Database Development and Design (CPT201)

Lecture 6a: Transaction Management – Concepts of Transaction

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Transaction Concept: Atomicity, Consistency, Isolation, Durability
- Concurrent Executions and Schedule
  - Serialisability
  - Recoverability
  - Testing for Serialisability

Transaction Concept

- A transaction is a *unit* of program execution that accesses and possibly updates various data items.
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully (is committed), the database must be consistent.
  - After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Two main issues to deal with:
  - Concurrent execution of multiple transactions
  - Recovery from failures of various kinds, such as hardware failures and system crashes

ACID Properties

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

Example of Fund Transfer

- The following is a 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)

Example of Fund Transfer cont'd

- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else inconsistency will result.
- Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.

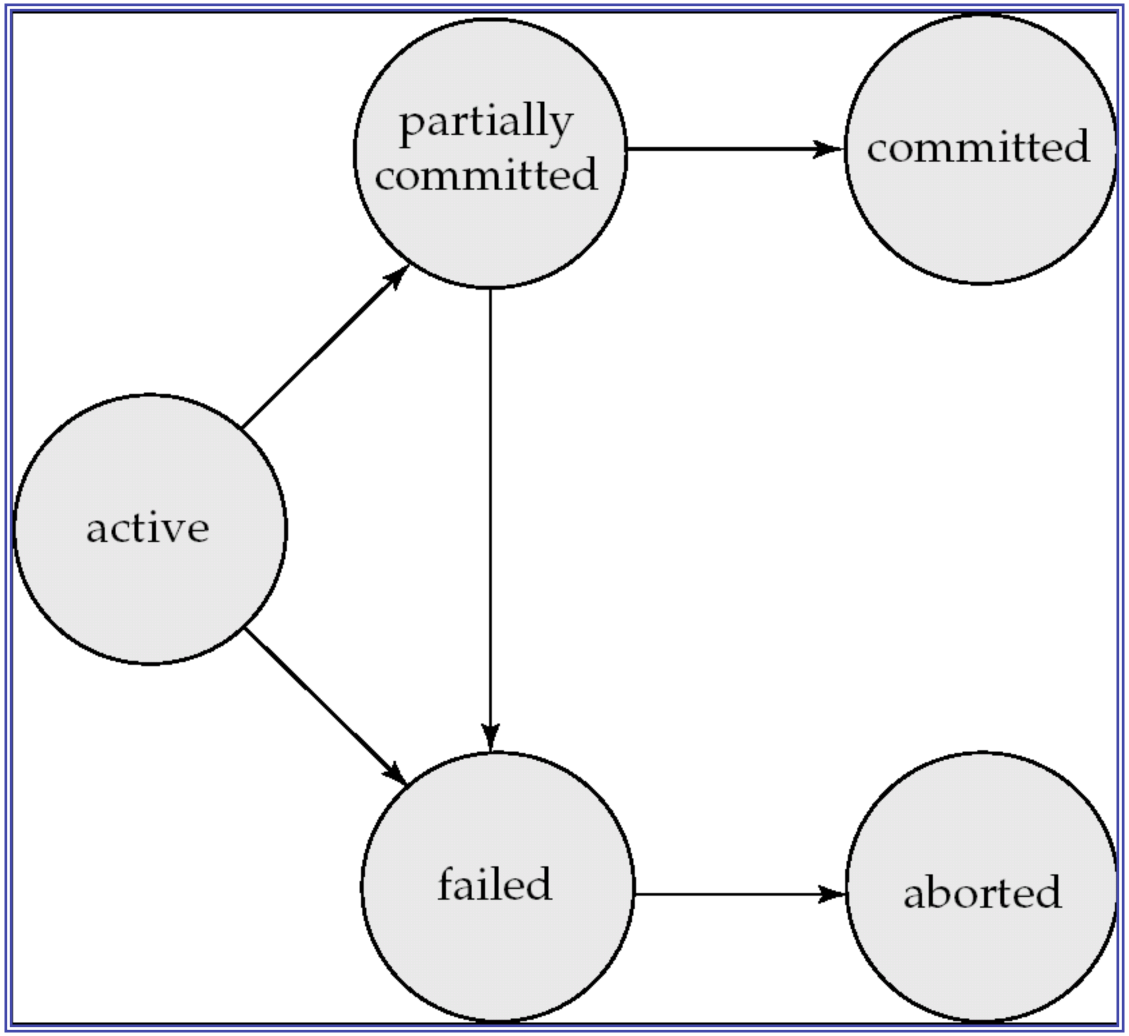
1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Example of Fund Transfer cont'd

- Isolation requirement — if between steps 3 and 6, another transaction 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).
- 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 despite failures.

