Transactions

Lecture 15

Outline

1. Why Transactions are Important
   - Recovery
   - Concurrency

2. Transaction Support in SQL
   - Isolation Levels

3. DBMS Theory & Implementation
   - Schedule Characterization
   - Concurrency Control via Two-Phase Locking

Transactions So Far

A transaction is a logical sequence of database operations (reads/writes)

- In SQL, starts with BEGIN, ends with either COMMIT or ROLLBACK

Desirable properties…

- **A**tomicity: all or nothing
- **C**onsistency: start/end with all constraints met
- **I**solation: appear as though independent of others
- **D**urability: changes via committed transactions persist

So what does a DBMS have to do in order to support <u>correct</u> and <u>efficient</u> transaction processing?

Issues Related to Recovery

- Various kinds of failures can occur
  - System crash, error (e.g. divide by zero), hardware failure, external failure (e.g. power)
  - Local error detected by transaction (e.g. insufficient funds)

- Atomicity: undo all actions for rollback/err
- Durability: redo actions after err for commit

Note: more detail in Chapter 22

Side Note: Consistency

- Most of the time we think of consistency from the DBMS standpoint
  - Often in context of failure, concurrency
- But it may be the case that transactions themselves are poorly written w.r.t. database constraints
  - And thus are legitimately aborted

Checkup

- Assume a database has the following asserted constraint: $A > B > 0$

- Which transactions will NOT necessarily preserve consistency of the database?
  - Provide an example

i. $A = 2A; B = 2B$

ii. $A = 2A; B = A - 1$

Answer (A > B > 0)

i. A = 2A; B = 2B
   - WILL preserve
   - If both started > 0, will remain so under multiplication
   - If A > B, 2A > 2B

ii. A = 2A; B = A – 1
   - WILL NOT (always) preserve
   - Start: A=0.5, B=0.4
     - Result: A=1, B=0

Issues Related to Concurrency

- Multiple users (or one user with multiple requests) submit transactions at about the same time
  - Isolation: shouldn't affect one another
  - Consistency: committed effects might cause rollback
- One approach to transaction processing: one transaction gets to execute at a time
  - Pro: simple, correct
  - Con: slow :(
- This **serial** schedule is our baseline for correctness

Checkup

- Suppose users Alice and Bob are issuing transactions to a common database
  - Alice issues transaction A1, then A2
  - At about the same time …
  - Bob issues transaction B1, then B2

- What are the possible serial schedules in this scenario?

Answer

- A1, A2, B1, B2
- A1, B1, A2, B2
- A1, B1, B2, A2
- B1, A1, A2, B2
- B1, A1, B2, A2
- B1, B2, A1, A2

Interleaving Operations

- The core question for a DBMS…
  - How to improve resource utilization in efficient transaction processing while maintaining correct results?

- Stated another way…
  - To what extent can we interleave transactions without introducing errors?

Example Scenario

- Consider a flight reservation system
  - Need to keep track of reserved seats per flight

- T1(X, M): reserve M seats on flight X

```
a) UPDATE reservations
   SET seats=seats+M
   WHERE flight=X
```

- T2(X, Y, N): transfer N seats from flight X to Y

```
a) UPDATE reservations
   SET seats=seats-N
   WHERE flight=X
b) UPDATE reservations
   SET seats=seats+N
   WHERE flight=Y
```

Start with seats on X=10, Y=20

If the following requests are made at about the same time, what are final values?

T1(X, 2); T2(X, Y, 5)

Modeling Transactions

Let's break up the transactions into primitive operations: reading into memory (r), performing computations in memory, and writing (w) results to disk (or at least a log)

T1(X, M)

- r(X)
- X = X + M
- w(X)

T2(X, Y, N)

- r(X)
- X = X – N
- w(X)
- r(Y)
- Y = Y + N
- w(Y)

What Could Go Wrong?

