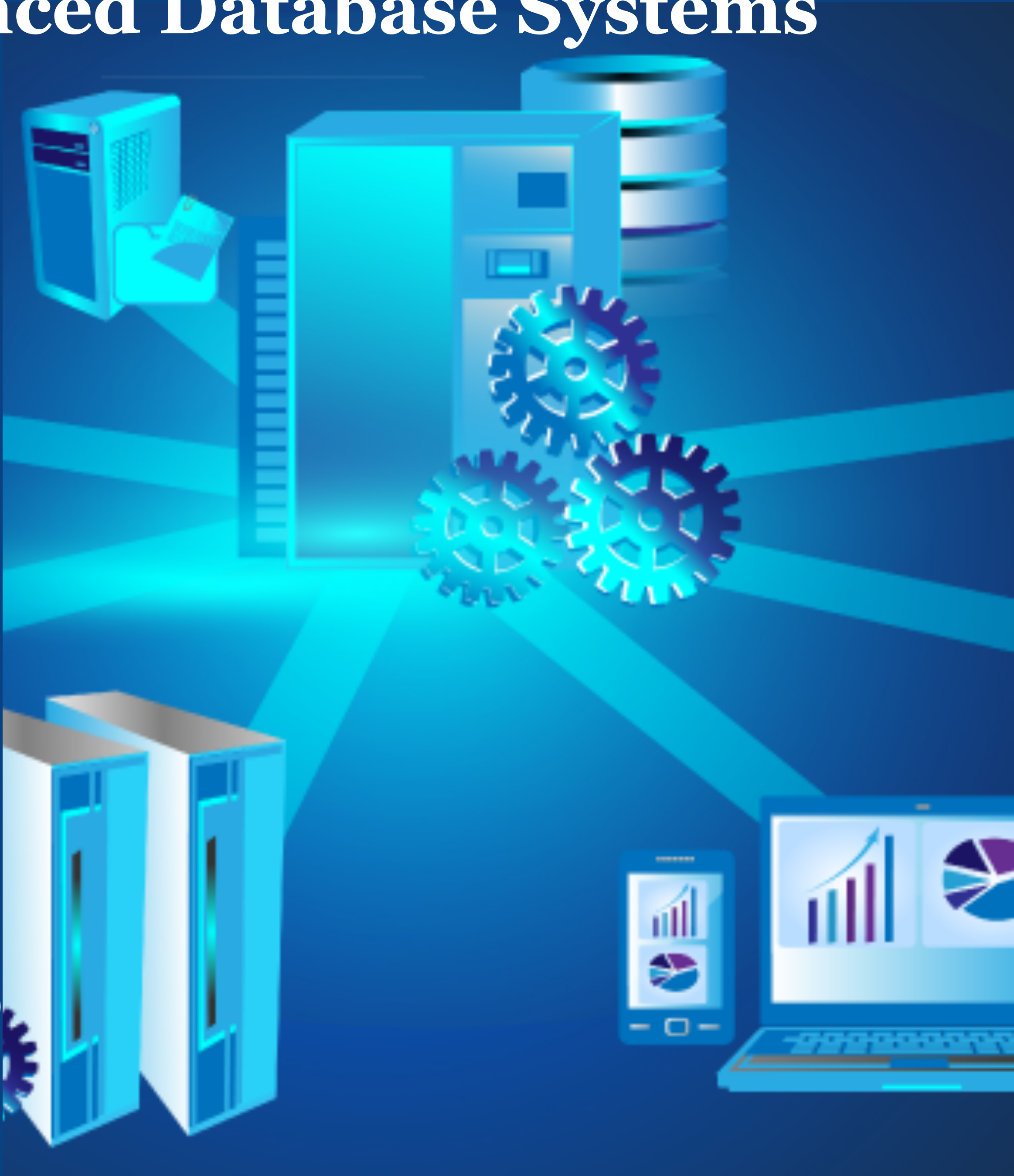
THE UNIVERSITY OF
MELBOURNE

COMP90050
Advanced Database Systems



Semester 2, 2024

Lecturer: Farhana Choudhury (PhD)

Live lecture – Week 6

Concurrency problem

Multiple concurrently running transactions may cause conflicts

- Still we try to allow concurrent runs as much as possible for a better performance, while avoiding conflicts as much as possible

