

Transactions and Concurrency Control

CS 143 Introduction to Database Systems

TA: Jin Wang and Mingda Li

03/06/2020

Information

- Project 2 is due this Sunday night
 - Cut-off date: next Tuesday night
- Homework 6 is due next Monday
- Tips for Project 2
 - DO NOT modify the functions that have been defined
 - Pack the whole project strictly following the script
 - We will have hidden tests with different values
 - Regarding functionalities like exceptions, empty input, just follow given tests

Outline

- Transaction
- Concurrency Control
- Locks

Transactions

- Sequence of instructions you want to execute as if one:

CookieJar := CookieJar - 1

CookiesEaten := CookiesEaten + 1

- Want to ensure:
 - Either both execute, or neither
 - No other transaction sees only part of this execution:
- In SQL, they are typically represented with the following block:

BEGIN TRANSACTION

... # Data operations

END TRANSACTION



ACID Property

● Atomicity

- The result of all operations in a transaction are represented in the database, or none are.
- During a transaction, the RDBMS will start executing data operations. If at any point a failure occurs, the transaction stops and undoes the operations it just did.
- This is called a **rollback**

● Consistency

- When transactions are executed in isolation, data remains consistent
- `CookiesInJar >= 0`
- The definition of consistent depends on the use case. We do not want to lose data, we do not want additional extraneous data, and all data must have the proper values.

ACID Property (cont.)

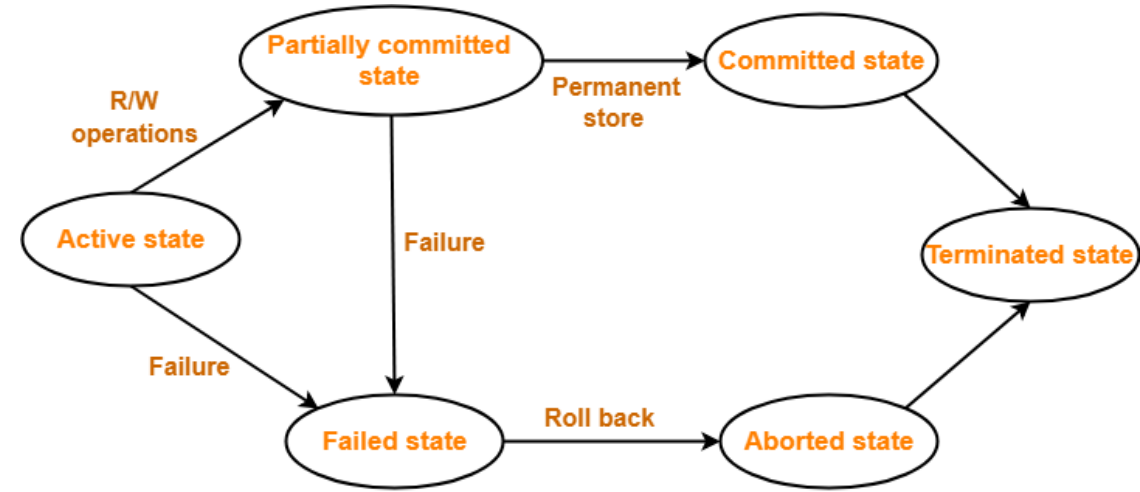
- **Isolation**

- Transaction runs as if it's the only one

- **Durability**

- After a transaction completes successfully the changes it makes persist to the database, even if there are system failures.
- Basically, changes are never lost

Stages of a Transaction



Transaction States in DBMS

- **Committed:** A transaction completes successfully.
- **Partially committed:** After all steps have run successfully, but before the result has been written to the database.
- If a transaction has **failed**, all changes are rolled back (undone) and the transaction is then in the **aborted state**.
- A transaction is **active** while it is still processing data. Once a transaction has failed, the database system can restart it (if the issue was transient) or kill it.

Outline

- Transaction
- Concurrency Control
- Locks

Isolation & concurrency control

- If just execute 1 transaction at a time...
 - Very inefficient!
- Want to maximize parallelism while maintaining a sense of isolation
 - **“Concurrency Control!”**
- Ultimately, there may be tradeoff between strictly enforcing ACID requirements like isolation and performance

Serializability

- When multiple transactions access and modify data concurrently, we can get a whole slew of problems with **consistency**.
 - It is much easier to just have transactions run in serial
 - But concurrency usually allows faster processing times, and improve resource utilization.
- Serial schedule – “run one at a time”

Time	Transaction 1	Transaction 2
1		CookiesBaking := 5
2		Jar := Jar + CookiesBaking
3	Jar := Jar – 1	
4	Ated := Ated + 1	

Serial schedule

Serializability

- A schedule is **serializable** if
 - It is serial. OR
 - It is equivalent in result to a serial schedule.