Lost Update

Time	T1(X, M)	T2(X, Y, N)
↓	r(X)	
	X = X + M	
		r(X)
		X = X - N
	w(X)	
		w(X)
		r(Y)
		Y = Y + N
		w(Y)

Start with seats on X=10, Y=20

Final Values for… T1(X, 2); T2(X, Y, 5)

What Could Go Wrong?

Dirty Read

Time	T1(X, M)	T2(X, Y, N)
↓		r(X)
		X = X - N
		w(X)
	r(X)	
	X = X + M	
	w(X)	
		r(Y)
		ROLLBACK

Start with seats on X=10, Y=20

Final Values for… T1(X, 2); T2(X, Y, 5)

What Could Go Wrong?

Incorrect Summary

Time	SUM()	T2(X, Y, N)
↓	SUM = 0	
		r(X)
		X = X - N
		w(X)
	r(X)	
	SUM = SUM + X	
	r(Y)	
	SUM = SUM + Y	
		r(Y)
		Y = Y + N
		w(Y)

What Could Go Wrong?

Unrepeatable Read

Time	T3(X)	T2(X, Y, N)
↓	r(X)	
	…	
		r(X)
		X = X - N
		w(X)
	r(X)	
	…	
		r(Y)
		Y = Y + N
		w(Y)

Transactions in SQL

- By default, according to SQL-92, transaction execution…

  is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.

- You have two knobs at your disposal to improve performance
  - Access Mode (default: **`READ WRITE`**)
    - If **`READ ONLY`**, SELECT allowed, might be faster
  - Isolation Level (default: **`SERIALIZABLE`**)
    - If other, allows certain kinds of isolation violations for potential speed improvement

Isolation Levels in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

- Dirty Read
  - Can read values uncommitted by other transactions
  - Think issues with ROLLBACK
- Nonrepeatable Read
  - Can read values changed by other committed transactions
  - Values in T1 can change in subsequent reads
- Phantom:
  - A row that did not exist at the start of a transaction, but then visible

DBMS Theory & Implementation

- Now that we understand some of the issues of transactions, we'll more formally characterize interleaved operations

- Then we'll look at one mechanism by which RDBMSs efficiently support correct transaction processing

Schedules of Transactions

- A **schedule**, S, of n transactions T_1, T_2, … T_n is an ordering of the operations of the transactions
- Operations of interest, with shorthand…
  - Read=r, Write=w
  - Commit=c, Rollback=a (abort)

Example A

Time	T1(X, M)	T2(X, Y, N)
↓	r(X)	
	X = X + M	
		r(X)
		X = X - N
	w(X)	
		w(X)
		r(Y)
		Y = Y + N
		w(Y)

S_A: $r_1(X)$, $r_2(X)$, $w_1(X)$, $w_2(X)$, $r_2(Y)$, $w_2(Y)$

Example B

Time	T1(X, M)	T2(X, Y, N)
↓		r(X)
		X = X - N
		w(X)
	r(x)	
	X = X + M	
	w(x)	
		r(Y)
		ROLLBACK

S_B: $r_2(X)$, $w_2(X)$, $r_1(X)$, $w_1(X)$, $r_2(Y)$, a_2

Characterizing Recoverability

- Some schedules allow for easy recovery; others are difficult or impossible
- We now look to characterize these levels
- These distinctions don't tell us how the DBMS implements recovery/scheduling, but at least defines the expected outputs

Defining Recoverability

- To satisfy durability, once a transaction is committed, it should never have to be rolled back
- A schedule that satisfies this criterion is **recoverable**
- A schedule S is recoverable if …
  – No transaction T in S commits until …
  – All transactions T' that have written some item X that T reads have committed

Recoverable?

S: $r_1(X)$, $w_1(X)$, $r_2(X)$, $r_1(Y)$, $w_2(X)$, c_2

- This schedule is NOT recoverable because…
  - T2 reads X after T1 wrote it
  - AND T2 commits before T1

- SO, if T1 rolls back, so too must T2…
  - But T2 has already committed!!???