What we need to know –

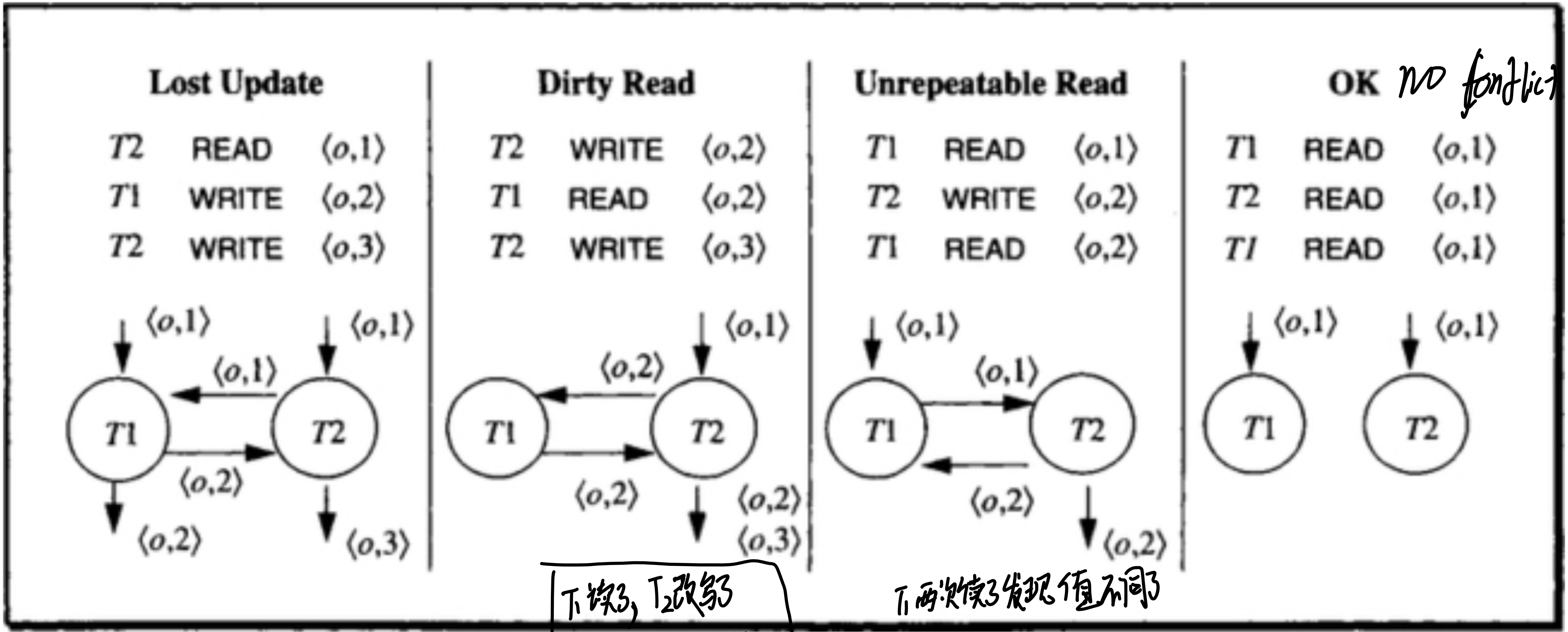
- What are the possible conflicts/dependencies

- Given a set of concurrent transactions, can we/DBMS determine whether there will be any conflict or not?

- Can conflicts be avoided (without making any change to the intended final output/final state of the database)?

Dependencies

When dependency graph has cycles then there is a violation of isolation and a possibility of inconsistency.



