THE UNIVERSITY OF MELBOURNE

COMP90050 Advanced Database Systems

Winter Semester, 2023

Lecturer: Farhana Choudhury (PhD)

Week 3 part 4

What we have seen in previous lecture

- Problems may arise from multiple concurrent transactions
- Concurrency control is needed
  - Take <u>exclusive access</u> to shared resources to handle concurrency problems

In this lecture we will see the concurrency control more formally and in more details

Isolation Concepts – I in ACID

Isolation ensures that concurrent transactions leaves the database in the same state as if the transactions were executed separately.

Isolation guarantees consistency, provided each transaction itself is consistent.

Isolation – expected output example

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently.

a) counter == 110	b) counter == 130	c) counter == 140;
Sequence of actions	Sequence of actions	Sequence of actions
T1: Reads counter == 100	T1: Reads counter == 100	T1: Reads counter == 100
T2: Reads counter == 100	T2: Reads counter == 100	T1: Writes counter == 100+10
T2: Writes counter == 100+30	T1: Writes counter == 100+10	T2: Reads counter == 110
T1: Writes counter == 100+10	T2: Writes counter == 100+30	T2: Writes counter == 110+30

Alternatively, T2 executing before T1 will also have the same state

Isolation Concepts ...

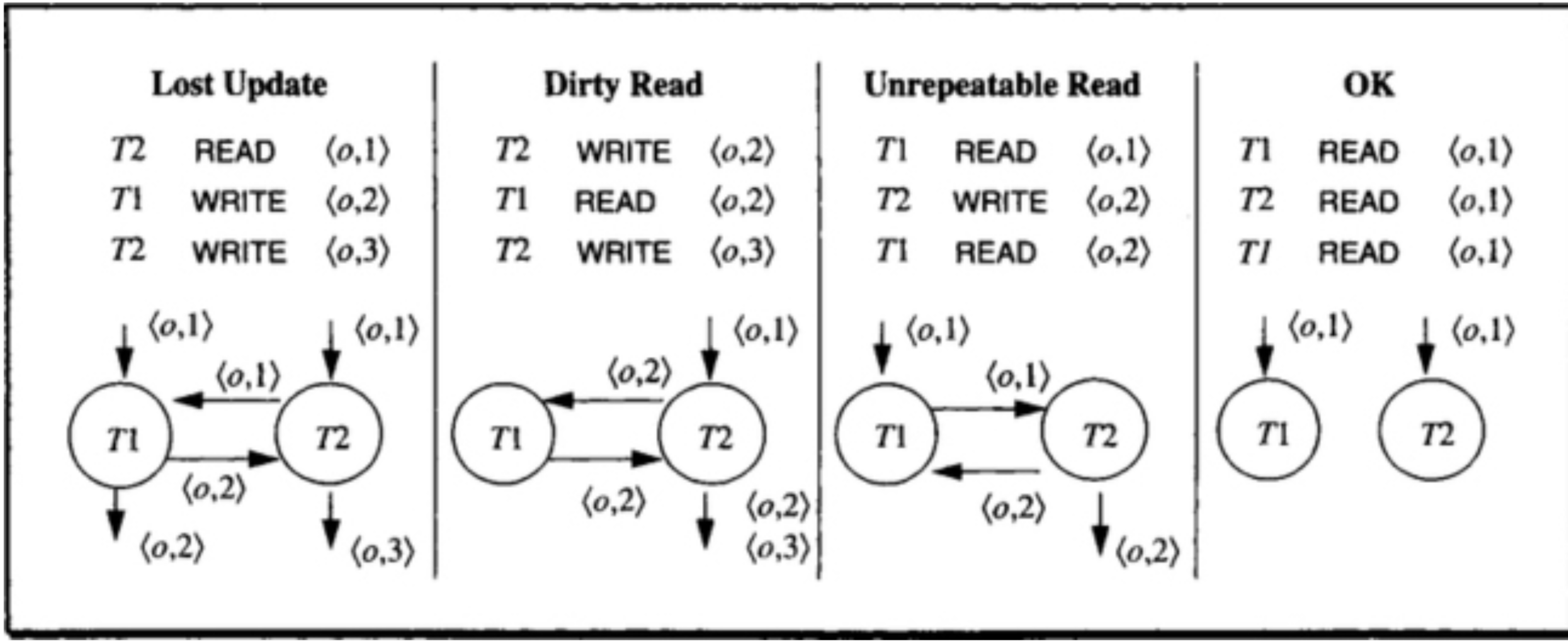
We can achieve isolation by sequentially processing each transaction - generally not efficient and provides poor response times.

