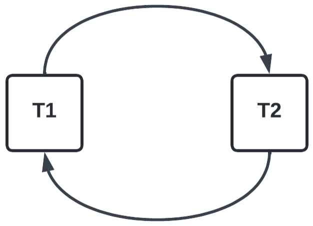
Assignment 2
Group 5: Qi Lin, Xiaoying Li, Ziying Yu
4/28/2022

Question 1
Relationship:
CSR → VSR (Every CSR is also a VSR, the set of CSR schedules is a subset of the set of VSR schedules.)
VSR → FSR

Assume:
Initially x=1, y=1, z=1
w1(y=10), w1(x=15), w1(z=12)
w2(y=20), w2(x=25), w2(z=14)
w3(y=30), w2(x=35), w3(z=16)

- Schedule 1: final-state serializable, not view serializable, not conflict serializable
  - s = w1(y) w2(x) r1(x) w1(x) r2(y)

FSR:
y: 1 → 10
x: 1 → 25 → 15
CSR:
conflicts: T1: w(y) & T2: r(y) and T2: w(x) & T1: r(x)
SG(s): → not CSR (SG(s) is cyclic graph)



T1	T2
w(y)	
	w(x)
r(x)	
w(x)	
	r(y)

	x	y
Initial Read	null	null
Update Read	T2 → T1	T1 → T2
Final Write	T1	T1

- s’ = w1(y) r1(x) w1(x) w2(x) r2(y)

FSR: → not FSR: s, s’ FSR are different

y: 1 → 10

x: 1 → 15 → 25

VSR:

T1	T2
w(y)	
r(x)	
w(x)	
	w(x)
	r(y)

	x	y
Initial Read	T1	null
Update Read	null	T1 → T2
Final Write	T2	T1

not VSR: s, s’: initial read x are not the same (s: null; s’: T1 initial read x)

s, s’: update read x are not the same (s: update read T2: w(x) → T1: r(x); s’: null)

s, s’: final write x are not the same (s: T1 final write x; s’: T2 final write x)

- s’’ = w2(x) r2(y) w1(y) r1(x) w1(x)

FSR: → FSR: s, s’’ FSR are the same

y: 1 → 10

x: 1 → 25 → 15

VSR:

T1	T2
	w(x)
	r(y)
w(y)	
r(x)	
w(x)	

	x	y
Initial Read	null	T2
Update Read	T2 → T1	null
Final Write	T1	T1

not VSR: s, s’’: initial read y are not the same (s: null; s’’: T2 initial read y)

s, s’’: update read y are not the same (s: update read T1: w(y) → T2: r(y); s’’: null)

s, s’’: final write x, y are the same

- Schedule 2: final-state serializable, not view serializable, not conflict serializable
  - s = w1(y) r2(y) w2(x) r1(x) w1(x)

FSR:

x: 1 → 25 → 15

y: 1 → 10

CSR:

conflicts: T1: w(y) & T2: r(y), T2: w(x) & T1: r(x), and T2: w(x) & T1: w(x)

SG(s): → not CSR (SG(s) is cyclic graph)

$$\begin{tikzcd}
\boxed{T1} \arrow[r, bend left] & \boxed{T2} \arrow[l, bend left]
\end{tikzcd}$$

T1	T2
w(y)	
	r(y)
	w(x)
r(x)	
w(x)	

	x	y
Initial Read	null	null
Update Read	T2 → T1	T1 → T2
Final Write	T1	T1

- s' = r2(y) w2(x) w1(y) r1(x) w1(x)

FSR: → FSR: s, s' FSR are the same

x: 1→ 25 → 15

y: 1 → 10

VSR:

T1	T2
	r(y)
	w(x)
w(y)	
r(x)	
w(x)	

	x	y
Initial Read	null	T2
Update Read	T2 → T1	null
Final Write	T1	T1

not VSR: s, s': initial read y are not the same (s: null; s': T2 initial read y)

s, s': update read y are not the same (s: update read T1: w(y) → T2: r(y); s': null)

s, s': final write x, y are the same

- s'' = w1(y) r1(x) w1(x) r2(y) w2(x)

FSR: → not FSR: s, s'' FSR are different

x: 1 → 15 → 25

y: 1 → 10

VSR:

T1	T2
w(y)	
r(x)	
w(x)	
	r(y)
	w(x)

	x	y
Initial Read	T1	null
Update Read	null	T1 → T2
Final Write	T2	T1

not VSR: s, s’’: initial read x are not the same (s: null; s’’: T1 initial read x)

s, s’’: update read x are not the same (s: update read T2: w(x) → T1: r(w); s’’: null)

s, s’’: final write x are not the same (s: T1 write x; s’’: T2 write x)

- Schedule 3: final-state serializable, view serializable, not conflict serializable
  - s = w1(y) r3(y) w2(y) w2(y) r1(x) w1(x) w3(y)

FSR:

x: 1 → 25 → 15

y: 1 → 10 → 20 → 30

CSR:

conflicts:

T1: w(y) T3: r(y)

T1: w(y) T2: w(y)

T1: w(y) T3: w(y)

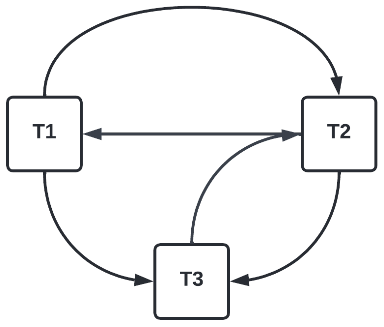
T3: r(y) T2: w(y)

T2: w(y) T3: w(y)

T2: w(x) T1: r(x)

T2: w(x) T1: w(x)

SG(s): → not CSR (SG(s) is cyclic graph)



VSR:

T1	T2	T3
w(y)		
		r(y)
	w(y)	
	w(x)	
r(x)		
w(x)		
		w(y)

	x	y
Initial Read	null	null
Update Read	T2 → T1	T1 → T3
Final Write	T1	T3

- ○ s'= w2(y) w2(x) w1(y) r1(x) w1(x) r3(y) w3(y)

FSR: → FSR: s, s' FSR are the same

x: 1 → 25 → 15

y: 1 → 20 → 10 → 30

VSR: → VSR: s, s' initial read/update read/final write x, y are the same

T1	T2	T3
	w(y)	
	w(x)	
w(y)		
r(x)		
w(x)		
		r(y)
		w(y)

	x	y
Initial Read	null	null
Update Read	T2 → T1	T1 → T3
Final Write	T1	T3

- Schedule 4: final-state serializable, view serializable, not conflict serializable
  - ○ s = w1(y) w2(y) w2(x) r1(x) w1(x) w3(y)

FSR: → FSR: s, s' FSR are the same

x: 1 → 25 → 15

y: 1 → 10 → 20 → 30

CSR:

conflicts:

T1: w(y) T2: w(y)

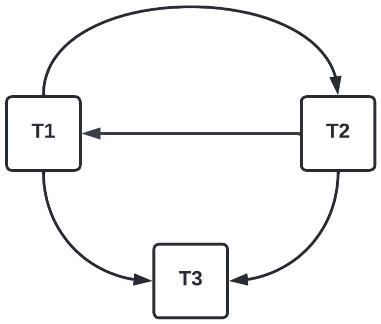
T1: w(y) T3: w(y)

T2: w(y) T3: w(y)

T2: w(x) T1: r(x)

T2: w(x) T1: w(x)

SG(s): ⟶ not CSR (SG(s) is cyclic graph)



VSR:

T1	T2	T3
w(y)		
	w(y)	
	w(x)	
r(x)		
w(x)		
		w(y)

	x	y
Initial Read	null	null
Update Read	T2 ⟶ T1	null
Final Write	T1	T3

- s' = w2(y) w2(x) w1(y) r1(x) w1(x) w3(y)

FSR: ⟶ FSR: s, s' FSR are the same

x: 1 ⟶ 25 ⟶ 15

y: 1 ⟶ 20 ⟶ 10 ⟶ 30

VSR: ⟶ VSR: s, s' initial read/update read/final write x, y are the same

T1	T2	T3
	w(y)	
	w(x)	
w(y)		
r(x)		
w(x)		
		w(y)

	x	y
Initial Read	null	null
Update Read	T2 ⟶ T1	null
Final Write	T1	T3

- Schedule 5: final-state serializable, view serializable, conflict serializable
  - s = w1(x) r2(x) r2(y) r3(z) w1(z)