- Corrected, either…
  - $r_1(X)$, $w_1(X)$, $r_2(X)$, $r_1(Y)$, $w_2(X)$, c_1, c_2
  - $r_1(X)$, $w_1(X)$, $r_2(X)$, $r_1(Y)$, $w_2(X)$, a_1, a_2

Avoiding Cascade

- We have defined a baseline for a recoverable schedule (i.e. one that supports durability)
- However, some recoverable schedules lead to **cascading rollbacks**: where T1 needs to rollback because T2 did
  - This is expensive!
- A schedule is **cascadeless** if every transaction reads only items that were written by committed transactions

Characterize

S: $r_1(X)$, $w_1(X)$, $r_2(X)$, $r_1(Y)$, $w_2(X)$, c_1, c_2

- This schedule is recoverable
  - T2 reads X after T1 wrote it
  - AND T2 commits after T1
- This schedule is not cascadeless
  - T2 reads X after T1 wrote it, but before T1 has committed

Strict Schedules

- The most restrictive type is **strict**: transactions can neither read nor write X until the last transaction that wrote X has committed/rolled back
- Makes recovery very easy
  - Can store "before image", or old value, of each changed variable
- Strict -> Cascadeless -> Recoverable

Characterize

S: $w_1(X), w_2(X)$

- This schedule is recoverable
  - No reading between transactions
- This schedule is cascadeless
  - No reading between transactions
- This schedule is NOT strict
  - T2 writes X before T1 commits
  - Imagine T1 rolls back
    - If X=10 before, can't simply restore 10
    - We'd lose T2's version

Characterizing Serializability

- We now shift to characterizing correctness of concurrent transactions

- Recall: schedule S is **serial** if, for every transaction T participating in the schedule, all operations of T are executed consecutively in the schedule

Example Serial Schedules

(a)

Time ↓

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time ↓

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

Serializable

- Serial scheduling is typically too slow for real-world use
- A schedule is **serializable** if it is "equivalent" to some serial schedule
  – Note: related to, but not the same as SQL
- We will focus on one definition of how to compare two schedules, **conflict serializability**, which involves the idea of **conflicting** operations

Conflicting Operations

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions…

1. They belong to different transactions
2. They access the same item (e.g. X)
3. At least one is a write operation

Checkup

- List all conflicts in the following schedule

S_A: $r_1(X)$, $r_2(X)$, $w_1(X)$, $w_2(X)$, $r_2(Y)$, $w_2(Y)$

Answer

Read-Write

- $r_1(X)$, $w_2(X)$
- $r_2(X)$, $w_1(X)$

Write-Write

- $w_1(X)$, $w_2(X)$

S_A: $r_1(X)$, $r_2(X)$, $w_1(X)$, $w_2(X)$, $r_2(Y)$, $w_2(Y)$

Conflict Serializability

- Two schedules are **conflict equivalent** if the relative order of any two conflicting operations is the same in both schedules
  - Another view: two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations
  - Note: can't change relative ordering *within* each transaction
- A schedule is **conflict serializable** if it is conflict equivalent to a serial schedule

Example

Are the following schedules conflict equivalent?

S_A: $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$, $r_2(X)$, $w_2(X)$
S_D: $r_1(X)$, $w_1(X)$, $r_2(X)$, $w_2(X)$, $r_1(Y)$, $w_1(Y)$

Yes: swap $r_1(Y)/r_2(X)$, $w_1(Y)/w_2(X)$

- Alternatively…
  - $r_1(X) < w_2(X)$
  - $w_1(X) < r_2(X)$
  - $w_1(X) < w_2(X)$

