Chapter 7: Transactions

Contents

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- Collection of operations that form a single logical unit of work are called transactions

Example Transaction to transfer \$50 from account A to account B:

1. read(A)
2. $A := A - 50$
3. write(A)
4. read(B)
5. $B := B + 50$
6. write(B)

Two main issues to deal with:

1. Failures of various kinds, such as hardware failures and system crashes
2. Concurrent execution of multiple transactions

Example of Fund Transfer Cont.

Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

System Failure

- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database

Example of Fund Transfer Cont.

Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- **Durability requirement**
  - once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer Cont.

Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

A+B = 500

A+B = 450

A+B = 500

- Consistency requirement
  - the sum of A and B is unchanged by the execution of the transaction

- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent

Example of Fund Transfer Cont.

Transaction T_1 to transfer $50 from account A to account B, Transaction T2 print out the sum of the balance of A and B

T_1

1. **read**(A)
2. $A := A - 50$
3. **write**(A)

4. **read**(B)
5. $B := B + 50$
6. **write**(B)

T_2

1. **read**(A)
2. **read**(B)
3. **print**(A+B)

Output
A+B = 450

- **Isolation requirement**
  - if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

Example of Fund Transfer Cont.

Transaction T_1 to transfer $50 from account A to account B, Transaction T2 print out the sum of the balance of A and B

T_1

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

T_2

1. **read**(A)
2. **read**(B)
3. **print**(A+B)

- Isolation can be ensured trivially by running transactions serially
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

To preserve the integrity of data in the database system, a transaction must ensure:

- Atomicity. Either all operations of the transaction are properly reflected in the database or none are.
- Consistency. Execution of a transaction in isolation preserves the consistency of the database.

Cont.

ACID Properties

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.

That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Cont.

ACID Properties

- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

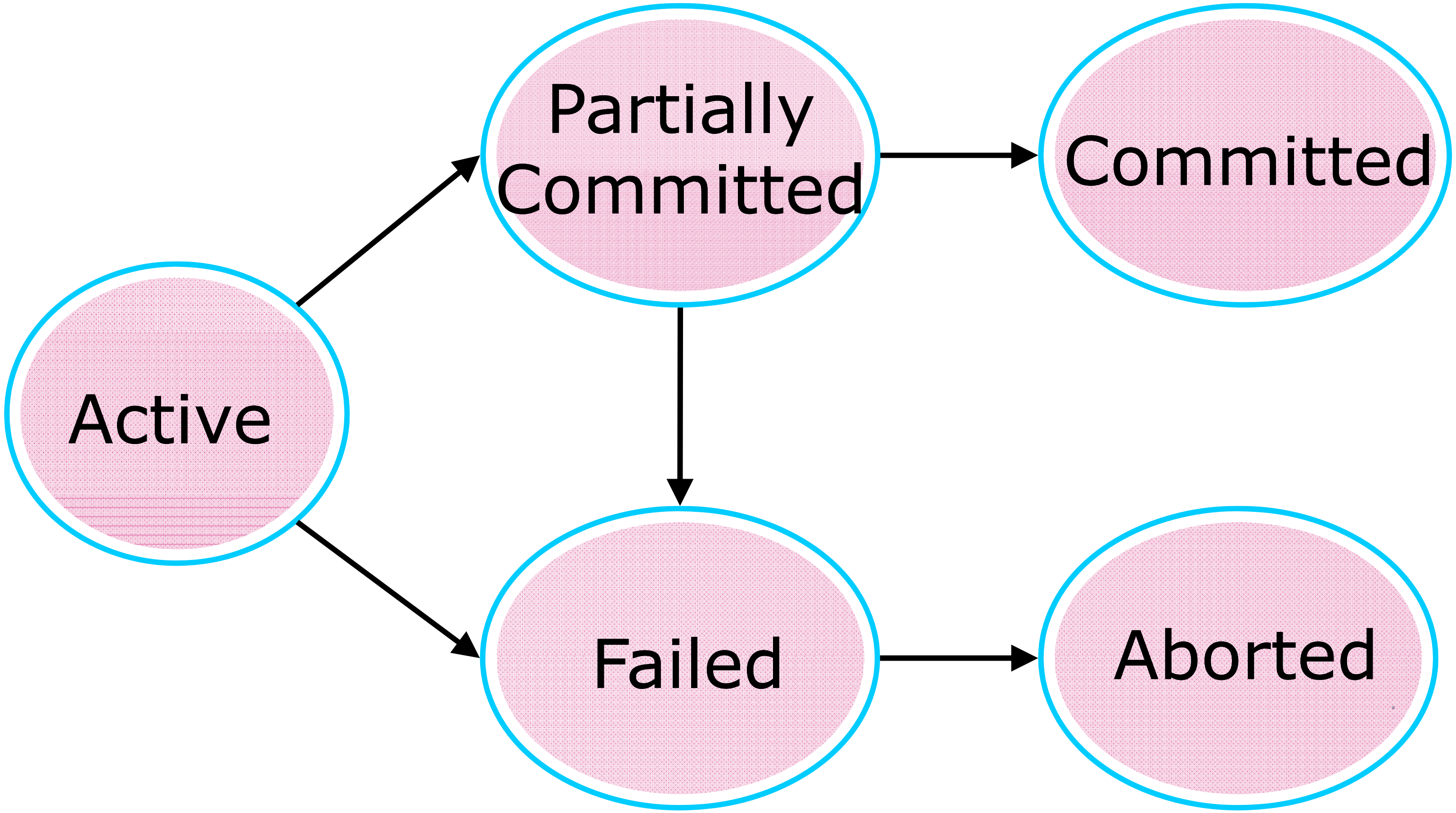
- Active – the initial state; the transaction stays in this state while it is executing
- Partially committed – after the final statement has been executed.
- Failed -- after the discovery that normal execution can no longer proceed.
- Committed – after successful completion.