FSR:

x: 1 ⟶ 15

y: 1

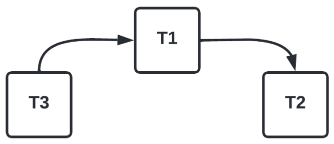
z: 1 ⟶ 12

T1	T2	T3
w(x)		
	r(x)	
	r(y)	
		r(z)
w(z)		

CSR:

Conflicts: T1: w(x) & T2: r(x) and T3: r(z) & T1: w(z)

SG(s):



→ CSR (SG(s) is acyclic) → VSR → FSR

Question 2

A schedule S is recoverable if Ti reads from Tj, then Tj commits before Ti commits.

A schedule S is ACA if in S Ti reads some data item x from Tj, then Ti reads x after Tj commits.

A schedule S is strict if whenever transaction Ti reads from Tj or Ti overwrites value written by Tj, it does so after Tj commits or aborts.

- Schedule 1: recoverable, ACA, strict
    - Schedule 1 is recoverable since no other transactions read from x after T2 writes on x, and also no other transactions read from y after T1 writes on y.
    - Schedule 1 is ACA since no other transactions read from x after T2 writes on x and before T2 commits, and also no other transactions read from y after T1 writes on y and before T1 commits.
    - Schedule 1 is strict since no other transactions read from or write on x after T2 writes on x and before T2 commits, and also no other transactions read from or write on y after T1 writes on y and before T1 commits.

- Schedule 2: not recoverable, not ACA, not strict
    - Schedule 2 is not recoverable since T1 reads from y after T2 writes on y, which means T1 reads from T2, but then T2 commits after T1 commits.
    - Schedule 2 is not ACA since T2 reads from x after T1 writes on x but before T1 commits, which means T2 reads data from T1 before T1 commits. And also, T1 reads from y after T2 writes on y but before T2 commits, which means T1 reads data from T2 before T2 commits.
    - Schedule 2 is not strict since T2 read from x after T1 writes on x but before T1 commits, and also T1 reads from y after T2 writes on y but before T2 commits.

- Schedule 3: not recoverable, not ACA, not strict
  - Schedule 3 is not recoverable since T2 reads from x after T1 writes on x, which means T2 reads from T1, but then T1 aborts before T2 commits and T2 cannot roll back to the original value of x.
  - Schedule 3 is not ACA since T2 reads from x after T1 writes on x but before T1 aborts, which means T2 reads data from T1 before T1 aborts.
  - Schedule 3 is not strict since T2 reads from x after T1 writes on x but before T1 aborts.

- Schedule 4: recoverable, ACA, not strict
  - Schedule 4 is recoverable since no other transactions read from x after T2 writes on x, and also no other transactions read from y after T2 writes on y and T1 writes on y.
  - Schedule 4 is ACA since no other transactions read from x after T2 writes on x and before T2 commits, no other transactions read from x after T1 writes on x and before T1 aborts, and also no other transactions read from y after T2 writes on y and before T2 commits.
  - Schedule 4 is not strict since T1 overwrites on y after T2 writes on y but before T2 commits.

- Schedule 5: recoverable, not ACA, not strict
  - Schedule 5 is recoverable since T3 reads from x after T2 writes on x, which means T3 reads from T2, and then T2 commits before T3 commits.
  - Schedule 5 is not ACA since T3 reads from x after T2 writes on x but before T2 commits, which means T3 reads data from T2 before T2 commits.
  - Schedule 5 is not strict since T2 overwrites on x after T1 writes on x but before T1 aborts, and also T3 read from x after T2 writes on x but before T2 commits.