Testing for Conflict Serializability

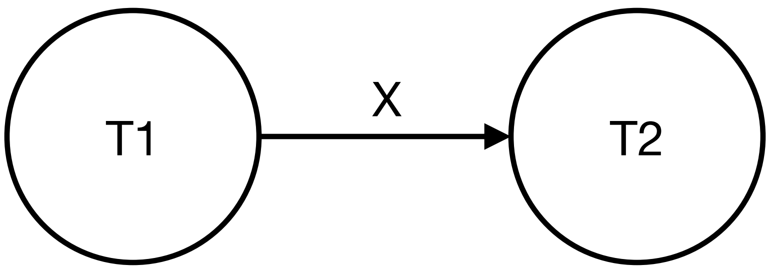
Construct a **precedence/serialization** graph

1. Create nodes for every transaction
2. Draw an edge from node J to K if a pair of conflicting operations exist in T_J and T_K and the conflicting operation in T_J appears in the schedule before the conflicting operation in T_K

A cycle indicates non-serializability

Example A

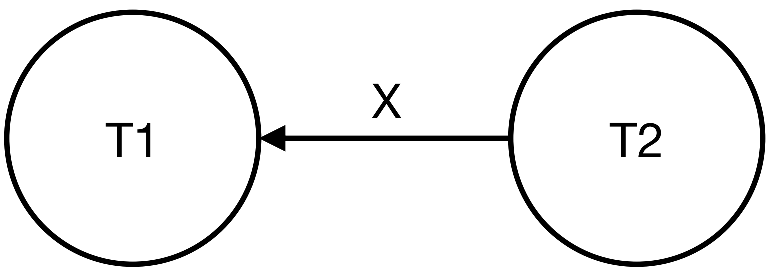
S_A: $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$, $r_2(X)$, $w_2(X)$



Conflict Serializable: {(T1, T2)}

Example B

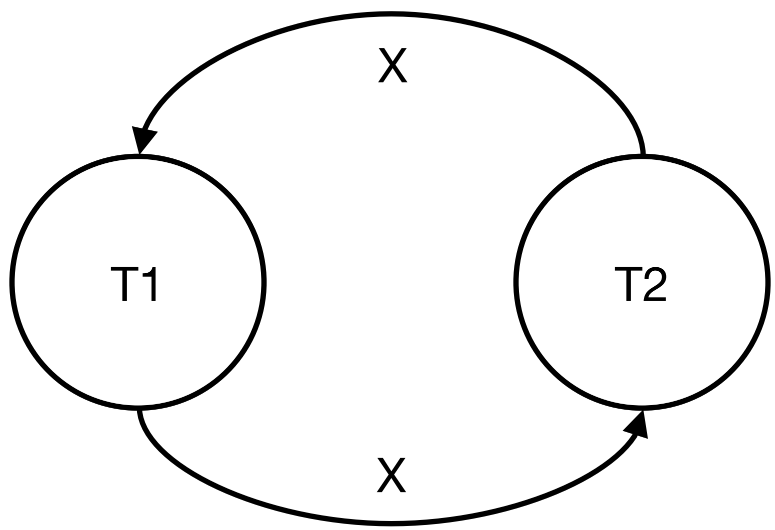
S_B: $r_2(X)$, $w_2(X)$, $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$



Conflict Serializable: {(T2, T1)}

Example C

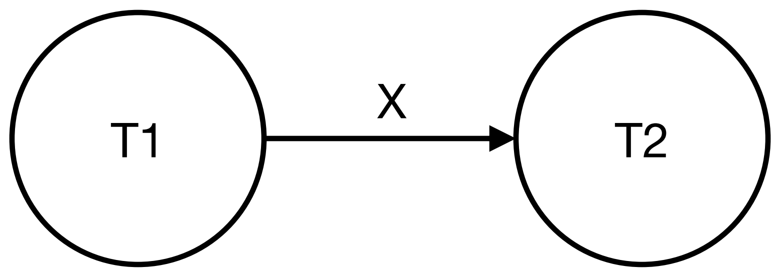
S_C: $r_1(X)$, $r_2(X)$, $w_1(X)$, $r_1(Y)$, $w_2(X)$, $w_1(Y)$



