Unit 9: Transactions

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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:
  - Failures of various kinds, such as hardware failures and system crashes while executing transactions.
  - Concurrent execution of multiple transactions without inconsistency.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.

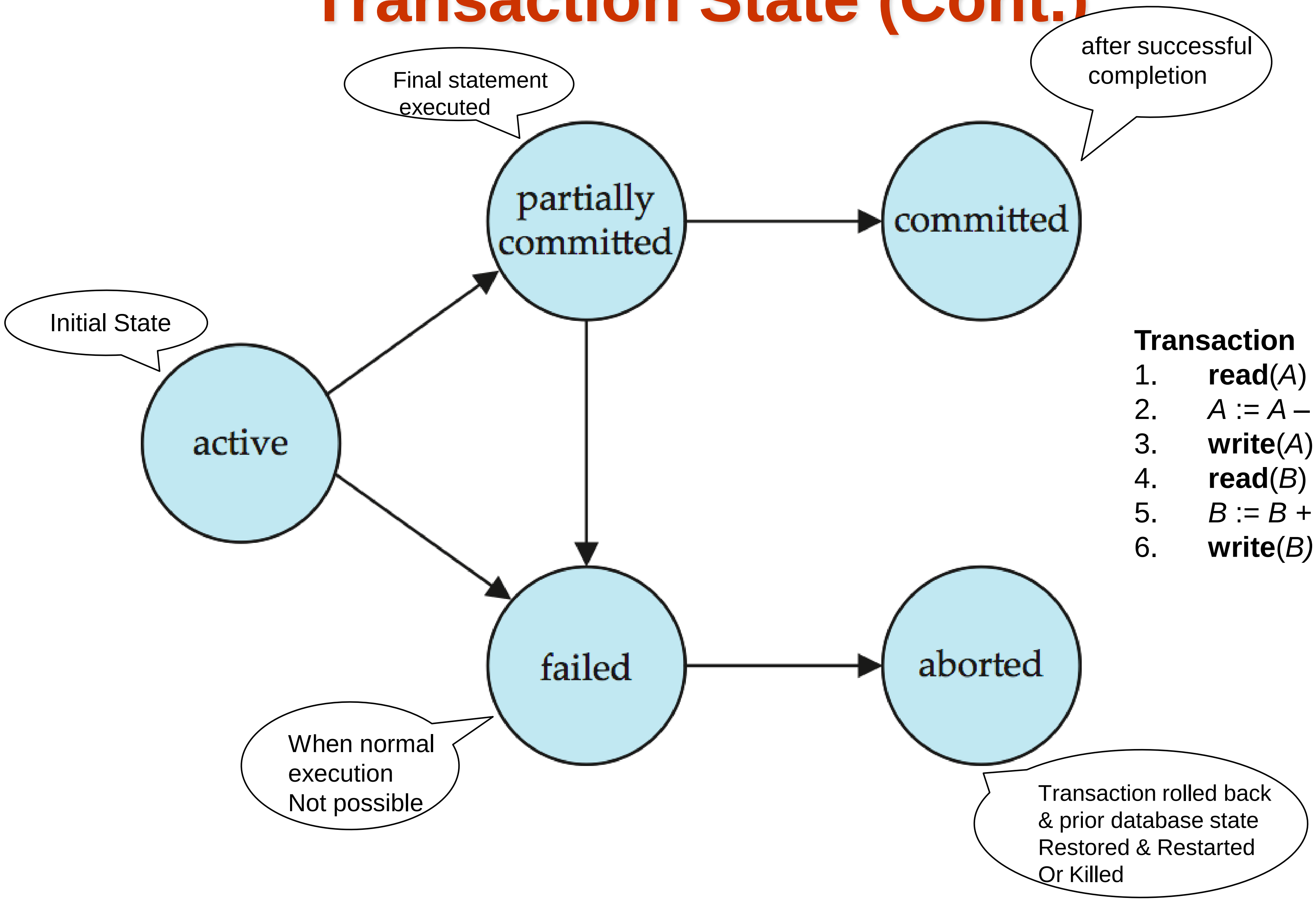
To preserve the integrity of data the database system must ensure:

- **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.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **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.
- **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
- **Committed** – after successful completion.