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

Transaction States



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 –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.
- Committed – after successful completion.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilisation**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- 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.

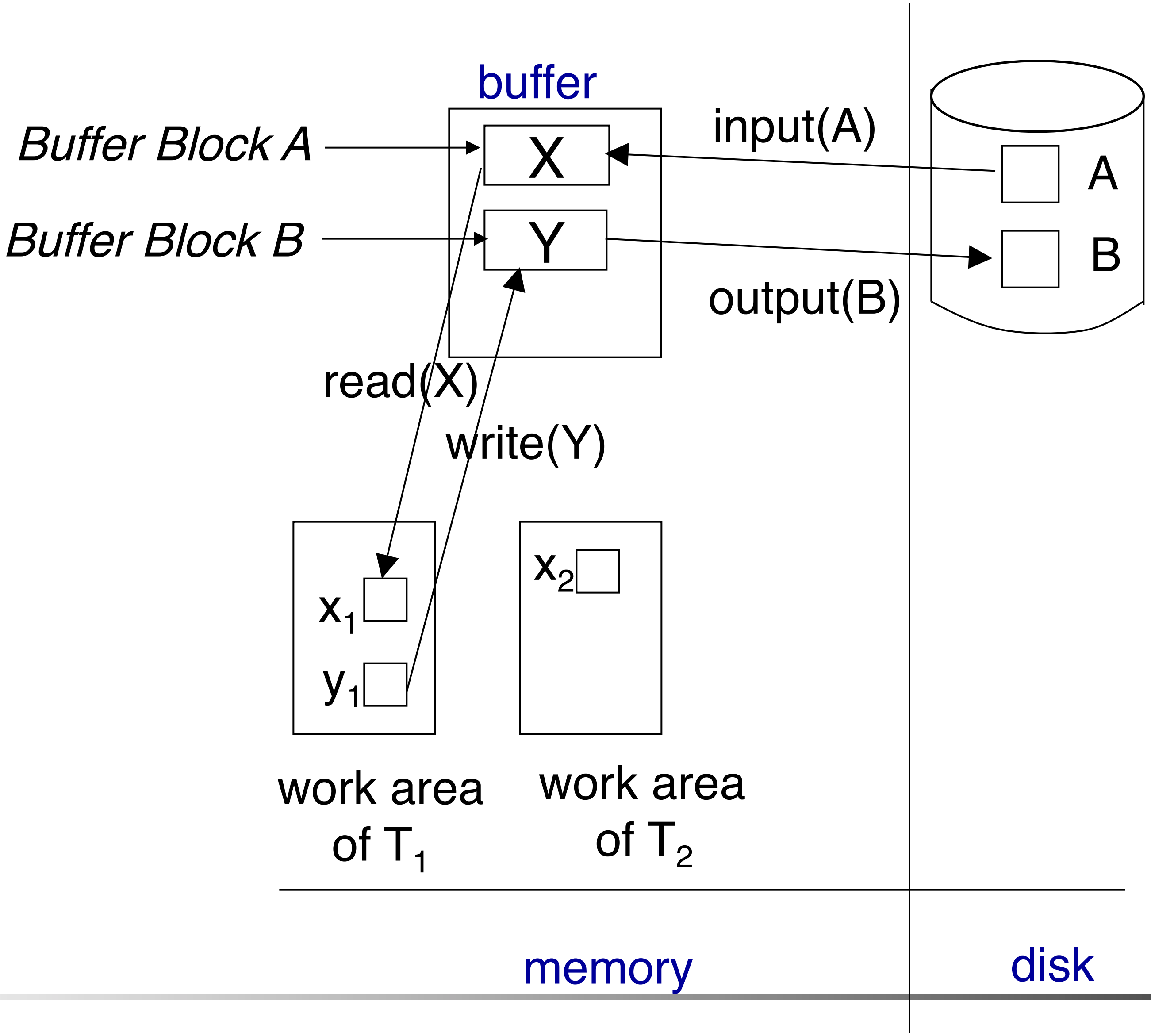
Schedules

- Schedule – a sequence of instructions that specifies the chronological order in which 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 instruction as the last statement (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement (will be omitted if it is obvious)

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory (buffer).
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Example of Schedules

Let A=1,000, B=2,000. Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.