Conflict Serializable: {}

Example D

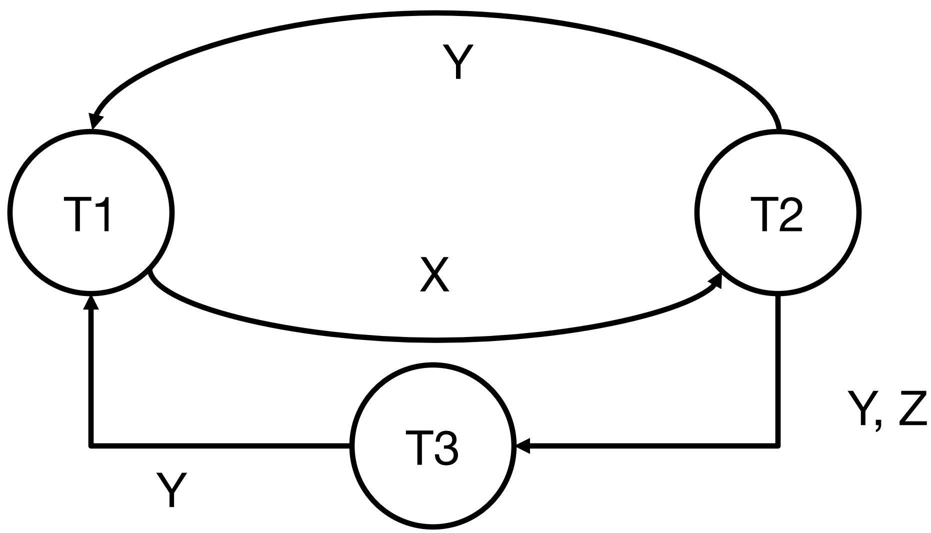
S_D: $r_1(X)$, $w_1(X)$, $r_2(X)$, $w_2(X)$, $r_1(Y)$, $w_1(Y)$



Conflict Serializable: {(T1, T2)}

Example E

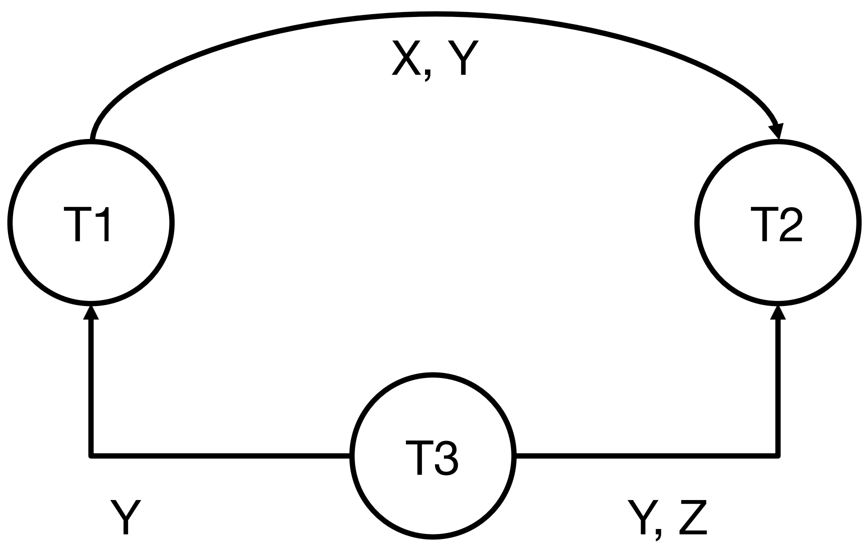
S_E: $r_2(Z)$, $r_2(Y)$, $w_2(Y)$, $r_3(Y)$, $r_3(Z)$, $r_1(X)$, $w_1(X)$, $w_3(Y)$, $w_3(Z)$, $r_2(X)$, $r_1(Y)$, $w_1(Y)$, $w_2(X)$