Reference:
https://www.gatevidyalay.com/recoverable-schedules-irrecoverable-schedules-non-serializable-schedules/#:~:text=Case%2D02%3A,then%20the%20schedule%20is%20recoverable.
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/ACR.html

Question 3

- Schedule 1: not 1-copy serializable
  - s' = r1(xa) w1(yb) w2(xb) r2(ya)

T1	T2
r(xa)	
w(yb)	
	w(xb)
	r(ya)

⟶ For T1 read xa, there is no previous transaction that wrote xa; xa is the most recent data

⟶ For T2 read ya, T2 did not reads yb which the previous transaction T1 wrote yb

- s’’ = w2(xb) r2(ya) r1(xa) w1(yb)

T1	T2
	w(xb)
	r(ya)
r(xa)	
w(yb)	

⟶ For T1 read xa, T1 did not read xb which the previous transaction T2 wrote xb

⟶ For T2 read ya, there is no previous transaction that wrote ya; ya is the most recent data

- Schedule 2: not 1-copy serializable
  - s = w1(ya) w2(xa) r1(xb) w1(xb) r2(yb)

T1	T2
w(ya)	
	w(xa)
r(xb)	
w(xb)	
	r(yb)

⟶ T1 did not read xa which the previous transaction T2 wrote xa

- s’ = w1(ya) r1(xb) w1(xb) w2(xa) r2(yb)

T1	T2
w(ya)	
r(xb)	
w(xb)	
	w(xa)
	r(yb)

⟶ T2 did not reads yb which the previous transaction T1 wrote ya

- s’’=w2(xa)r2(yb)w1(ya)r1(xb)w1(xb)

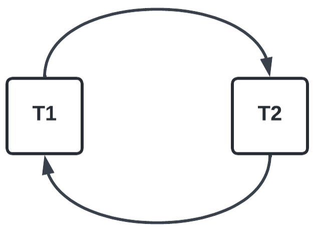
T1	T2
	w(xa)
	r(yb)
w(ya)	
r(xb)	
w(xb)	

⟶ T1 did not reads xb which the previous transaction T2 wrote xa

- Schedule 3: not 1-copy serialization

T1	T2
w(ya)	
	r(ya)
	w(xa)
r(xa)	
w(xa)	
w(xb)	

SG(s) : → not CSR (cyclic graph)



Question 4

- Summary of Quantifying Eventual Consistency with PBS

In this paper, the author proposed the probabilistic bounded staleness method, by using this method, the author quantifies the analysis of the staleness when achieving eventual consistency in different latency distributions. The author evaluates the staleness by two types of semantics, (K, p)-regular semantics and (Δ, p)-regular semantics. These two metrics provide quantitative expectations that a store will return a version that was written within the last K writes of the latest (where K = 1 is the latest) and that a store will return the latest version as of Δ seconds ago. By using sampling methods, the author sampled different process latency from different distributions, such as the latency of sending to N replicas, and the latency of waiting for a write response, all of which are sampled in different latency distributions. After sampling, the author builds a Monte Carlo simulation framework to test the quantitative expectations for the above two metrics. The results show that decreasing the mean and variance of write latency improves the probability of consistent reads. These results also explained why eventually consistent systems frequently return consistent data within tens of milliseconds while offering large latency benefits.

Here is an example to describe this simulation work, suppose we have some process latency distributions, like read latency distribution, write latency distribution, send to N replicas latency distribution, wait for write response latency distribution, wait for read response latency distribution. We can sample the process latency from these distributions, like 1ms, and 10ms. Now, we can calculate the expectation for returning the latest version of data, by calculating such expectation in the production latency distribution, we can get the practical performance of eventually consistent systems.

- Summary of Coordination Avoidance in Database systems

In this paper, the author proposed the theoretical answer that how we can avoid coordination during processing transactions, that is I-confluence property. When processing transactions in the database system, serializability is the key property to maintain the correctness of the whole processing, however, we usually need coordination to maintain the serializability or consistency, it increased latency, decreased throughput, and, in the event of partial failures, unavailability. The author explained that there existed some cases that can achieve correctness beyond the serializability, in such cases, we can keep the consistency of the system without coordination, which will reduce the latency of the system. As the author claims that this property—invariant confluence (I-confluence)—captures the potential scalability and availability of an application, independent of any particular database implementation. A set of transactions T is I-confluent with respect to invariant I if, for all I-T-reachable states D_i, D_j with a common ancestor state, $D_i \sqcup D_j$ is I-valid. In other words, for a given invariant in a set of transactions, if two states D_i, D_j are I-T-reachable states, which means there exists a (partially ordered) sequence of transaction and merge function invocations that yields these states, and these states have a common ancestor state, then the final state or merge state for D_i, D_j is still I-valid. I-valid means the consistency of state. The intuition behind this paper is that, if we start from a common state, there are many ways to achieve different states, but these states still keep the I-T-reachable property, when we merge or union these states, the final state is still keeping the valid property, like the correctness or consistency.

