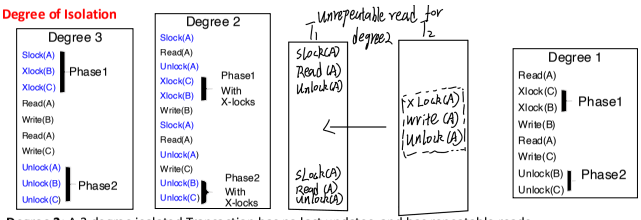
Degree of Isolation



Degree 3: A 3 degree isolated Transaction has no lost updates, and has repeatable reads.
Lock protocol is two phase and well formed. Sensitive(可以搞定):write->write; write ->read; read->write

Degree 2: A 2 degree isolated transaction has no lost updates and no dirty reads. (**only slock(x) is not two-phase**)
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; (两个读锁之间，object不会被lock，其他trans可以修改when this trans is still running→ some inconsistency happening)

Degree 1: A 1 degree isolation has no lost updates. Lock protocol is two phase with respect to exclusive locks and well formed with respect to writes (不管读了，写全管(well-formed+Two phase transactions)). It is sensitive the following conflicts: write->write;

Granular locks

The following transactions are issued in a system at the same time. Answer for both scenarios.

(i) Scenario 1: When the value of the variable some_input is 3, which of the following transactions can run concurrently from the beginning till commit (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give explanation for the delayed transactions.

(ii) Scenario 2: When the value of the variable some_input is 1, which of the following transactions can run concurrently from the beginning till commit (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give explanation for the delayed transactions.

Compatibility Mode of Granular Locks

Current / Request	None	IS	IX	S	SIX	U	X
	+/- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

T1	T2	T3
Lock (S,A)	Lock (IS,A)	Lock (IX,A)
Read A	If(some_input == 3){	If(some_input == 3){
Unlock A	Lock(S,A)	Lock (X,A)
	Read A	Write A
	}	}
	Unlock A	Unlock A

A compatibility matrix is as follows -

(i) T1&T2: Lock (S,A) and Lock (IS,A) are compatible with each other; Lock (S,A) of T1 and Lock (S,A) of T2 are compatible with each other as well. Therefore, T1 and T2 can run concurrently. **T2&T3**: Lock(IS,A) and Lock(Ix, A) are compatible with each other. However, lock(S,A) and Lock (X,A) conflict with each other, hence T1 and T3 cannot fully run concurrently from the beginning till commit. **T1&T3**: Lock(S,A) and Lock(IX, A) conflict with each other, hence they cannot run concurrently. Overall, T1&T2 can run concurrently and then in that case T3 will be delayed.

(ii) When some_input is 1, for T2 and T3 we will not execute any further operations but just unlock A directly. **T1&T2**: Lock (S,A) and Lock (IS, A) are compatible with each other, hence T1 and T2 can run concurrently. **T2&T3:** Lock(IS,A) and Lock(IX, A) are compatible with each other, hence T2 and T3 can run concurrently. **T1&T3:** Lock(IX,A) and Lock(S) are conflict with each other. Overall, if T1&T2 run concurrently, then T3 would be delayed; while if T2&T3 run concurrently, then T1 would be delayed.

Locks

Lock Compatibility Matrix

Current	Free	Shared (Slock)	Exclusive (Xlock)
Request - Slock	√	√	×
Request - Xlock	√	×	×

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}.

Well-formed transactions: A transaction is well formed if all READ, WRITE and UNLOCK operations are covered by appropriate LOCK operations (只有lock,没有unlock,也算Well-formed)
Two phase transactions: A transaction is two phased if all LOCK operations precede all its UNLOCK operations.

1.Locking theorem: If all transactions are well formed and two-phased, then any legal (does not grant conflicting grants) history will be isolated. 2.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. 3.Rollback theorem: An update transaction that does an UNLOCK and then does a ROLLBACK is not two phase.

Not a two-phase:	没有unlock也算well-formed
Lock(A)	Slock(A)
Update(A)	Read(A)
Unlock(A)	Lock(A)
Rollback	Write(A)

Simultaneously, And locks can be held on for the duration they are needed, allowing multiple transactions running concurrently.
Cons: Deadlock can occur, since locks are required incrementally, cyclic waiting dependencies (把死锁那个环画上去) may lead to deadlocks, and additional mechanisms might be needed to detect and resolve deadlocks.
For all locks at the beginning: Pros:1.simplistic management and deadlock can be prevented. Cons: Trans may have to wait longer before starting, as they need all locks to be available, potentially leading to increased contention and idle times.