Time	Transaction 1	Transaction 2
1		CookiesBaking := 5
2	Jar := Jar - 1	
3	Ated := Ated + 1	
4		Jar := Jar + CookiesBaking

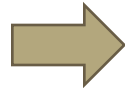
Schedule T1, T2 is serializable

Schedule Abstraction

Talking about the actual semantics of a program becomes hard

- Instead, we just **simplify to Reads and Writes**

Time	Transaction 1	Transaction 2
1		CookiesBaking := 5
2	Jar := Jar - 1	
3	Ated := Ated + 1	
4		Jar := Jar + CookiesBaking



Time	Transaction 1	Transaction 2
1		W(CookiesBaking)
2	R(Jar)	
3	W(Jar)	
4	R(Ated)	
5	W(Ated)	
6		R(Jar)
7		W(Jar)

Serializability

Time	Transaction 1	Transaction 2
1		W(CookiesBaking)
2		R(Jar)
3	R(Jar)	
4	W(Jar)	
5	R(Ated)	
6	W(Ated)	
7		W(Jar)

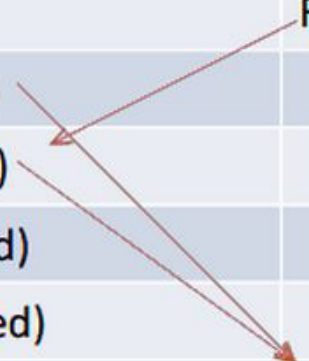
Is this schedule serializable?

How can we tell in general?

Serializability

- Conflict might happen for:
 - Two operations in **different transactions** on the **same object** where at least one is a **Write**.
- Concurrency issues can only arise in the face of conflicts.

Time	Transaction 1	Transaction 2
1		W(CookiesBaking)
2		R(Jar)
3	R(Jar)	
4	W(Jar)	
5	R(Ated)	
6	W(Ated)	
7		W(Jar)



Conflict Serializability

- Two schedules S and $S1$ are called **conflict equivalent** if we can convert S into $S1$ by swapping non-conflicting operations.
- If a schedule S is conflict equivalent to a serial schedule, we call it **conflict serializable**.
- A schedule is conflict serializable if and only if its dependency graph is acyclic.