Lost Update

T2	READ	⟨o,1⟩
T1	WRITE	⟨o,2⟩
T2	WRITE	⟨o,3⟩

Dirty Read

T2	WRITE	⟨o,2⟩
T1	READ	⟨o,2⟩
T2	WRITE	⟨o,3⟩

Unrepeatable Read

T1	READ	⟨o,1⟩
T2	WRITE	⟨o,2⟩
T1	READ	⟨o,2⟩

OK no conflict

T1	READ	⟨o,1⟩
T2	READ	⟨o,1⟩
T1	READ	⟨o,1⟩

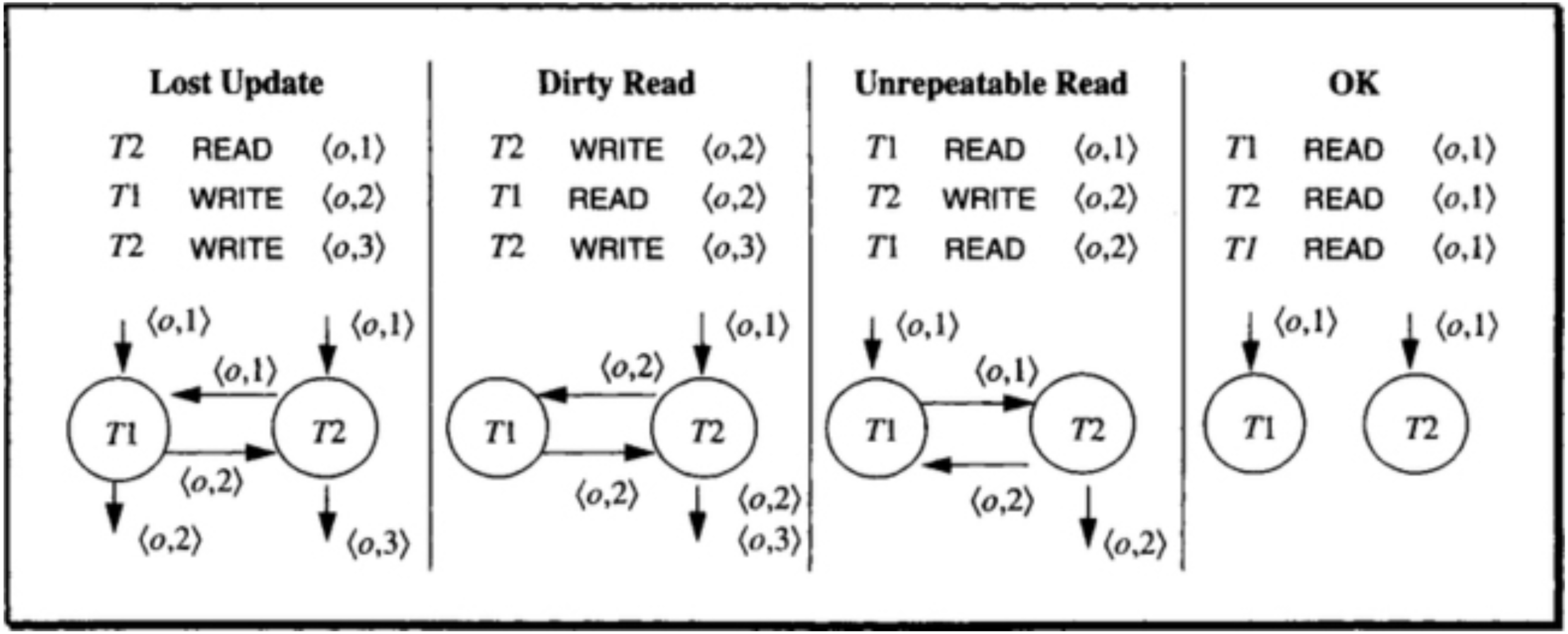
T_1读了, T_2改写了
T_1得一个脏数的值了
T_1获一个old

T_1两次读了发现值不同了

two transactions writing over the same object o, second write overwrite the first write (不管T_1写了什么,都被T_2重写了)

☆ while T_1 is running, another transaction change that. ☆
inconsistency

Some activities !



Lost Update

T2	READ	⟨*o*,1⟩
T1	WRITE	⟨*o*,2⟩
T2	WRITE	⟨*o*,3⟩

Dirty Read

T2	WRITE	⟨*o*,2⟩
T1	READ	⟨*o*,2⟩
T2	WRITE	⟨*o*,3⟩

Unrepeatable Read

T1	READ	⟨*o*,1⟩
T2	WRITE	⟨*o*,2⟩
T1	READ	⟨*o*,2⟩

OK

T1	READ	⟨*o*,1⟩
T2	READ	⟨*o*,1⟩
TI	READ	⟨*o*,1⟩

Time for a poll - Pollev.com/farhanachoud585 two concurrent transactions, is there any dependency among them?