Here is an example from the paper that can explain this proof work, the constraint of equality. This constraint is invariant, assume two data states S_1 and S_2 are I-T-reachable based on the per-record in-equality invariant I_e, but $S_1 \sqcup S_2$ based on I_e is false. Then there must be an $r \in S_1 \sqcup S_2$ that violates I_e, r must appear in S_1, S_2, or both. But that would imply that one of S_1 or S_2 is not I-valid based on I_e, a contradiction. So, this invariant is I-confluence, we can maintain the correctness in such transactions without coordination.

Question 5

Consider the following schedule of two transactions:

w1(x:=1) r2(x) w2(y:=1) c2 r1(y) r1(x) w1(z:=1) c1

and originally x=0, y=0, and z=0.

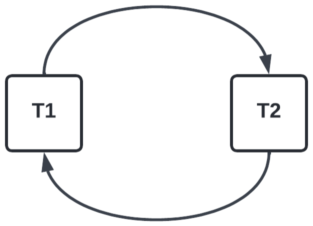
- At first, T1 starts, and it takes a snapshot of committed data, where x=0, y=0, and z=0.
- Then T1 writes on x with value 1 in its own snapshot.
- After that, T2 starts, and it also takes a snapshot of committed data, where x=0, y=0, and z=0, which doesn't reflect anything that T1 does since T1 hasn't committed any change.
- Then T2 reads from x with value 0, since it can only see its own update, but cannot see the concurrent update to x made by T1.
- Then T2 writes on y with value 1 in its own snapshot.

- Then T2 checks if there is no other concurrent transaction that has already written data on y that T2 intends to write, since T1 hasn’t committed any change, then T2 commits y=1.
- Then T1 reads from y with value 0, since it can only see its own update, but cannot see the concurrent update to y made by T2.
- Then T1 reads from x with value 1, since it can see its own update to x.
- Then T1 writes on z with value 1 in its own snapshot.
- Then T1 checks if there is no other concurrent transaction that has already written data on x and z that T1 intends to write, since T2 only commits y=1, then T1 commits x=1 and z=1.

T1	T2
start w(x:=1)	
	start r(x) → 0 w(y:=1) commit
r(y) → 0 r(x) → 1 w(z:=1) commit	

The above example schedule includes two transactions. And both transactions take a snapshot of committed data at the start; always reads and modifies data in their own snapshot; update of concurrent transactions are not visible to each other; writes complete when they commit and commits only if no other concurrent transaction has already written data that it intends to write. Therefore, the example schedule satisfies snapshot isolation.

The above schedule has two pairs of conflicting operations: w1(x:=1) & r2(x) and w2(y:=1) & r1(y). Its serialization graph is:



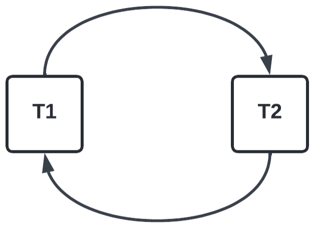
which is cyclic. Therefore, the example schedule does not satisfy CSR.

Question 6

- H1: w1(y) w2(x) r1(x) w1(x) r2(y) c1 c2
    - not RC: T1 read x from T2(wrote x), but T1 commit before T2
    - not ACA: since not RC
    - not ST: since not ACA
    - not CSR:

T1	T2
w(y)	
	w(x)
r(x)	
w(x)	
	r(y)

SG: $\longrightarrow$ not CSR (cyclic graph)



- not VSR:
    - s’ = w1(y) r1(x) w1(x) w2(x) r2(y)

T1	T2
w(y)	
r(x)	
w(x)	
	w(x)
	r(y)

	x	y
Initial Read	T1	null
Update Read	null	T1 $\rightarrow$ T2
Final Write	T2	T1

$\longrightarrow$ not VSR: s, s’: initial read x are not the same (s: null; s’: T1 initial read x)

s, s’: update read x are not the same (s: update read T2: w(x) $\longrightarrow$ T1: r(x) ; s’: null)

s, s’: final write x are not the same (s: T1 final write x; s’: T2 final write x)