Transaction State Cont.

- Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction

Transaction State

Cont.



Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability
  - Log file
  - Backup

Log based recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
  - Before T_i executes write(*X*), it write a log record

$<T_i, X, V_1, V_2>$

Transaction ID	Data Item ID	Old Value	New Value
T_i	X	V_1	V_2

Log based recovery

- When transaction T_i starts, it registers itself by writing a log record

$$\boxed{T_i \text{ start}}$$

- When T_i finishes its last statement, it writes a log record

$$\boxed{T_i \text{ commit}}$$

Log based recovery

- Two approaches using logs
  - Deferred database modification
    - Can only output updates to hard disk while in partial commit state
    - Using Redo
  - Immediate database modification
    - Can output updates to hard disk while still in the active state
    - Using Undo, Redo

Output log record to hard disk first

Checkpoints

- **Problems in recovery procedure as discussed earlier :**
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.

Checkpoints

- Streamline recovery procedure by periodically performing check pointing

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record <checkpoint> onto stable storage.

Checkpoints

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

1. Scan backwards from end of log to find the most recent <checkpoint> record
2. Continue scanning backwards till a record <T_i start> is found.

Checkpoints

assume transactions are run serially

3. Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.

4. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i commit>, execute redo(T_i).

5. For all transactions (starting from T_i or later) with no <T_i commit>, execute undo(T_i). (Done only in case of immediate modification.)

Concurrent Executions Cont.

- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*).

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

A: 150

temp: 20

B: 300

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is **serializable** if it is equivalent to a serial schedule.

Serializability Cont.

- Different forms of schedule equivalence give rise to the notions:

    1. conflict serializability

    2. view serializability

Serializability Cont.

- *Simplified view of transactions*
  – We ignore operations other than read and write instructions
  – We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  – Our simplified schedules consist of only read and write instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

  1. I_i = read(Q), I_j = read(Q). I_i and I_j don't conflict.
  2. I_i = read(Q), I_j = write(Q). They conflict.
  3. I_i = write(Q), I_j = read(Q). They conflict
  4. I_i = write(Q), I_j = write(Q). They conflict

Cont.

Conflicting Instructions

- If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

	T1	T2	
Q=2000	Read(Q)	Read(Q)	Q=2000
Q=2000	Read(Q)	Read(Q)	Q=2000

Q = 2000

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent.**
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability

Cont.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializable

Conflict Serializability

- **Example of a schedule that is not conflict serializable:**

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are view equivalent if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*

View Serializability

2. If in schedule S transaction T_i executes read(Q), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same write(Q) operation of transaction T_j.
3. The transaction (if any) that performs the final write(Q) operation in schedule *S* must also perform the final write(Q) operation in schedule *S'.*

View Serializability Cont.

- A schedule *S* is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

T_3	T_4	T_6
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

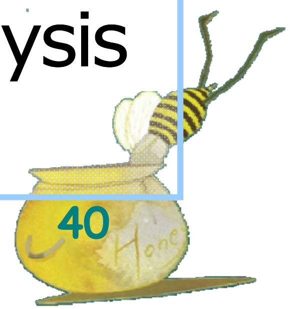
Blind write

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Determining such equivalence requires analysis of operations other than read and write.



Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — **if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.**

Cont.

Recoverable Schedules

- **The following schedule is not recoverable if T_9 commits immediately after the read**

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

fail

commit

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cont.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable,
  - and are recoverable
  - and preferably cascadeless

Lock-Based Protocols

- **A lock is a mechanism to control concurrent access to a data item**
- **Data items can be locked in two modes :**
  - ***exclusive (X) mode*. Data item can be both read as well as written**
  - ***shared (S) mode*. Data item can only be read.**
- **Lock requests are made to concurrency-control manager. Transaction can proceed only after lock is granted.**

Cont.