We need to run transactions concurrently with the following goals:

- concurrent execution should not cause application programs (transactions) to malfunction.
- Concurrent execution should not have lower throughput or bad response times than serial execution.

To achieve isolation we need to understand dependency of operations

Possible dependencies



How can we find the dependencies?

Given a set of transactions, how can we determine which transaction depends on which other transaction?

Dependency model

I_i : set of inputs (objects that are read) of a transaction T_i

O_i : set of outputs (objects that are modified) of a transaction T_i

Note O_j *and* I_j are not necessarily disjoint that is $O_j \cap I_j \neq \emptyset$

Given a set of transactions τ , Transaction T_j has no dependency on any transaction T_i in τ if -

$$O_i \bigcap (I_j \bigcup O_j) = empty\ for\ all\ i \neq j$$

This approach cannot be planed ahead as in many situation inputs and outputs may be state dependant/not known in prior.

$$O_i \bigcap (I_j \bigcup O_j) = \textit{empty for all } i \neq j$$

	T_1	T_2	T_3	T_4
Trans In ∪ Out ($I_i \cup O_i$)	$O_i \bigcap (I_1 \cup O_1)$	$O_i \bigcap (I_2 \cup O_2)$	$O_i \bigcap (I_3 \cup O_3)$	$O_i \bigcap (I_4 \cup O_4)$
T_1	O_1	$\emptyset$	$\emptyset$	$\emptyset$
T_2	$\emptyset$	O_2	$\emptyset$	$\emptyset$
T_3	$\emptyset$	$\emptyset$	O_3	$\emptyset$
T_4	$\emptyset$	$\emptyset$	$\emptyset$	O_4

If the inputs and outputs of the concurrent transactions are not disjoint, the following dependencies can occurs –