T_1 - Read A, write B
T_2 - Read B, Read A, Read B

unRepeatable read: it might get two different values of B ← T_2 firstly read B, then T_1 write B, B got upated by T_1, then T_2 read B but different value.

当 depen

Dependency relations - equivalence

Given two different order of executions, can we have some insight on the final output/state of the database?

R: read O_i: object-i

W: write T_i: Transaction_i

☆☆ → 这样T1与T3无依赖!!

R

! 就多了一组 ☆

H1 = <(T1,R,O1), (T2, W, O5), (T1,W,O3), (T3,~~W~~R,O1), (T5,R,O3), (T3,W,O2), (T5,R,O4), (T4,R,O2), (T6,W,O4)>

<T1,O1,T3>, <T3,O1,T5>

DEP(H1) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

T1读 T3写 (T3 depends on T1 for object O1)

先 后

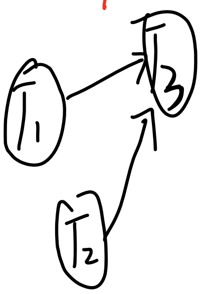
H2 = <(T1,R,O1), (T3,W,O1), (T3,W,O2),(T4,R,O2),(T1,W,O3), (T2, W, O5), (T5,R,O3), (T5,R,O4), (T6,W,O4)>

DEP(H2) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

DEP(H1) = DEP(H2)   equivalent history

(1) ☆ $\langle (T_1, R, O_1), (T_2, R, O_1), (T_3, W, O_1) \rangle$ 中间是读就都能换

$\{ \langle T_1, O_1, T_3 \rangle, \langle T_2, O_1, T_3 \rangle \}$ ☆



都依赖 T_3

在 DAG(H) 内的顺序可以随意换

triple

$\{ \langle T_2, O_1, T_3 \rangle \langle T_1, O_1, T_3 \rangle \}$ 也对

☆

中间是写就断了

(2) $\langle (T_1, W, O_1), (T_2, W, O_1), (T_3, W, O_1) \rangle$

$\{ \langle T_1, O_1, T_2 \rangle, \langle T_2, O_1, T_3 \rangle \}$ $T_1 \to T_2 \to T_3$ (中间有写就和前面的没关了)..