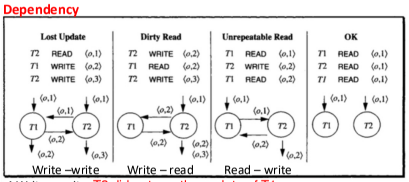
Granular lock 表越往下级别越高

Granular lock rule:
1.Acquire locks from root to leaf.(从上往下) Release locks from leaf to root.(从下往上)
2.To acquire an S mode or IS mode lock on a non-root node, **one** parent must be held in IS mode or higher (one of {IS,IX,S,SIX,U,X}). (在下面的节点请求S或IS，有一个父母必须大于等于这个级别)
3. To acquire an X, U, SIX, or IX mode lock on a non-root node, **all** parents must be held in IX mode or higher (one of {IX,SIX,U,X}). (在下面的节点请求X, U, SIX, or IX，所有父母必须大于等于这个级别)

Why update lock is necessary?
在这种情况下, T1和T2都在等待彼此释放A的slock，但是双方都hold这个锁不会释放，所以deadlock。

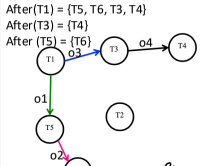
T1:	T2:
SLock A	SLock A
Read A	Read A
If (A== 3)	If (A==3)
{	{
% Upgrading Slock to Xlock	% Upgrading Slock to Xlock
Xlock A	Xlock A
Write A	Write A
}	}
Unlock A	Unlock A

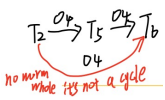
Dependency



1.Write – write: T2 did not see the update of T1
2. Write – read: 改的过程中有人读了. What if T2 aborts? T1's read will be invalid
3. Read – write: 读的过程中被改了. The value of o changes by another transaction T2 while T1 is still running.

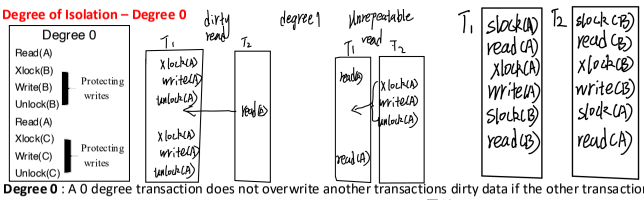
有环才是circle $T' \in Before(T) \cap After(T)$
After(T1) = {T5, T6, T3, T4}
After(T3) = {T4}
After (T5) = {T6}





依赖关系
1.按object的顺序写; 2.只要中间没有写就都依赖，中间有写两边就断; 3. 自己不依赖自己
e.g., 中间是**读全依赖**: <(T1,R,O1), (T2,R,O1), (T3,W,O1)> =>H1={<T1, O1,T3>,< T2, O1,T3 >}
e.g., 中间是**写依赖就断**: <(T1,W,O1), (T2,W,O1), (T3,W,O1)> => H2={<T1, O1,T2>,< T2, O1,T3 >}
Implication of equivalent history: After the execution done based on H1 or H2, the final state of DB will be the same.
DEP(H1)≠DEP(H2)
Wormhole theorem: A history is isolated **if and only if** it has no wormholes. (isolated history=serial history=no wormholes)

Degree of Isolation – Degree 0



Degree 0 : A 0 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. (只锁write) It ignores all conflicts.

Q: What is the difference between Two-phase locking and a locking strategy where all locks are taken at the beginning of a transaction and released at the end of a transaction? Compare the two approaches in a few paragraphs, i.e. their benefits and disadvantages. Would there be any concurrency if the second method mentioned above is used. Briefly explain.

Ans: (right-top is two-phase case) It is possible appears deadlock case when T1 is holding the xlock on A but also attempts to requests a lock on (B) and read(B), while T2 is still hold xlock on B but also wants to request a lock on (A) and read(A). So a deadlock occurs because T1 is waiting for B (held by T2 now), and T2 is waiting for A (held by T1 now). In contrast, using the locking strategy mentioned above: Bothe T1 and T2 will request locks on both A and B at the beginning. If both locks are available, then T1 proceeds firstly, then T2 will wait till all resources get released. Since all locks are required before any operation, deadlocks can be avoided, but concurrency is reduced because transactions may have to wait longer before starting. Pros for 2PL: Higher concurrency, transaction require locks as needed, allowing other transactions to access non-conflicting data

update lock

```
T1:
Ulock A
Read A
If (A== 3){
        Xlock A
        Write A
}
Unlock A
```

Solution B

```
T2:
Ulock A
Read A
If (A==3){
    Xlock A
    Write A
}
Unlock A
```

```
T3:
SLock A
Read A
Unlock A
```