- s’’ = w2(x) r2(y) w1(y) r1(x) w1(x)

T1	T2
	w(x)
	r(y)
w(y)	
r(x)	
w(x)	

	x	y
Initial Read	null	T2
Update Read	T2 → T1	null
Final Write	T1	T1

→ not VSR: s, s’’: initial read y are not the same (s: null; s’’: T2 initial read y)

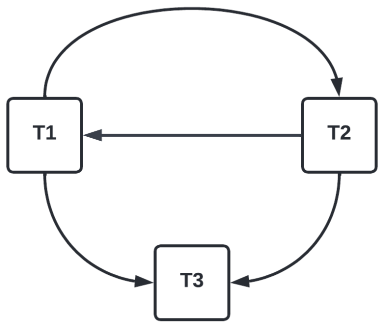
s, s’’: update read y are not the same (s: update read T1: w(y) → T2: r(y) ; s’’: null)

s, s’’: final write x, y are the same

- H2: w1(x) w2(x) w2(y) r1(y) w1(y) w3(x) c3 c1 c2
  - not RC: T1 read y after T2 wrote y, but T1 commit before T2 commit
  - not ACA: since not RC
  - not ST: since not ACA
  - not CSR:

T1	T2	T3
w(x)		
	w(x)	
	w(y)	
r(y)		
w(y)		
		w(x)

SG: cyclic



  - VSR:
    - s:

	x	y
Initial read:	null	null
Update read:	null	T2 → T1
Final write:	T3	T1

- s’ = w2(x) w2(y) w1(x) r1(y) w1(y) w3(x)

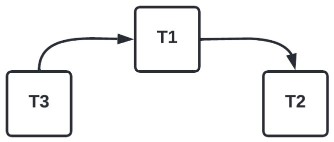
T1	T2	T3
	w(x)	
	w(y)	
w(x)		
r(y)		
w(y)		
		w(x)

	x	y
Initial read:	null	null
Update read:	null	T2 → T1
Final write:	T3	T1

→ VSR: s, s’ all initial read/update read/final write are the same

- H3: w1(x) r2(x) r2(y) r3(z) w1(z) c2 c3 c1
  - not RC: T2 read x from T1(wrote x), but T2 commit before T1 commit
  - not ACA: since not RC
  - not ST: since not ACA
  - CSR:

SG: acyclic graph → not VSR



$$\begin{tikzcd} & T1 \arrow[dr] & \\ T3 \arrow[ur] & & T2 \end{tikzcd}$$

  - not VSR:
- s = w1(x) r2(x) r2(y) r3(z) w1(z)

T1	T2	T3
w(x)		
	r(x)	
	r(y)	
		r(z)
w(z)		

	x	y	z
Initial Read	null	T2	T3
Update Read	T1 → T2	null	null
Final Write	T1	null	T1

- s’ = r2(x) r2(y) r3(z) w1(x) w1(z)

T1	T2	T3
	r(x)	
	r(y)	
		r(z)
w(x)		
w(z)		

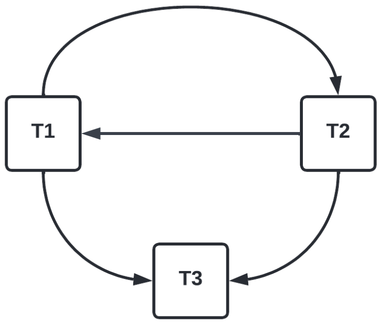
	x	y	z
Initial Read	T2	T2	T3
Update Read	null	null	null
Final Write	T1	null	T1

→ not VSR: s, s’ neither initial read nor update read are the same

- H4: w1(x) w2(x) r3(x) r1(x) c2 c3 c1
  - RC: T3 read x from T2(wrote x), T2 commit then T3 commit
  - not ACA: T3 read x from T2(wrote x), but T2 did not commit before T3 read x
  - not ST: since not ACA
  - not CSR

T1	T2	T3
w(x)		
	w(x)	
		r(x)
r(x)		

SG: → not CSR(cyclic graph)



- not VSR
  - s = w1(x) w2(x) r3(x) r1(x)

	x
Initial read:	null
Update read:	T2 → T3 T2 → T1
Final write	T2

  - s’ = w2(x) r3(x) w1(x) r1(x)