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)

1

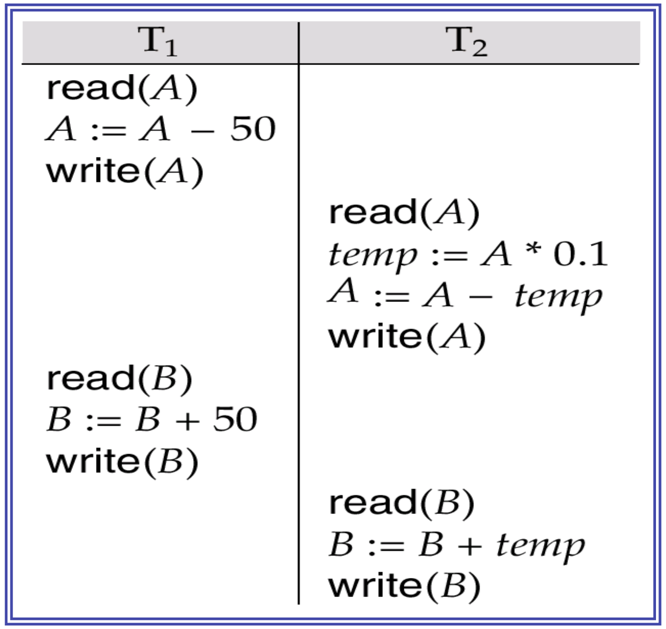
T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

2

- 1 and 2 are serial schedules;
- How much is A, B, and A+B?

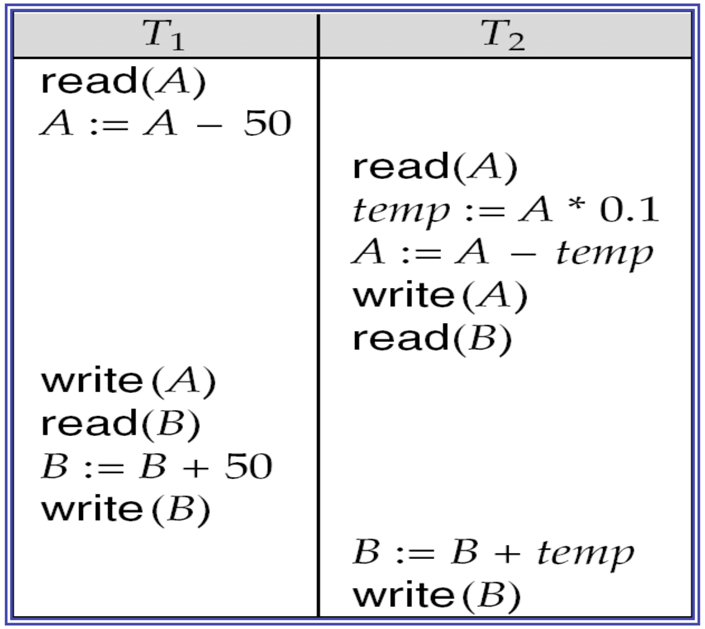
Example of Schedules cont’d

Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.



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)

3



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)

4

- 3 and 4 are not serial schedules;
- Does 3 preserve the value of (A+B)? How much is A and B?
- Does 4 preserve the value of (A+B)? How much is A and B?

Serialisability

- Basic Assumption – Each transaction preserves database consistency.
- A (possibly concurrent) schedule is serialisable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  - conflict serialisability
  - view serialisability
- We ignore operations other than read and write instructions for now for simplicity.
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j, conflict if and only if there exists some data item Q accessed by both I_i and I_j, and at least one of these instructions is write Q.
- Three conflicts can be identified
  - write-read (WR) conflict: reading uncommitted data
  - read-write (RW) conflict: unrepeatable reads
  - write-write (WW) conflict: overwriting uncommitted data
- 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 (the order of the two instructions can be swapped).