Conflict Serializable: {}

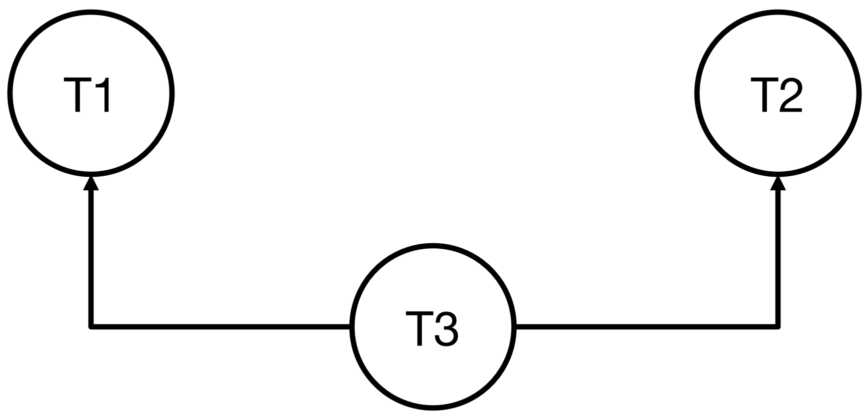
Example F

S_F: $r_3(Y)$, $r_3(Z)$, $r_1(X)$, $w_1(X)$, $w_3(Y)$, $w_3(Z)$, $r_2(Z)$, $r_1(Y)$, $w_1(Y)$, $r_2(Y)$, $w_2(Y)$, $r_2(X)$, $w_2(X)$



Conflict Serializable: {(T3, T1, T2)}

Conflict Serializable?



{(T3, T1, T2), (T3, T2, T1)}

Implementing Transactions

- The characterizations presented thus far can be computationally expensive to use in practice
- Instead, DBMSs typically utilize **protocols** (sets of rules) that will ensure desired properties
- We focus on one: **Two-Phase Locking** (2PL)
  – Most common for concurrent processing
  – Others: see Ch. 21

Locking Primer

- A **lock** is a variable associated with a data item, used to describe item status w.r.t. some set of operations
  - "Data item" intentionally left vague (e.g. value, row, table, database)
- Simplest example: binary lock
  - Lock: I can read/write, no other can access
    - Attempts simply "wait"
  - Unlock: available for locking

Read/Write Lock

- Binary locks restrict access, but at too high a computational cost
- If we recognize that two transactions can safely read the same data item, we enter the idea of shared/exclusive locking
- So now reading requires a read lock, writing requires a write lock
  - Keep track of number of shared users

Using Locks ≠ Serializability (1)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

Using Locks ≠ Serializability (2)

Time ↓

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

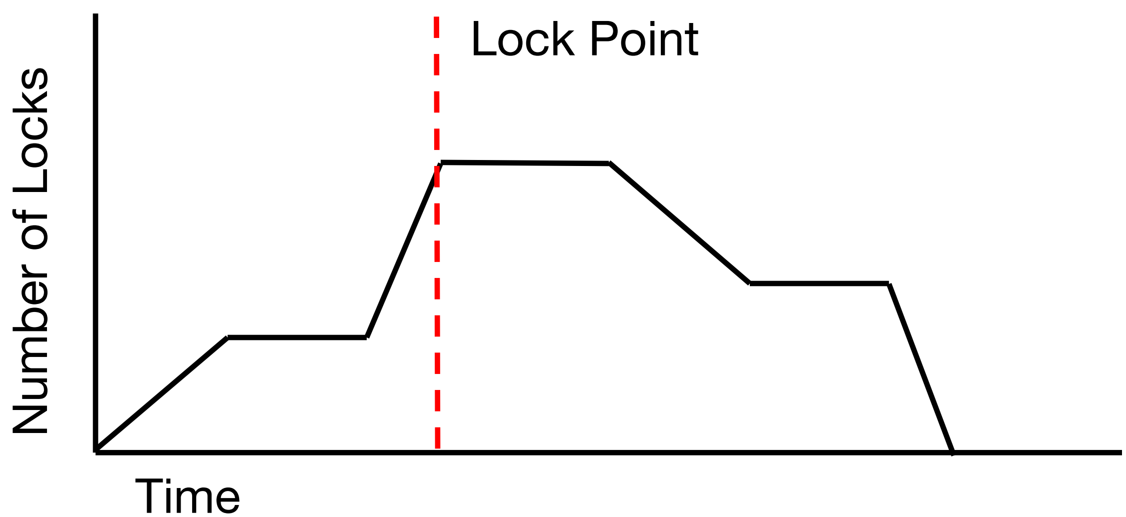
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Initial values: X=20, Y=30