T1	T2	T3
	w(x)	
		r(x)
w(x)		
r(x)		

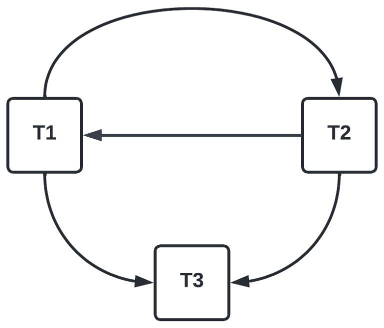
	x
Initial read:	null
Update read:	T2 → T3
Final write	T1

→ not VSR: s, s’: final write are not the same

- H5: w1(y) w2(y) w2(x) r1(x) w1(x) w3(y) c2 c1 c3
  - RC: T1 read x from T2(wrote x), T2 commit then T1 commit
  - not ACA: T1 read x from T2(wrote x), but T2 did not commit after T2 wrote x
  - not ST: since not ACA
  - not CSR

T1	T2	T3
w(y)		
	w(y)	
	w(x)	
r(x)		
w(x)		
		w(y)

SG: cyclic graph



  - VSR
    - s = w1(y) w2(y) w2(x) r1(x) w1(x) w3(y)

	x	y
Initial read:	null	null
Update read:	T2 → T1	null
Final write:	T1	T3

    - s'=w2(y)w2(x)w1(y)r1(x)w1(x)w3(y)

T1	T2	T3
	w(y)	
	w(x)	
w(y)		
r(x)		
w(x)		
		w(y)

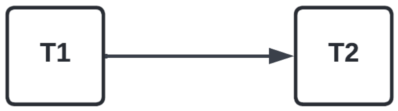
	x	y
Initial read:	null	null
Update read:	T2 → T1	null
Final write:	T1	T3

→ VSR: s, s' all initial read/update read/final write are the same

- H6: w1(x) r2(y) r1(y) r2(x) c1 c2
    - RC: T2 read y after T1 wrote y, T1 commit then T2 commit
    - not ACA: T2 read y after T1 wrote y, but T1 did not commit after T1 wrote y
    - not ST: since not ACA
    - CSR:

T1	T2
w(x)	
	r(y)
r(y)	
	r(x)

→ SG: acyclic graph

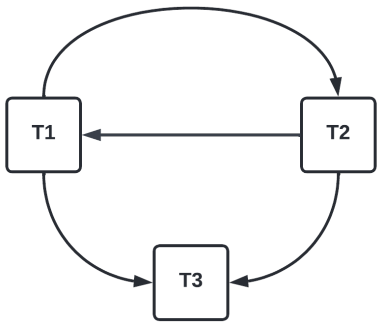


- VSR: since CSR

- H7: w1(x) w2(x) c2 r3(x) r1(x) c3 c1
    - ACA: T2 commit after T2 wrote x
    - RC: since ACA
    - not ST: T1 did not commit after T1 wrote x
    - not CSR

T1	T2	T3
w(x)		
	w(x)	
		r(x)
r(x)		

SG: cyclic graph



- not VSR
  - s:

	x
Initial read:	null
Update read:	T2 → T3 T2 → T1
Final write	T2

  - s’ = w2(x) r3(x) w1(x) r1(x)

T1	T2	T3
	w(x)	
		r(x)
w(x)		
r(x)		

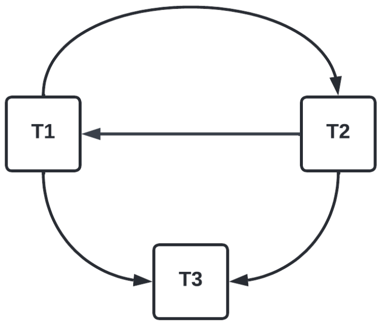
	x
Initial read:	null
Update read:	T2 → T3
Final write	T1

→ not VSR: s, s’ update read/final write are not the same

- H8: w1(y) w2(y) w2(x) c2 r1(x) w1(x) w3(y)
  - RC: T1 read x after T2 wrote x, T2 commit before T1 commit
  - ACA: T1 read x after T2 wrote x, T2 commit after T2 wrote x
  - ST: T2 wrote y after T1 wrote y, but T1 did not commit after T1 wrote y
  - not CSR