Why not use Xlock but use Ulock? When an update lock is granted, it doesn't allow any other type of updating or writing locks. Update locking is blokcing another trans (T2) from acquiring this ulock or xlock on the same object A. However, if a shared type of lock is granted first then an update type of request can be granted concurrently. For T3 gets shared lock firstly, then at the same time T1 or T2 can still be granted an update lock on the same object A, which allows more transaction run concurrently and improves the level of concurrency of the whole system.
However, if T1 later attempts to acquire an X-lock (if the condition becomes true), a conflict will occur, and T1 will need to wait. Therefore, T1 and T3 cannot fully run concurrently, but u lock is still good compare with solution A.

update lock Solution A add unlock in the middle and check if A is changed after Xlock

```
T1:
SLock A
Read A
If (A == 3){
% Release lock and
try in Xlock mode
  Unlock(A)
  Xlock A
  Read A
  if(A == 3){
        Write A
  }
}
Unlock A
```

```
T2:
SLock A
Read A
If (A == 3){
% Release lock and
    try in Xlock mode
    Unlock(A)
    Xlock A
    Read A
    if(A == 3){
    Write A
    }
}
Unlock A
```

```
T3:
SLock A
Read A
Unlock A
```

第二个判断是否与之前的值一致

Unlock(A) is added at the middle, ensuring no deadlock if other transaction is waiting for it. Then that transaction will be allowed to acquire that lock, although it ensures deadlock not arise , but some potential inconsistency may arise. A can be modified by another trans, although next lock is taken an exclusive lock to check the value.
Cons: 1.The first read and the second read may not give the same result, causing unrepeatable read. A can be changed by. Another transaction while T1 is being executed.2. This is not a Two-phased locking anymore as unlock happens before lock, hence it is possible to generate some wormhole transactions, causing inconsistency.

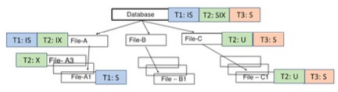
Granular locks

Given the hierarchy of database objects and the corresponding granular locks in the following picture, which transactions can run concurrently from the beginning till the end (that is, all operations and locks are compatible to run concurrently with another one), and which ones need to be delayed, answer for both scenarios –

A) Scenario 1: If the transactions arrive in the order T1-T2-T3. Explain your answer.

B) **Scenario 2:** If the transactions arrive in the order T1-T3-T2. Explain your answer.

Note that all transactions need to take the locks when they start to run.



Compatibility Mode of Granular Locks

Current	None	IS	IX	S	SIX	U	X
Request	+/- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

A) At root, T1 and T2 can run concurrently as IS and SIX are compatible with each other, while T3 will be delayed due to the conflict between S and SIX. For File-A, T1 and T2 can run concurrently as IS and IX are compatible with each other. Additionally, T2 can request X lock on File:A3 and T1 can request S lock on File: A1, because they accessing the resources on different small branches.T2 can also request U lock on File-C firstly and then secondly request File-C1, while T3 would be still delayed at root node.
Overall, T1 and T2 can run concurrently from the beginning till the end but T3 will be delayed.
B) At root, T1 and T3 can run concurrently as IS and S are compatible with each other, while T2 will be delayed due to the conflict between S and SIX.
At File-A, T1 can request IS lock as T2 would still be delayed at root node; Similarly at File-C, T3 can request S lock without any conflict, and after that T3 also can request S File-C1. But T2 still would be delayed.
Overall, T1 and T3 can run concurrently from start till commit, but T2 has to delayed at root node due to conflict.

COMP90050 ADBS

Exercise – 2022 Win

Question 9 6 pts

Given a nested transaction where the parent transaction is P, the children transactions of P are T1 and T2, please answer for both scenarios.

(i) Scenario 1: When the value of the variable some_input is 3, which of the following transactions can run concurrently from the beginning till end (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give explanation for the delayed transactions.

(ii) Scenario 2: When the value of the variable some_input is 1, which of the following transactions can run concurrently from the beginning till end (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give explanation for the delayed transactions.

A compatibility matrix is as follows-

	T1	T2
P	BEGIN	BEGIN
BEGIN	Lock (IS,A)	Lock (IX,A)
Start both T1 and T2	If(some_input == 3){	If(some_input == 3){
Lock (S,A)	Lock(S,A)	Lock (X,A)
Read A	Read A	Write A
Unlock A	}	}
END	Unlock A	Unlock A
	END	END

Compatibility Mode of Granular Locks

Current	None	IS	IX	S	SIX	U	X
Request	+/- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)