Transaction State (Cont.)



Transaction

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

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.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1 (Serial)

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- Suppose the current values of accounts A and B are \$1000 and \$2000.
- A **serial schedule** in which T_1 is followed by T_2 :

The final values of accounts A and B, after the execution in Schedule 1 takes place, are \$855 and \$2145, respectively.

Thus, the total amount of money in accounts A and B—that is, the **sum A + B —is preserved** after the execution of both transactions.

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

At the end of T1 A=950 and B=2050
At the end of T2 A=855 and B=2145

Total Amount in the system is 2145+855=3000
Database consistency maintained.

Schedule 2 (Serial)

- A serial schedule where T_2 is followed by T_1

Assume current value of A=1000 and B=2000

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is in schedule 2

Again, as expected, the **sum A + B is preserved**, and the final values of accounts A and B are \$850 and \$2150, respectively.

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

At the end of T2 A=900 and B=2100
At the end of T1 A=850 and B=2150

Total Amount in the system is 2150+850=3000
Database consistency maintained.

Schedule 3 (Concurrent)

- 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.

accounts *A* and *B* are $1000 and $2000.

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) commit	
	read (B) $B := B + temp$ write (B) commit

Finally A=855 and B=2145 ; A+B=855+2145=3000 –consistency maintained

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

Schedule 4 (Concurrent)

- The following concurrent schedule does not preserve the value of $(A + B)$. (Assume initially A=1000/- and B=2000/-)

	T_1	T_2	
A=1000	read (A)		
1000-50=950	$A := A - 50$		
		read (A)	A =1000
		$temp := A * 0.1$	Temp=100
		$A := A - temp$	1000-100=900
		write (A)	A=900
		read (B)	B=2000
A =950	write (A)		
B=2000	read (B)		
B=2000+50=2050	$B := B + 50$		
B=2050	write (B)		
	commit		
		$B := B + temp$	2000 +100=2100
		write (B)	B=2100
		commit	

How to check which kind of concurrent Schedule maintain Consistency

Final A=950 and B=2100, now A+B =3050 which violates Consistency

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**.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

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

- Assume a schedule ***S*** in which there are two consecutive instructions, ***I*** and ***J*** , of transactions T_i and T_j , respectively **($i \neq j$)**.
  - If ***I*** and ***J*** refer to **different data** items

    then we can swap ***I*** and ***J*** without affecting the results of any instruction
- In the schedule. However,
  - if *I* and *J* refer to the **same data item** *Q*, then the order of the two steps may matter.
- 1. I_i = **read**(*Q*), I_j = **read(Q)**. I_i and I_j don't conflict. **(can be swapped)**
  2. I_i = **read(Q)**, I_j = **write**(*Q*). They conflict. **(can't be swapped)**
  3. I_i = **write**(*Q*), I_j = **read(Q)**. They conflict **(can't be swapped)**
  4. I_i = **write**(*Q*), I_j = **write**(*Q*). They conflict **(can't be swapped)**
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - 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.

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

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

S

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

S'

S is conflict serializable

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

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 Serializability (Cont.)

- Example of a schedule that is not conflict serializable: **(schedule 7)**

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 >.

Example

T1	T2
Read(A)	
	Read(A)
Read(B)	
Write(B)	
	Read(D)
	Write(D)
Write (A)	
	Read(C)
	Write(A)

Is it Conflict Serializable?

Concurrency Control

Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

One of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.

To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency control* schemes.

Lock-Based Protocols

- One way to **ensure serializability** is to require that data items be **accessed in a mutually exclusive manner**
- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
- 1*.* ***exclusive* (X)** *mode*. If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item *Q*, then ***T****$_i$* **can both read and write *Q*.**
- 2*.* ***shared* (S)** *mode*. If a transaction T_i has obtained a **shared-mode** lock (denoted by S) on item Q, then **T$_i$ can read**, but **cannot write, Q**.
- Lock requests are **made to concurrency-control manager**.
- Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **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
- 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.