Write-Read (WR) Conflicts

- Transaction T2 reads a database object that has been modified by transaction T1 which has not committed ("dirty read")

T1	T2
read(A)	
A:= A – 50	
write(A)	
	read(A)
	A:= A + A*10%
	write(A)
	read(B)
	B:= B + B*10%
	write(B)
	commit
read(B)	
B:= B + 50	
write(B)	
Rollback	

Read-Write (RW) Conflicts

- Transaction T2 could change the value of an object that has been read by a transaction T1, while T1 is still in progress. (unrepeatable read)
- "unrepeatable read" example

T1	T2
read(*A*)	
	read(*A*)
	write(*A*)
	commit
read(*A*)	
write(*A*)	
commit	

T1	T2
read(*A*)	
	read(*A*)
	A:=A-1
	write(*A*)
	commit
read(A)	
A:=A-1	
write(*A*)	
commit	

Let A=1 and assume that there is an integrity constraint that prevents A from becoming negative. T1 will get an error.

Write-Write (WW) Conflicts

- Transaction T2 could overwrite the value of an object which has already been modified by T1, while T1 is still in progress. (Blind Write)
- "Blind Write" example
  - T1: set both Steven and Paul salaries at USD 1m.
  - T2: set both Steven and Paul salaries at RMB 1m.
- With this schedule, Steven has salary 1m RMB and Paul has salary 1m USD.

T1	T2
write(*Steven*)	
	write(*Paul*)
	write(*Steven*)
	commit
write(*Paul*)	
commit	

Conflict Serialisability

- 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. A schedule S is conflict serialisable 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)

3

$$\xrightarrow{\text{swaps of non-conflicting instructions.}}$$

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

6

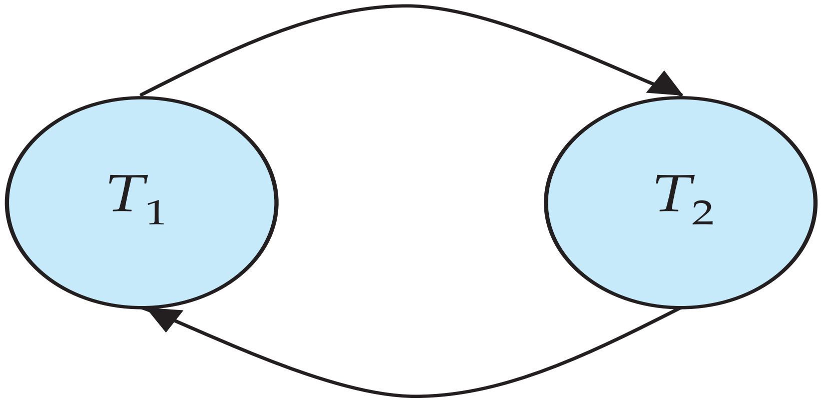
Conflict Serialisability cont'd

- Example of a schedule that is not conflict serialisable
- We are unable to swap instructions in the schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

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

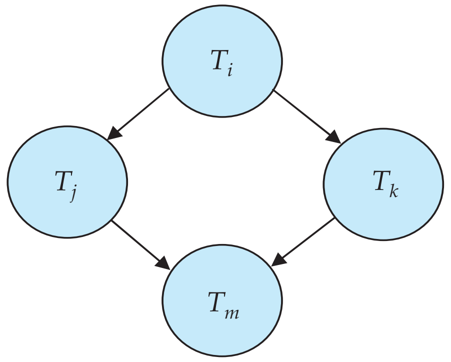
Testing for Serialisability

- Consider a schedule with a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a directed graph where
  - vertices are the transactions (names).
  - an arc (edge) is drawn from T_i to T_j if the two transactions conflict, and T_i accesses the same data item before T_j does.