Result of schedule S:
X=50, Y=50
(nonserializable)

Two-Phase Locking (2PL)

- 2PL Protocol: all locking operations precede the first unlock
  1. Growing Phase
  2. Shrinking Phase



Checkup: 2PL?

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Compare

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

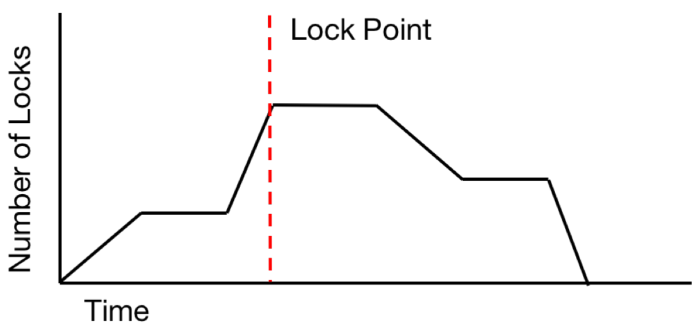
2PL = Serializability

- Following this "basic" 2PL protocol guarantees serializable schedules
  - Proof idea: think about what a cycle in the precedence graph implies about lock times

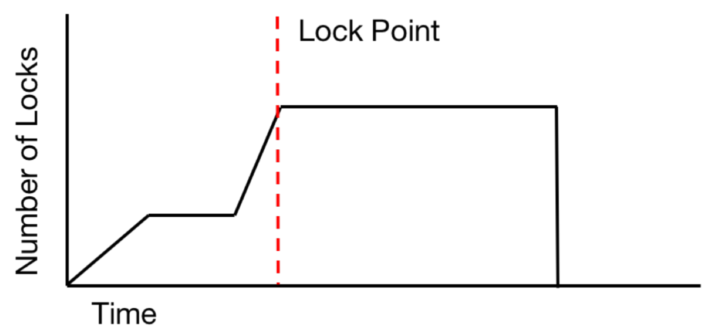
- A common **Strict 2PL** protocol also avoids cascading rollbacks
  - Hold all write locks till transaction end
  - The **Rigorous** or **Strong-Strict** (SS2PL) variant is easier to implement and holds for all locks

Compare

2PL



SS2PL



An Issue?

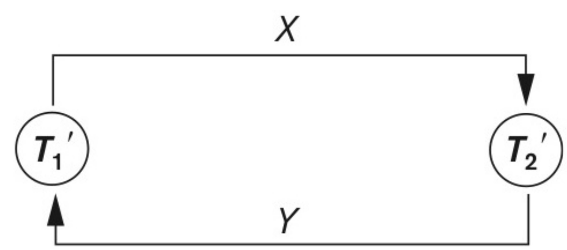
Time ↓

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Dealing with Deadlocks

- A **deadlock** occurs when each transaction is waiting to lock an item that is locked by another transaction

- Typical approaches…
  - Detection via **wait-for graph**
    - But when to pay the cost?
  - Timeout



- Make sure to avoid **starvation** via a fair **victim-selection** policy