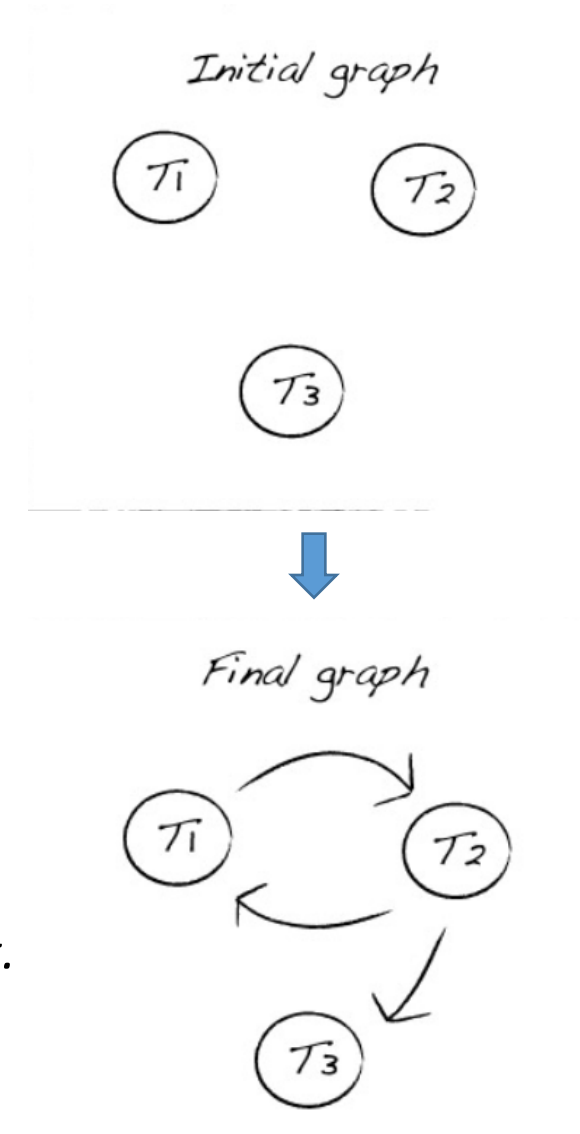
Conflict Serializability

- Simple and efficient method for determining conflict serializability of a schedule.
 - Consider some schedule S.
 - Create a **dependency graph** G from S.
 - The vertices consist of the individual transactions T_i , and we construct an edge from T_i to T_j if:
 - T_i executes write(X) before T_j executes read(X)
 - T_i executes read(X) before T_j executes write(X)
 - T_i executes write(X) before T_j executes write(X)
- 1) If an edge $T_i \rightarrow T_j$ exists in graph, then in any serial schedule S1 equivalent to S, T_i must appear before T_j .
- 2) If there are **no cycles** in the graph, then S is **conflict serializable**.

Dependency Graph

T1	R1(X)		R1(X)
T2		R2(Y) R2(Y)	W2(X)
T3			W3(Y)

- Check if there is a Tx that reads an item after a different Tx writes it.
 - We have T1 that reads X after T2 writes it, so draw arrow from T2 -> T1
- Check if there is a Tx that writes an item after a different Tx reads it.
 - We have T2 that writes X after T1 reads it, so draw arrow from T1 -> T2
 - We also have T3 that writes Y after T2 reads it, so draw arrow from T2 -> T3
- Check if there is a Tx that writes an item after a different Tx writes it.
 - not in this case



1. Consider the following schedules:

T1		R(A)	W(A)	R(B)					
T2					W(B)	R(C)	W(C)	W(A)	
T3	R(C)								W(D)

(a) Draw the dependency graph (precedence graph) for the schedule.

T1 -> T2: R(A), W(A) before W{A}

T3 -> T2: R(C) before W(c)

(b) Is the schedule conflict serializable? If so, what are all the (conflict) equivalent serial schedules? If not, why not?

Yes.

T1T3T2, T3T1T2

T1	R(A)		R(B)				W(A)	
T2		R(A)		R(B)				W(B)
T3					R(A)			
T4						R(B)		

T1 -> T2 (B)

T2 -> T1 (A)

T3 -> T1 (A)

T4 -> T2 (B)

(c) Draw the dependency graph (precedence graph) for the schedule.

(d) Is the schedule conflict serializable? If so, what are all the (conflict) equivalent serial schedules? If not, why not?

No. There is a cycle between T1 and T2

Outline

- Transaction
- Concurrency Control
- Locks

Locks

- We use locks to control access to objects.
 - Shared lock: multiple transactions can have a shared lock on the same item (e.g., reading)
 - Exclusive lock: only one (and no other lock) on this item (e.g., writing)

Time	Transaction 1	Transaction 2
1	Lock_X(Jar)	Lock_X(CookiesBaking)
2	R(Jar)	W(CookiesBaking)
3	W(Jar)	Unlock(CookiesBaking)
4	Unlock(Jar)	
5	Lock_X(Ated)	Lock_X(Jar)
6	R(Ated)	R(Jar)
7	W(Ated)	W(Jar)
8	Unlock(Ated)	Unlock(Jar)