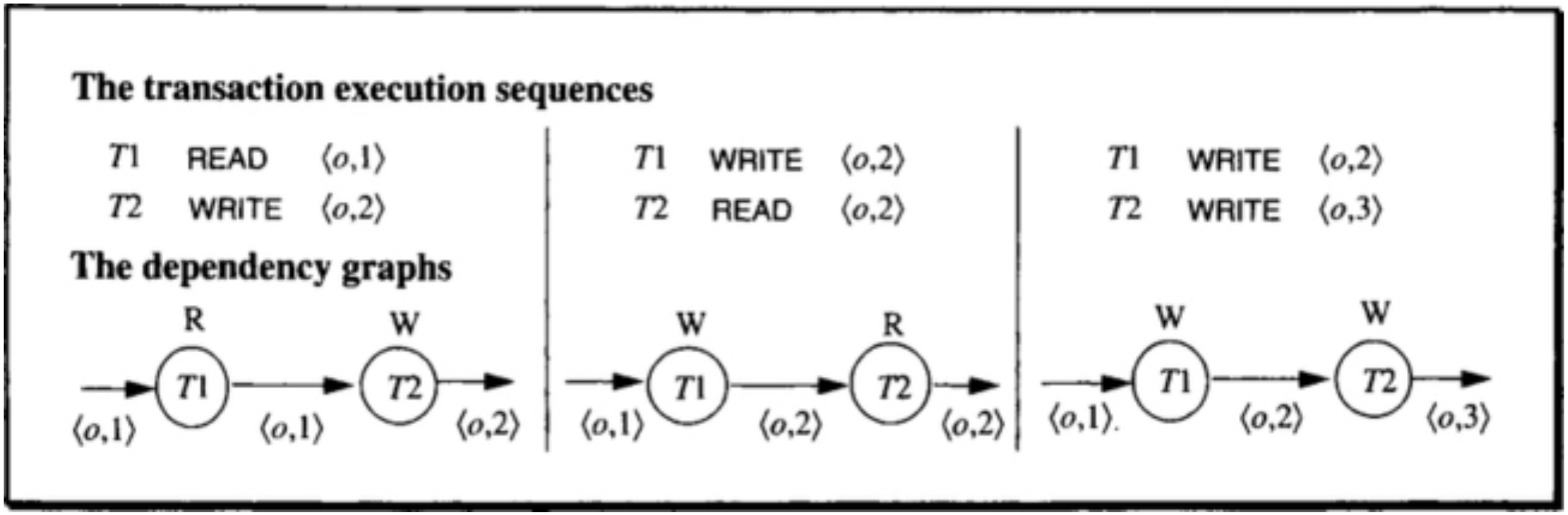
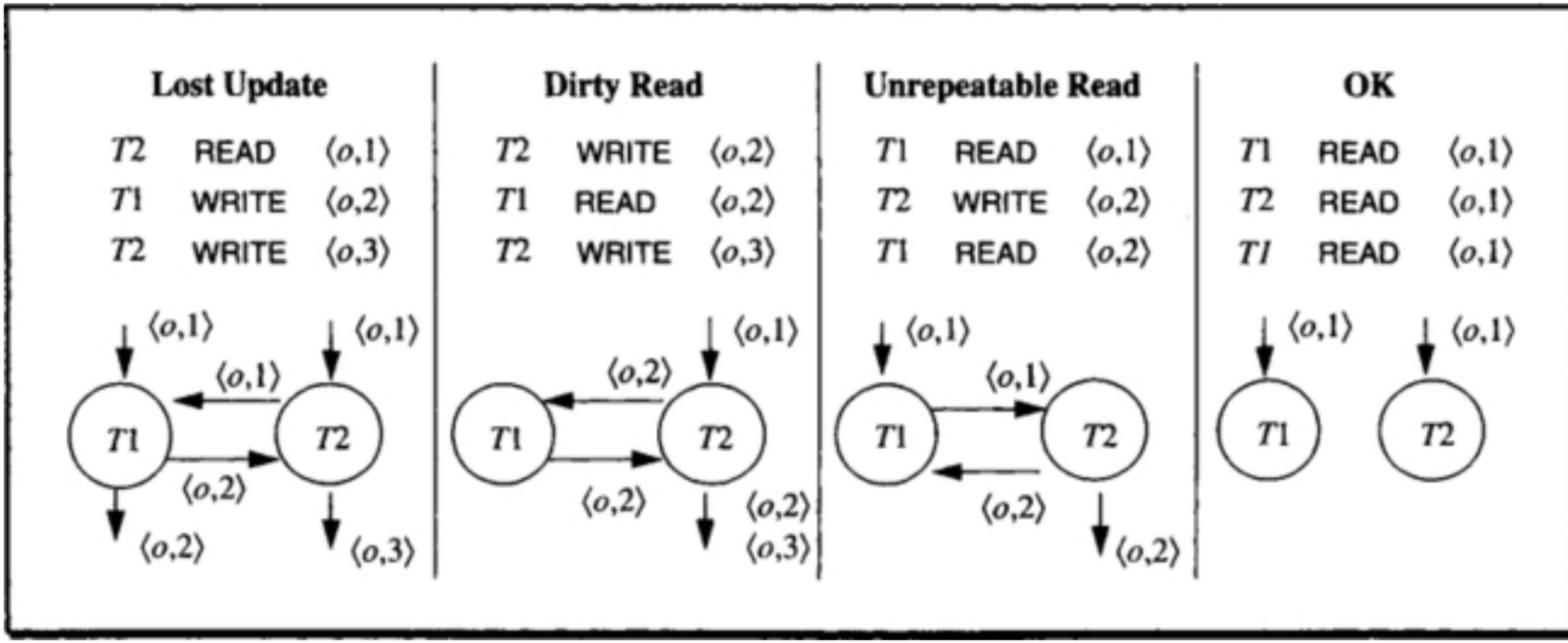


