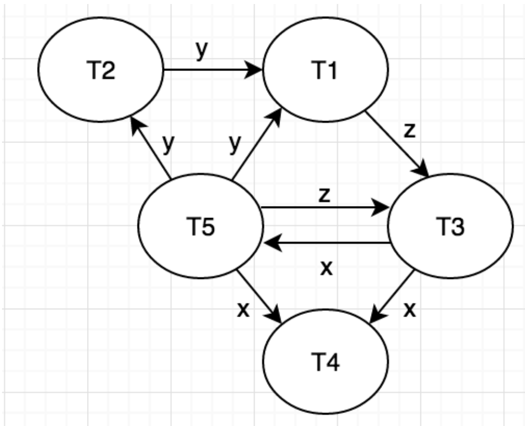
Final writes = {w5 on x, w3 on y, w5 on z, w3 on u, w3 on v}
Reads from = {r2 from w3 on v}

On x: r5; **w5**
On y: r6; **w3** (T6 < T3)
On z: w1; w2; **w5** (T1 < T5, T2 < T5)
On u: r1; r5; **w3** (T1 < T3, T5 < T3)
On v: r4; **w3**; **r2** (T4 < T3, T3 < T2)

According to final writes: T2 < T5 and T5 < T3, we can say T2 < T3
According to reads from: T3 < T2
We cannot say if T2 comes before or after T3 in a serial schedule.
It is not possible to find serial schedule equivalent to the original schedule.

c) r1(z); w5(y); r5(z); w3(x); r2(y); w5(x); w1(y); w3(z); w4(x) In case the schedule is CSR and/or VSR, write down one possible equivalent serial schedule.

This schedule is Non-CSR. There exist a cycle in the dependency graph:



Cycle: z(T5->T3),x(T3->T5)
Cycle: y(T5->T1),z(T1->T3),x(T3->T5)

On x: w3; w5; w4
On y: w5; r2; w1
On z: r1; r5; w3

This schedule is VSR: Final writes = {w4 on x, w1 on y, w3 on z}
Reads from = {r2 from w5 on y}

On x: w3; w5; **w4** (T3 < T4, T5 < T4)
On y: w5; **r2**; **w1** (T5 < T1, T2 < T1, T5 < T2)
On z: r1; r5; **w3** (T1 < T3, T5 < T3)

One possible equivalent serial schedule:
w5(y); r5(z); w5(x); r2(y); r1(z); w1(y); w3(x); w3(z); w4(x)
(T5 -> T2 -> T1 -> T3 -> T4)
Final writes = {w4 on x, w1 on y, w3 on z}
Reads from = {r2 from w5 on y}

3.Two-PhaseLock(2PL):

Verify whether the following schedule is consistent with 2PL.
r1(y); r3(z); r1(x); r2(z); r3(y); r2(x); r1(x); w1(x); w2(y); w3(z); w1(y); r1(z)

	T1	T2	T3
1	r1(y)		
2			r3(z)
3	r1(x)		
4		r2(z)	
5			r3(y)
6		r2(x)	
7	r1(x)		
8	w1(x)		
9		w2(y)	
10			w3(z)
11	w1(y)		
12	r1(z)		

A: **(1 < Unlock r1(y) < Lock w2(y) < 9)**
B: (5 < Unlock r3(y) < Lock w2(y) < 9)
C: **(4 < Unlock r2(z) < Lock w3(z) < 10)**
D: (6 < Unlock r2(x) < Lock w1(x) < 8)
E: (9 < Unlock w2(y) < Lock w1(y) < 11)
F: **(10 < Unlock w3(z) < Lock r1(z) < 12)**

Conflicting lock operation: {A,C,F}, {B,C}, {A,D} and {A,E}

As it is shown in the conflicting lock operations, T1, T2 and T3 cannot separate their growing and shrinking phases in order to be consistent with the lock compatibility on the same objects.

T1 need to release readlock(y) after time 1 for T2 to get a writelock(y) before time 9. Then T1 needs to get a writelock(y), a readlock(z) and a writelock(x) before releasing readlock(y) to separate its growing and shrinking phases, and this writelock(y) must be released only after time 11. This is conflicting with T2 to get a writelock(y) before time 9. Also, this readlock(z) must be released only after time 12, it is conflicting with T3 to get a writelock(z) before time 10. Therefore, T1, T2 and T3 have conflicts, and they cannot be 2PL at the same time. The give schedule is not consistent with 2PL.

The tables on next page illustrate how T1, T2 and T3 are conflicting.

Following is one example of how T1, T2 and T3 are conflicting and the given schedule is not consistent with 2PL:

	T1	T2	T3
	read_lock(y)		
1	r1(y)		
			read_lock(z)
2			r3(z)
	read_lock(x)		
3	r1(x)		
		read_lock(z)	
4		r2(z)	
			read_lock(y)
5			r3(y)
		read_lock(x)	
6		r2(x)	
7	r1(x)		
	T1 needs to perform **read_lock(z)**		
	T1 needs to perform **write_lock(y)**		
	T1 needs to perform **unlock(y)**		
			unlock(y)
		T2 need to perform **write_lock(y)**	
		unlock(x)	
	write_lock(x)		
8	w1(x)		
	unlock(x)		
9		w2(y)	
		unlock(y)	
		unlock(z)	
			T3 need to perform **write_lock(z)**
10			w3(z)
			unlock(z)
	T1 needs to keep **write_lock(y)** until time 11		
11	w1(y)		
	unlock(y)		
	T1 needs to keep **read_lock(z)** until time 12		
12	r1(z)		
	unlock(z)		

Following is an additional explanation of the previous table about how T1, T2 are conflicting and the given schedule is not consistent with 2PL:

	T1	T2	T3
	read_lock(y)		
1	r1(y)		
			read_lock(z)
2			r3(z)
	read_lock(x)		
3	r1(x)		
		read_lock(z)	
4		r2(z)	
			read_lock(y)
5			r3(y)
		read_lock(x)	
6		r2(x)	
7	r1(x)		
	To make 2PL, T1 need to perform **write_lock(x)** here to separate its growing and shrinking phases.	**(T1 need perform write_lock(x) before T2 perform unlock(x), which is a conflict.)**	
	unlock(y)		
			unlock(y)
		To make 2PL, T2 need to perform **write_lock(y)** here to separate its growing and shrinking phases.	
		unlock(x)	
	write_lock(x)		
8	w1(x)		
	unlock(x)		
9		w2(y)	
		unlock(y)	
		unlock(z)	
			write_lock(z)
10			w3(z)
			unlock(z)
	write_lock(y)		
11	w1(y)		
	unlock(y)		
	read_lock(z)		
12	r1(z)		
	unlock(z)		