(3) $\langle (T_1, R, O_1), (T_2, R, O_2) \rangle$

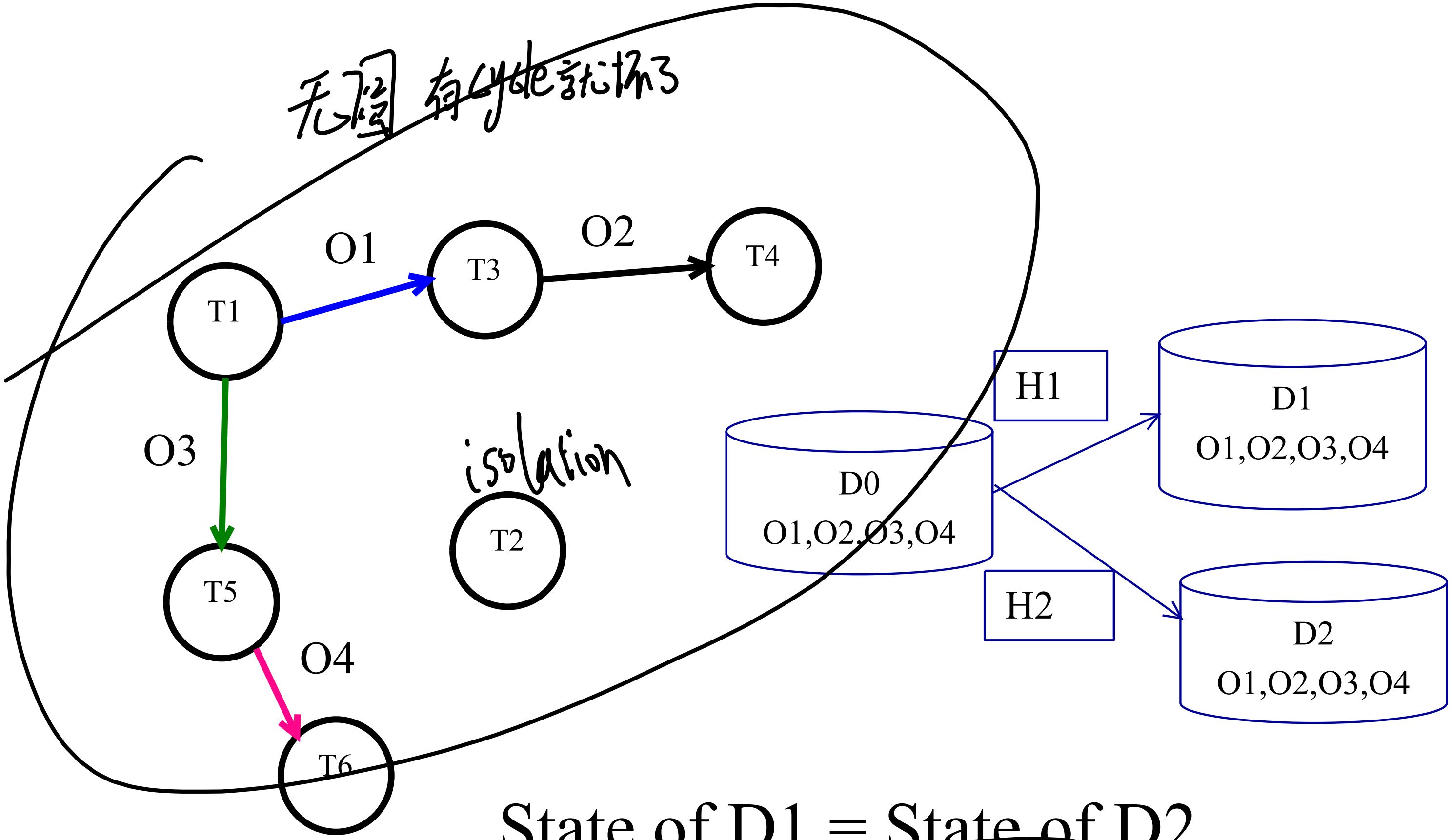
No dependency.

Dependency relations - equivalence

implication of equivalent history: After the executions done based on H1 or H2, final state of the database will be same

DEP(H1) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

DEP(H2) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }



State of D1 = State of D2

Dependency relations

Goal: Can we run transactions concurrently, but still have the same final output/state of the database as if the transactions are serially executed? (one start and complete) ⇒ Then the other one start and complete.

no ~~the~~ concurrency problem ⇒ no isolation problem

run concurrently but get the same result with serial execution.

在图

Isolated history

Given a history, how can we determine whether it is 'isolated history' (that is, equivalent to a serial history)? – We try to find a cycle.

Presence of a wormhole transaction in a history implies it is not isolated. A transaction T' is called a wormhole transaction if

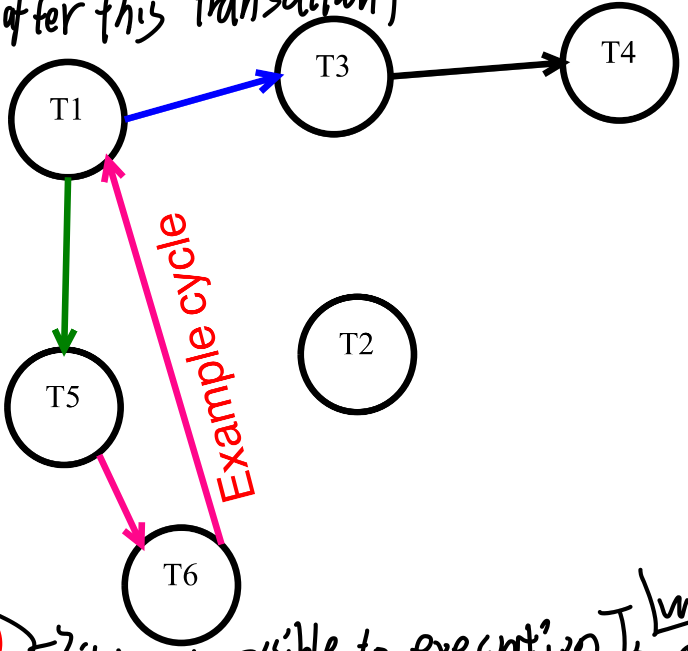
T' is wormhole Transaction ← $T' \in Before(T) \cap After(T)$ → T' comes before and also comes after this transaction T → cycle in the graph