Fig 7.1 in the main reference book

Read-Read dependency do not affect isolation

Dependencies

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



% T2 did not see the update of T1

What if T2 aborts? T1's read will be invalid

The value of o changes by another transaction T2 while T1 is still running

Formal definition of dependency

Let H is a history sequence of tuples of the form (T, action, object).

Let T1 and T2 are transactions in H. If T1 performs an action on an object O, then T2 performs an action on the same O, and there is no write action in between by another transaction on O – then T2 depends on T1.

Formally, the dependency of T2 on T1 (T1, O, T2) exists in history H if there are indexes i and j such that i < j, H[i] involves action a1 on O by T1, (i.e., H[i] = (T1,a1,O)) and H[j] involves action a2 on O by T2 (i.e., H(j) = (T2, a2,O)) and there are no other H[k] = (T',WRITE,O) for i < k < j

Dependency graph : Transactions are nodes, and object labels the edges from the node Ti to Tj if (Ti, O, Tj) is in DEP(H).

Dependency relations

We focus on the dependency in three scenarios

- a1 = WRITE & a2 = WRITE;
- a1 = WRITE & a2 = READ;
- a1 = READ & a2 = WRITE (dependency as T1 may read again after a2).

Dependency relations - equivalence

DEP(H) = { (Ti, O, Tj) | Tj depends on Ti }.

Given two histories H1 and H2 contain the same tuples, H1 and H2 are equivalent if DEP(H1) = DEP(H2)

This implies that a given database will end up in exactly the same final state by executing either of the sequence of operations in H1 or H2

E.g.,

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

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

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)