T1	T2	T3
w(y)		
	w(y)	
	w(x)	
r(x)		
w(x)		
		w(y)

SG: cyclic graph



- VSR
  - s:

	x	y
Initial read:	null	null
Update read:	T2 → T1	null
Final write:	T1	T3

  - s'= w2(y) w2(x) w1(y) r1(x) w1(x) w3(y)

T1	T2	T3
	w(y)	
	w(x)	
w(y)		
r(x)		
w(x)		
		w(y)

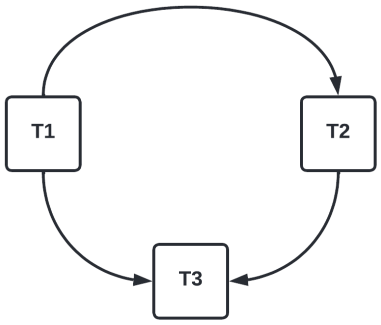
	x	y
Initial read:	null	null
Update read:	T2 → T1	null
Final write:	T1	T3

→ VSR: s, s': all initial read/update read /final write are the same

- H9: w1(y) w2(y) c2 r3(y) c1 c3
  - RC: T3 read y after T2 wrote y, T2 commit before T3 commit
  - ACA: T3 read y after T2 wrote y, T2 commit after T2 wrote y
  - not ST: T2 wrote y after T1 wrote y, but T1 did not commit after T1 wrote y
  - CSR

T1	T2	T3
w(y)		
	w(y)	
		r(y)

SG: acyclic graph

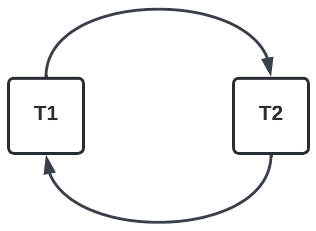


  - VSR: since CSR

- H10: r1(x) w2(x) r2(y) w1(y) c1 c2
  - RC: for x/y, there is no write-read operations
  - ACA: T2 read y after T1 wrote x, so T2 no need to commit after T2 wrote x
  - ST: for all read x/y, they all happen before write x/y
  - not CSR

T1	T2
r(x)	
	w(x)
	r(y)
w(y)	

SG: cyclic graph



  - not VSR
    - s:

	x	y
Initial read:	T1	T2
Update read:	null	null
Final write:	T2	T1

    - s' = r1(x) w1(y) w2(x) r2(y)

T1	T2
r(x)	
w(y)	
	w(x)
	r(y)

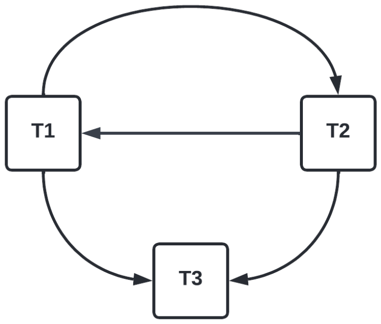
	x	y
Initial read:	T1	null
Update read:	null	T1 ⟶ T2
Final write:	T2	T1

⟶ not VSR: s, s': update read for y are not the same

- H11: r1(y)w2(y)c2w1(y)c1w3(y)c3
  - ST: every transaction commit after write operations
  - ACA: since ST
  - RC: since ACA
  - not CSR

T1	T2	T3
r(y)		
	w(y)	
w(y)		
		w(y)

SG: cyclic graph



  - VSR
    - s = r1(y) w2(y) c2 w1(y) c1 w3(y) c3

	y
Initial read:	T1
Update read:	null
Final write	T3

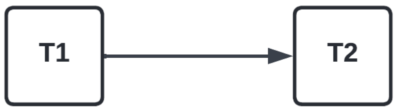
    - s’ = r1(y) w1(y) w2(y) w3(y)

T1	T2	T3
r(y)		
w(y)		
	w(y)	
		w(y)

	y
Initial read:	T1
Update read:	null
Final write	T3

→ VSR: s, s’ all initial read/update read/final write are the same

- H12: r1(x) r2(x) w2(x) c1 c2
    - RC: for x, there is no write-read operations
    - ACA: for x, there is no write-read operations
    - ST: for x, there are no write-read and no write-write operations
    - CSR: there is only x and only T1 → T2



    - VSR: since CSR