Let ***{T_0, T1, . . ., Tn}*** be a set of transactions participating in a schedule *S*. We say that *Ti* **precedes** *Tj* in *S*, written ***Ti* → *Tj*, if there exists** a data item ***Q*** such that ***Ti* has held lock mode *A (S or X)* on *Q***, and *Tj* has held **lock mode *B (S or X)*** on *Q* later, and comp(*A*,*B*) = false.

If *Ti* → *Tj* , then that precedence implies that in any equivalent serial schedule, *Ti* must appear before *Tj* .

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking (serially (T_1, T_2) OR (T_2,T_1)) :

Assume A=100 & B=200

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2*: lock-S(A)*;
read *(A);*
unlock*(A)*;
lock-S*(B)*;
read *(B);*
unlock*(B)*;
display*(A+B)*

- If the transactions are executed serially, either in the order T1, T2 or the
- order T2, T1, then transaction T2 will display the **value $300**. No Loss of Consistency.
- **If**, however, these transactions are executed concurrently, (see schedule 1 next slide) then in this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.
- if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

Assume A=100 & B=200

Schedule 1-Concurent

If, however, these transactions are executed concurrently, then schedule 1

T_1	T_2	concurrency-control manager
lock-x (B)		
		grant-x (B, T_1)
read (B) $B := B - 50$ write (B) unlock (B)		
	lock-s (A)	
		grant-s (A, T_2)
	read (A) unlock (A) lock-s (B)	
		grant-s (B, T_2)
	read (B) unlock (B) display ($A + B$)	
lock-x (A)		
		grant-x (A, T_2)
read (A) $A := A + 50$ write (A) unlock (A)		

In this case, transaction T_2 displays \$250, which is incorrect.

Reason is -T1 unlocked data item B too early.

What if hold lock for longer duration ?

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols (Deadlock)

- Consider the partial schedule

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)	

locking can lead to an undesirable situation- **Deadlock**

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
    - To handle a deadlock one of **T_3 or T_4 must be rolled back** and its locks released.

Pitfalls of Lock-Based Protocols (Deadlock)

If we do not use locking, or if we unlock data items as soon as possible after reading or writing them, we may get **inconsistent states**.

On the other hand, if we do not unlock a data item before requesting a lock on another data item, **deadlocks may occur**.

Which one to Choose ? Deadlock or getting into inconsistent state

Deadlocks are definitely **preferable to inconsistent states**, since they can be handled by rolling back of transactions, whereas

Inconsistent states may **lead to real-world problems** that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items.

Locking protocols **restrict the number of possible schedules**. The set of all such schedules is a **proper subset of all possible serializable schedules**.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for **deadlock exists in most locking protocols**. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting (T_1) for an **X-lock** on an item, while a sequence of other transactions (**T_2,$T_{3,..}$**) request and are granted an **S-lock** on the same item.
  - The same transaction (**T_1**) is **repeatedly rolled back due to deadlocks**.
- Concurrency control manager can be designed to prevent starvation
- **We can avoid starvation of transactions by granting locks in the following manner:**
- When a transaction T_i requests a lock on a data item ***Q*** in a particular mode ***M***, the concurrency-control manager grants the lock provided that:
  - **1.** There is no other transaction holding a lock on ***Q*** in a mode that conflicts with ***M***.
  - **2.** There is no other transaction that is waiting for a lock on ***Q*** and that made its lock request before T_i.

The 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
- The protocol assures serializability.
- The two-phase locking protocol ensures conflict serializability.
- Consider any transaction. The point in the schedule where the transaction has obtained its **final lock** (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points this ordering is, in fact, a serializability ordering for the transactions.

Pitfall in 2-phase Locking

In addition to being serializable, schedules should be cascade less. **Cascading rollback may occur under two-phase locking.**
consider the partial schedule given below-

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

failure of T_5 after the read(A) step of T_7

Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does* ***not* ensure freedom from deadlocks**
    - (see the schedule containing T3 & T4 slide 7)
- Cascading roll-back is possible under two-phase locking.
- To avoid this, follow a modified protocol called **strict two-phase locking**.
    - Here a transaction must hold all its exclusive locks till it commits/aborts.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

END