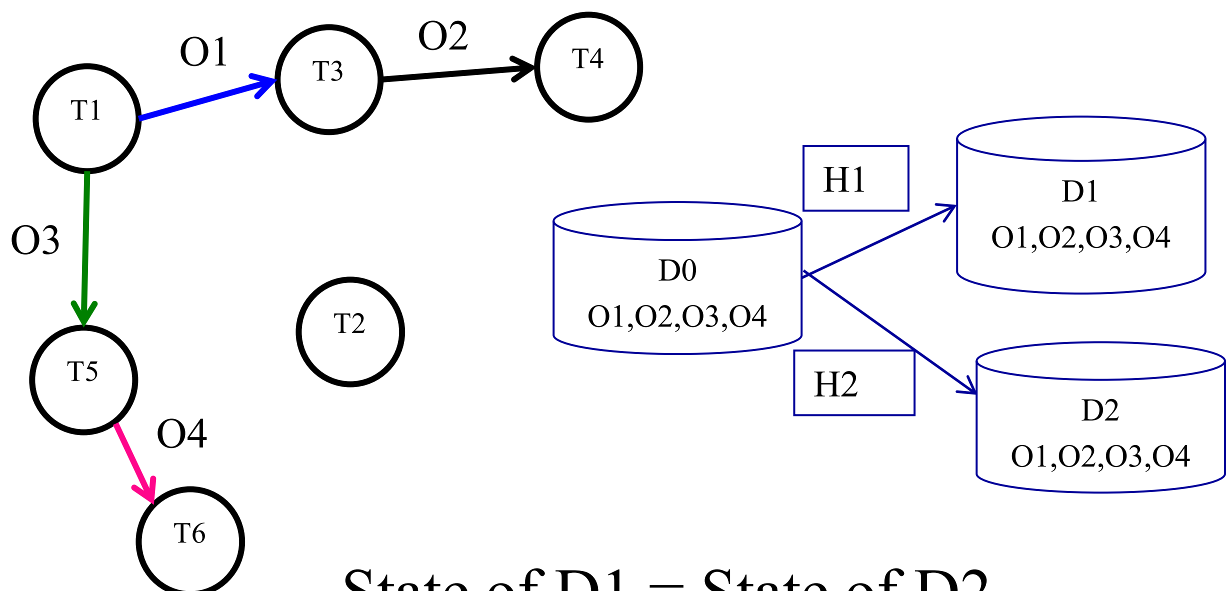
Dependency relations - equivalence

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

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(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

Isolated history

A history is said to be isolated if it is equivalent to a serial history (as if all transactions are executed serially/sequentially)

A serial history is history that is resulted as a consequence of running transactions sequentially one by one. N transactions can result in a maximum of N! serial histories.

If T1 precedes T2,

it is written as T1 << T2.

Before(T) = {T' | T' << T}

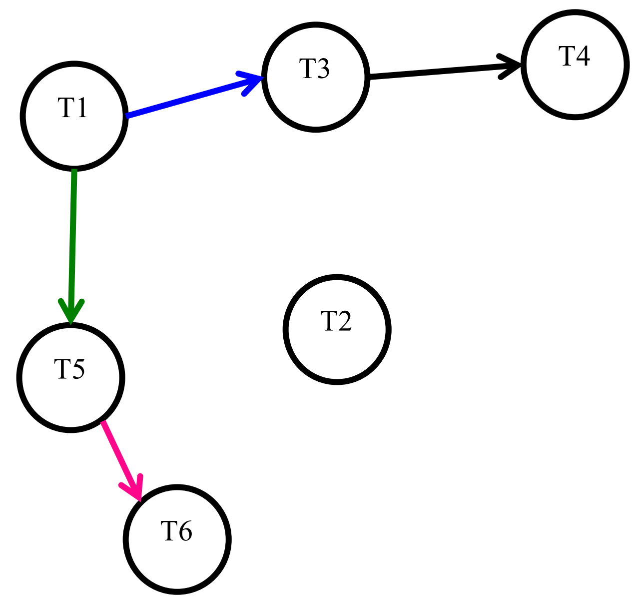
After(T) = {T'| T << T'}

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

After(T3) = {T4}

After (T5) = {T6}

After(T3) = {T4}



Isolation Concepts ...

A transaction T' is called a wormhole transaction if

$$T' \in Before(T) \bigcap After(T)$$

That is T << T' << T. This implies there is a cycle in the dependency graph of the history. Presence of a wormhole transaction implies it is not isolated (=> not a serial schedule).

A history is serial if it runs one transaction at a time sequentially, or equivalent to a serial history.
A serial history is an **isolated** history.
Wormhole theorem: A history is isolated if and only if it has no wormholes.

We will now introduce a new type of lock -

SLOCK (shared lock) that allows other transactions to read, but not write/modify the shared resource

The wormhole transaction concept will be useful in a later topic!

To grant lock or not to…

A lock on an object should not be granted to a transaction while that object is locked by another transaction in an **incompatible mode**.

Lock Compatibility Matrix

	Mode of Lock		
Current Mode	Free	Shared	Exclusive
Shared request (SLOCK) Used to block others writing/modifying	Compatible Request granted immediately Changes Mode from Free to Shared	Compatible Request granted immediately Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive
Exclusive request (XLOCK) Used to block others reading or writing/modifying	Compatible Request granted immediately Changes Mode from Free to Exclusive	Conflict Request delayed until the state becomes compatible Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive

When to use what type of lock

Actions in Transactions are: READ, WRITE, XLOCK, SLOCK, UNLOCK, BEGIN, COMMIT, ROLLBACK

T1
BEGIN
SLOCK A
XlOCK B
READ A
WRITE B
COMMIT
UNLOCK A
UNLOCK B
END

T2
BEGIN
SLOCK A
XlOCK B
READ A
WRITE B
ROLLBACK
UNLOCK A
UNLOCK B
END

Isolation Concepts ...