Lock-Based Protocols

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

Lock-Based Protocols

- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Pitfalls of Lock-Based Protocols

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Neither *T3* nor *T4* can make progress

Called **deadlock**

Solution:

Rollback one of the transactions

Wait

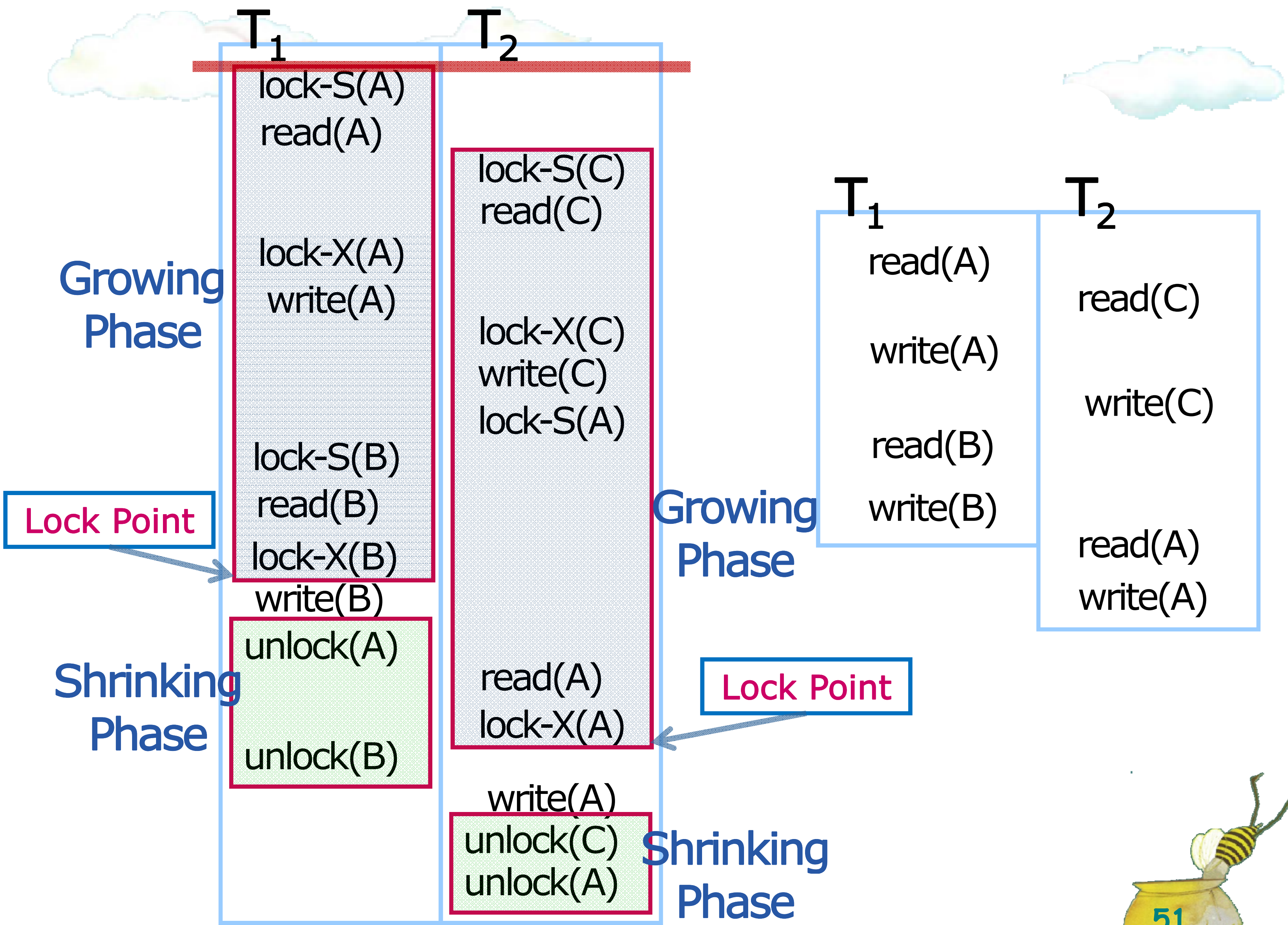
T3 to release lock on B

Wait

T4 to release lock on A

Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks



T1
T2
lock-S(A)
read(A)
lock-S(C)
read(C)
lock-X(A)
write(A)
lock-X(C)
write(C)
lock-S(A)
lock-S(B)
read(B)
lock-X(B)
write(B)
unlock(A)
unlock(B)
read(A)
lock-X(A)
write(A)
unlock(C)
unlock(A)
Growing Phase
Lock Point
Shrinking Phase
Growing Phase
Lock Point
Shrinking Phase
T1
T2
read(A)
read(C)
write(A)
write(C)
read(B)
write(B)
read(A)
write(A)

Two-Phase Locking Protocol

- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible

Conclusions