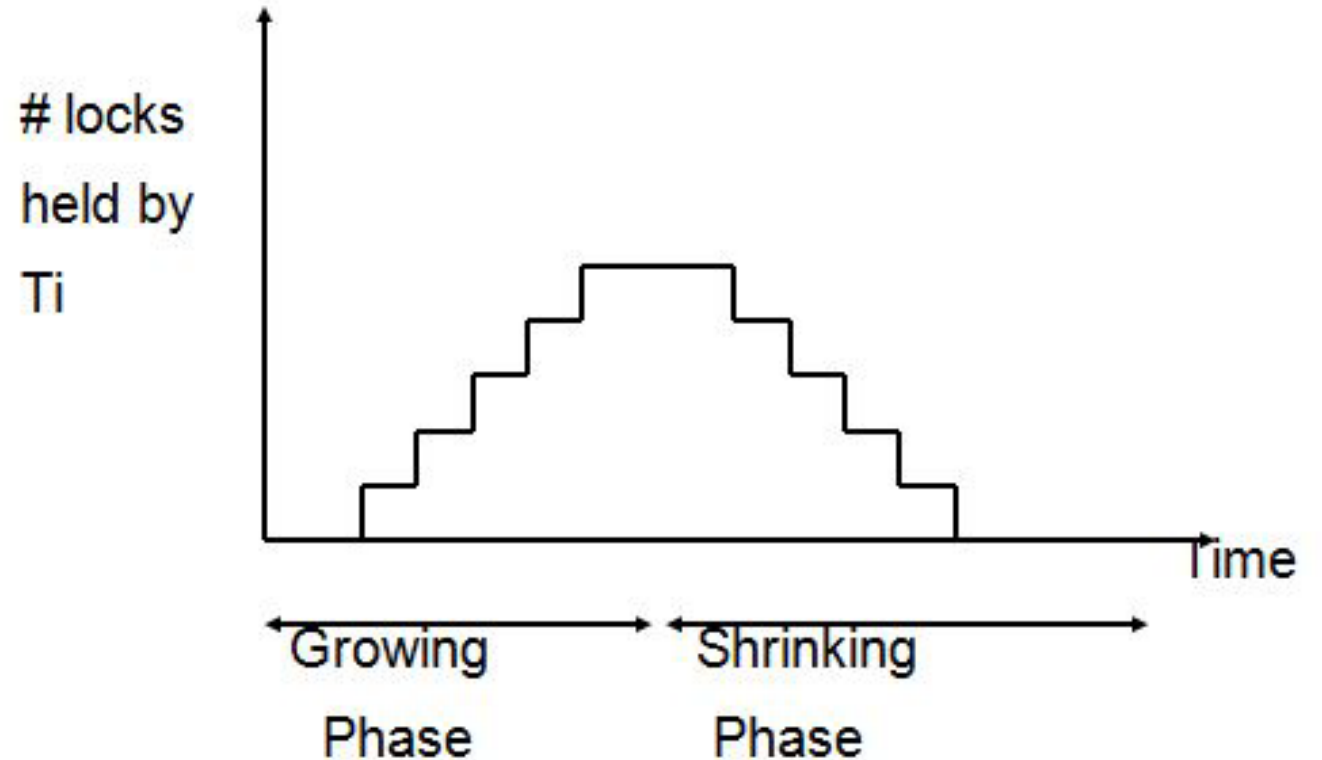
Two Phase Locking Protocol (2PL)

- We add a rule for how a transaction may acquire locks:
 - Once you release a lock, you may never acquire anew lock.
- Ensures conflict serializability!
 - In order for a conflict cycle to occur, we need to release a lock so other guy can use our object, then acquire a lock to use the other guy's object
 - You can convert into a serial schedule, where the transactions follow the order of the locking points.

2PL (cont.)

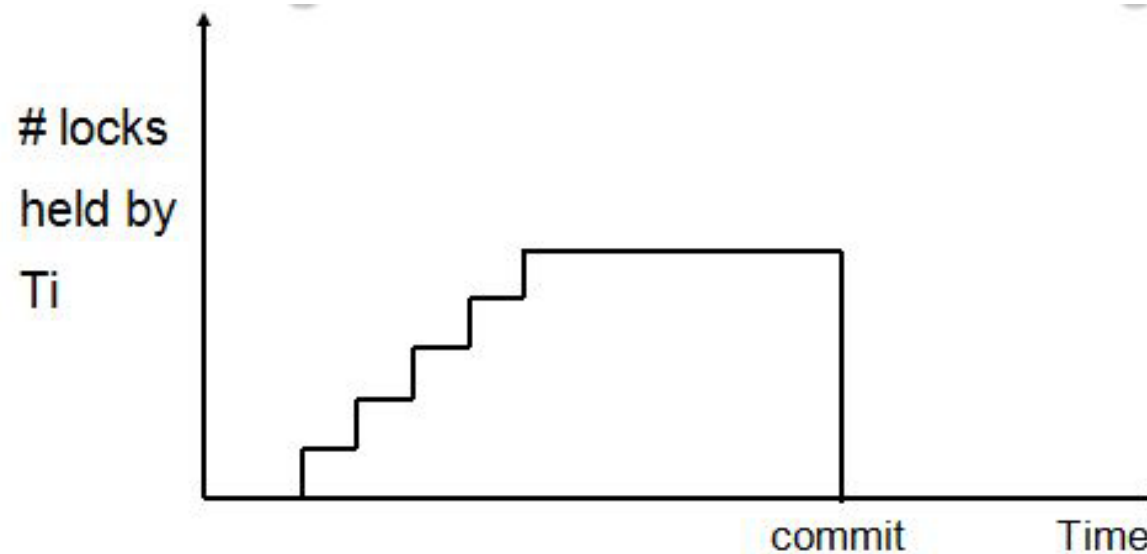
A transaction is said to follow Two Phase Locking protocol if Locking and Unlocking can be done in two phases.