This implies there is a cycle

E.g. After(T1) = {T5,T6, T3, T4} before (T1) = { }

After(T3) = {T4}

Before(T3) = {T1}



T1 will be both in Before(T6) and After(T6) → it is not possible to execution T6 without any conflict

Isolation Concepts ...

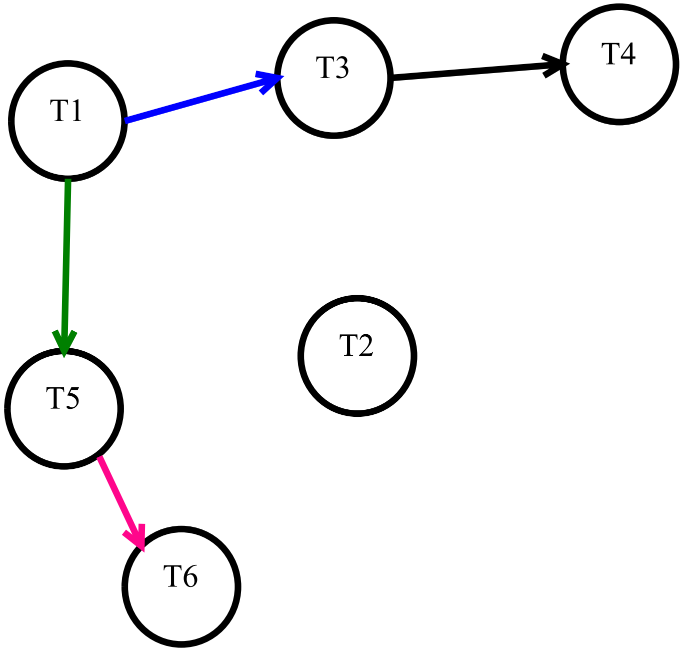
A history is serial if it runs one transaction at a time sequentially, or equivalent to a serial history. (no conflict)

A serial history is an **isolated** history.

Wormhole theorem: A history is isolated if and only if it has no wormholes.

Give an example order of execution avoiding conflicts (without making any change to the intended final output/final state of the database)?

Hint: If T3 runs before T1, that will change the output



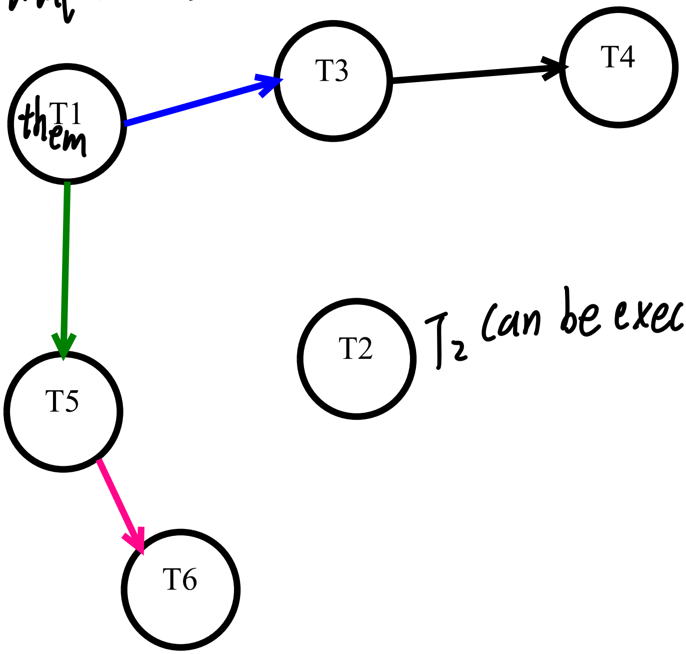
Some activities !

Time for a poll - Pollev.com/farhanachoud585

the order of execution for these transaction that same as a serial execution order.