BEGIN, END , SLOCK, XLOCK can be ignored as they can be automatically inserted in terms of the corresponding operations

E.g. if a transaction ends with a COMMIT, it is replaced with:

{UNLOCK A if SLOCK A or XLOCK A appears in T for any object A}. (That is to simply release all locks)

Similarly ROLLBACK can be replaced by

{WRITE(UNDO) A if WRITE A appears in T for any object A}

{ UNLOCK A if SLOCK A or XLOCK A appears in T for any object A}.

Isolation Concepts ...

We can replace previous transaction sequences by:

T1
READ A
WRITE B
COMMIT

↔

T1
BEGIN
SLOCK A
READ A
XLOCK B
WRITE B
UNLOCK A
UNLOCK B
END

T2
READ A
WRITE B
ROLLBACK

↔

T2
BEGIN
SLOCK A
READ A
XLOCK B
WRITE B
WRITE (UNDO) B
UNLOCK A
UNLOCK B
END

Isolation Concepts ...

Well-formed transactions: A transaction is well formed if all READ, WRITE and UNLOCK operations are covered by appropriate LOCK operations

Two phase transactions: A transaction is two phased if all LOCK operations precede all its UNLOCK operations.

Isolation Theorems

Summary:

A transactions is a sequence of READ, WRITE, SLOCK, XLOCK actions on objects ending with COMMIT or ROLLBACK.

A transaction is well formed if each READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation.

A history is legal if does not grant conflicting grants.

A transaction is two phase if its all lock operations precede its unlock operations.

Isolation Theorems

Locking theorem: If all transactions are well formed (READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation) and two-phased (locks are released only at the end), then any legal (does not grant conflicting grants) history will be isolated.

Locking theorem (Converse): If a transaction is not well formed or is not two-phase, then it is possible to write another transaction such that it is a wormhole.

Rollback theorem: An update transaction that does an UNLOCK and then does a ROLLBACK is not two phase.

Degrees of Isolation

Degree 3: A Three degree isolated Transaction has no lost updates, and has repeatable reads. This is “true” isolation.

Lock protocol is two phase and well formed.

It is sensitive to the following conflicts:

write->write; write ->read; read->write

Degree 2: A Two degree isolated transaction has no lost updates and no dirty reads.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to Reads and writes. (May have Non repeatable reads.)

It is sensitive to the following conflicts:

write->write; write ->read;

Slock(A)
Xlock(B)
Xlock(C)
Phase1
Read(A)
Write(B)
Read(A)
Write(C)
Unlock(A)
Unlock(B)
Unlock(C)
Phase2

Degree 2

Slock(A)
Read(A)
Unlock(A)
Xlock(C)
Xlock(B)
Phase1 With X-locks
Write(B)
Slock(A)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)
Phase2 With X-locks

Degree 1: A One degree isolation has no lost updates.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to writes.

It is sensitive the following conflicts:

write->write;

Degree 0 : A Zero degree transaction does not overwrite another transactions dirty data if the other transaction is at least One degree.

Lock protocol is well-formed with respect to writes.

It ignores all conflicts.

Degree 1

Read(A)
Xlock(C)
Xlock(B) — Phase1
Write(B)
Read(A)
Write(C)
Unlock(B)
Unlock(C) — Phase2

Degree 0

Read(A)
Xlock(B)
Write(B)
Unlock(B) — Protecting writes
Read(A)
Xlock(C)
Write(C)
Unlock(C) — Protecting writes