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.



Strict and Rigorous 2PL

- Strict 2PL
 - All x-locks held by transaction are only released at the end of the transaction.
- Rigorous 2PL
 - All locks held by transaction are only released at the end of the transaction.



Handling Deadlock in 2PL

- Dealing with deadlocks:
 - Prevention: stop them from occurring
 - Detection: stop them while occurring
 - In practice: timer

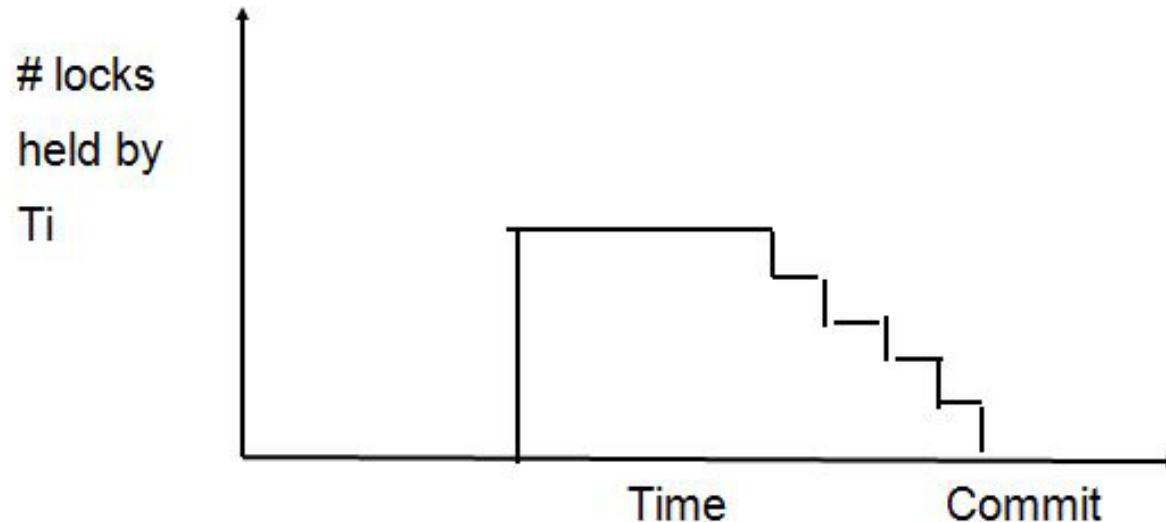
Time	Transaction 1	Transaction 2
1	Lock_X(A) (granted)	
2		Lock_X(B) (granted)
3	Lock_X(B) (waiting)	
4		Lock_X(A) (waiting)
5

Deadlock Prevention

- Disallow deadlocks from ever occurring: given two transactions, old one T_o and new one T_n .
- Wait-Die
 - If T_o is waiting for a lock from T_n , it just waits.
 - If T_n is waiting for a lock from T_o , it kills himself.
- Wound-Wait
 - If T_o is waiting for a lock from T_n , it kills T_n .
 - If T_n is waiting for a lock from T_o , it just waits.
 - If you die, you restart with original timestamp.

Deadlock Prevention

- Conservative 2PL.
- Our transaction requests all the locks at the beginning,
If it does not get them it does not start.
If it gets the resources and starts, it will complete.



Deadlock Detection

- Waits-for graph of all transactions.
 - $G = (V, E)$
 - V is a set of Transactions
 - E is a set of arcs between transactions: $A \rightarrow B$
 - $A \rightarrow B$ if A is waiting for B to release a lock on some data item.
- If cycle exists, shoot one of the transactions in the cycle.

2) a. What will be printed in the following execution (B=3, F=300)?

Print: 3030

Lock_X(B)	
Read(B)	
B = B*10	
Write(B)	Lock_S(B)
Lock_X(F)	
F = B*100	
Write(F)	
Unlock(B)	
Unlock(F)	
	Lock_S(F)
	Read(F)
	Read(B)
	Print(F+B)
	Unlock(B)
	Unlock(F)

T1

T2

b. Does the execution use: (a) 2PL or (b) Strict 2PL?

(a) Yes. It is 2PL; (b). Yes. It is strict 2PL