Should have all the transaction in them

complete



position of

T_2 can be executed at any order

valid :→ T_1 T_3 T_4 T_5 T_6 T_2

① 一定要在前，

③ 随意位置

② 同 {T_3 T_4} {T_5 T_6} 各自顺序要保证

valid : → T_1 T_2 T_3 T_5 T_4 T_6 也是对的

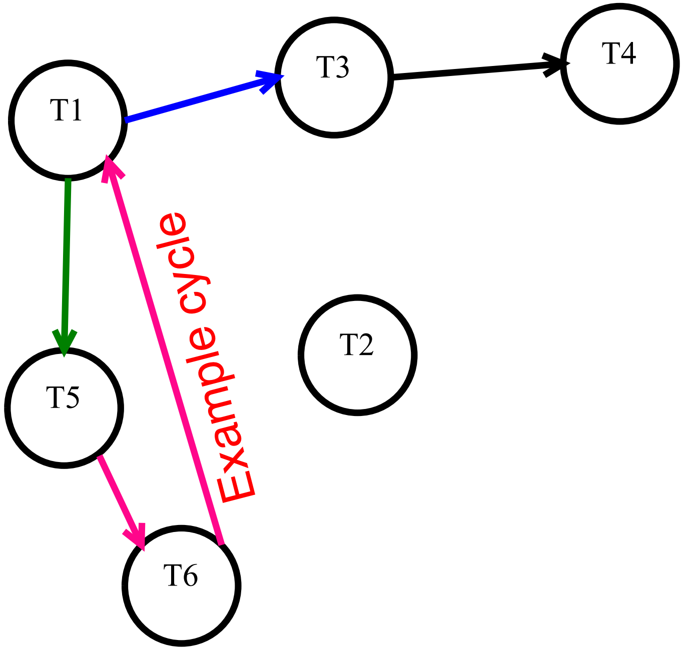
Isolated history

How can we ensure there’s no wormhole?

Solution: Through appropriate locks.

But we also need to carefully think about **types of locks based on the application and requirement.**

什么应用适合什么锁？



Degrees of Isolation

→ repeatable read

读不是 two-phased

Degree 3

Slock(A)
Xlock(B)
Xlock(C)
Read(A)
Write(B)
Read(A)
Write(C)
Unlock(A)
Unlock(B)
Unlock(C)

Phase1
all locks

all unlock.
Phase2

Degree 2

Slock(A)
Read(A)
Unlock(A)
Xlock(C)
Xlock(B)
Write(B)
Slock(A)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)

Phase1
With
X-locks

Phase2
With
X-locks

Degree 1

Read(A)
Xlock(C)
Xlock(B)
Write(B)
Read(A)
Write(C)
Unlock(B)
Unlock(C)

Phase1

Phase2

Degree 0

no locks on ready

Read(A)
Xlock(B)
Write(B)
Unlock(B)
Read(A)
Xlock(C)
Write(C)
Unlock(C)

Protecting
writes

after unlock another transaction will be able to take a lock on B

Protecting
writes

and it will be able to change the value of that object.

More restrictive locks mean less/no conflict,
but the overall transaction throughput gets slower

Since those objects are unlocked, other transaction will be able to change them. (other transaction will be able to take lock on object A when this

transaction is still running → Then there's
some inconsistency happening

Isolation Concepts ...

In SQL2 one can declare isolation level as follows:

SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED |

READ COMMITTED| REPEATABLE READ | SERIALIZABLE}

Slight difference with the four degrees of isolation

- SERIALIZABLE – degree 3
- REPEATABLE READ – like degree 3, but other transactions can insert new rows
- READ COMMITTED – Degree 2*
- READ UNCOMMITTED – Degree 0

*Options can also be paired with SNAPSHOT on/off

Source: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15

Summary

Multiple concurrently running transactions may cause conflicts

- Different types of conflicts

- Avoiding conflicts – using locks

Later we will see –

- More types of locks

- Relaxed isolation – for better performance e.g addiy somethiy into shoppiy cart ⇒ check out and payment ⇒ failed ☹

sorry item unavailable.

but it did allow to add this item to cart. then we see the item is unavailable until checkout → comes from different isolation level.