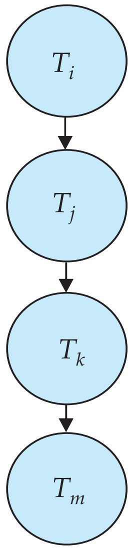


Test for Conflict Serialisability

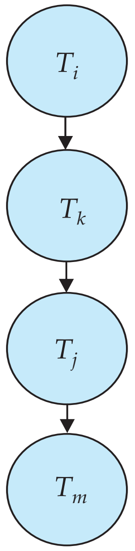
- A schedule is conflict serialisable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serialisability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - Could have more than one serialisability orders
- It is possible for two schedules to produce the same outcome, but are not conflict serialisable.



(a)



(b)



(c)

View Serialisability

- 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*,
  - 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*.
  - 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 .
  - 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'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serialisability cont'd

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

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

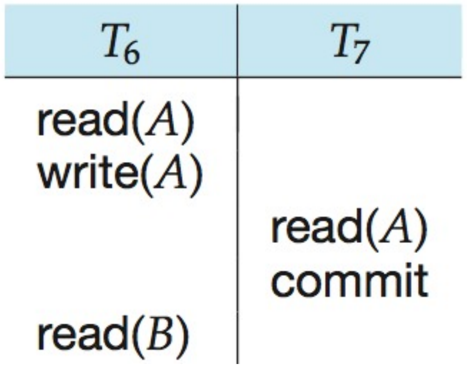
- What serial schedule is above equivalent to?
    - Answer: <T27, T28, T29>
- Every view serialisable schedule that is not conflict serialisable has **blind writes**.

Test for View Serialisability

- The precedence graph test for conflict serialisability cannot be used directly to test for view serialisability.
  - Extension to test for view serialisability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serialisable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some sufficient conditions for view serialisability can still be used.
- It is not used in practice due to its high degree of computational complexity

Recoverable Schedules

- <span>Recoverable schedule</span> — 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.
- The following schedule (Schedule 11) is not recoverable as T_7 commits immediately after the read



T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- If T_6 should abort, T_7 would have read (and possibly shown to the user) an inconsistent database state.
- Hence, database must ensure that schedules are recoverable.

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.
- If T_{10} fails (aborted later and needs to roll back), T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

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

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.

End of Lecture

- Summary
  - Transactions and the ACID properties
  - Schedules and conflict serialisable schedules
  - Potential anomalies with interleaving

- Reading
  - Textbook 6th edition, chapter 14.1, 14.2, 14.3, 14.4, 14.5, 14.6 and 14.7
  - Textbook 7th edition, chapter 17.1, 17.2, 17.3, 17.4, 17.5, 17.6 and 17.7

Database Development and Design (CPT201)

Lecture 6b: Transaction Management – Concurrency Control

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Concurrency control
  - Lock-based lock protocol
    - 2PL, strict 2PL
    - Graph-based protocols

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serialisable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serialisability *after* it has executed is too late!
- **Goal** – to develop concurrency control protocols that will assure serialisability.

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. X-lock is requested using lock-X instruction.
  - *shared (S) mode*. Data item can only be read. S-lock is requested using lock-S instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols cont’d

- 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 lock on the item no other transactions 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.

Lock-Based Protocols cont'd

- Example of a transaction performing locking:

```
T2: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
```

- 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.
- Locking as above is not sufficient to guarantee serialisability.

Pitfalls of Lock-Based Protocols

- 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)	

- 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 cont'd

- 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:
  - The most common solution to recover from deadlock is to roll back one or more transactions
  - If a transaction is repeatedly chosen as the victim, it will never complete its task, hence starvation.
- Concurrency control manager can be designed to prevent starvation.
  - The most common solution is to include the number of rollbacks in the cost factor for selecting a victim.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict serialisable 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 serialisability. It can be proved that the transactions can be serialised in the order of their lock points (i.e. the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol cont'd

- Two-phase locking *does not* ensure freedom from deadlocks
- 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.
- Rigorous two-phase locking is even stricter: here *all* locks (including the shared locks) are held till commit/abort. In this protocol transactions can be serialised in the order in which they commit.

Lock Conversions

- Refinement to increase concurrency: two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serialisability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

```
if $T_i$ has a lock on $D$ then
    read($D$)
else
    begin
        if necessary wait until no other transaction has a lock-X on $D$
        grant $T_i$ a lock-S on $D$;
        read($D$)
    end
```

Automatic Acquisition of Locks cont’d

- **write**(D) is processed as:

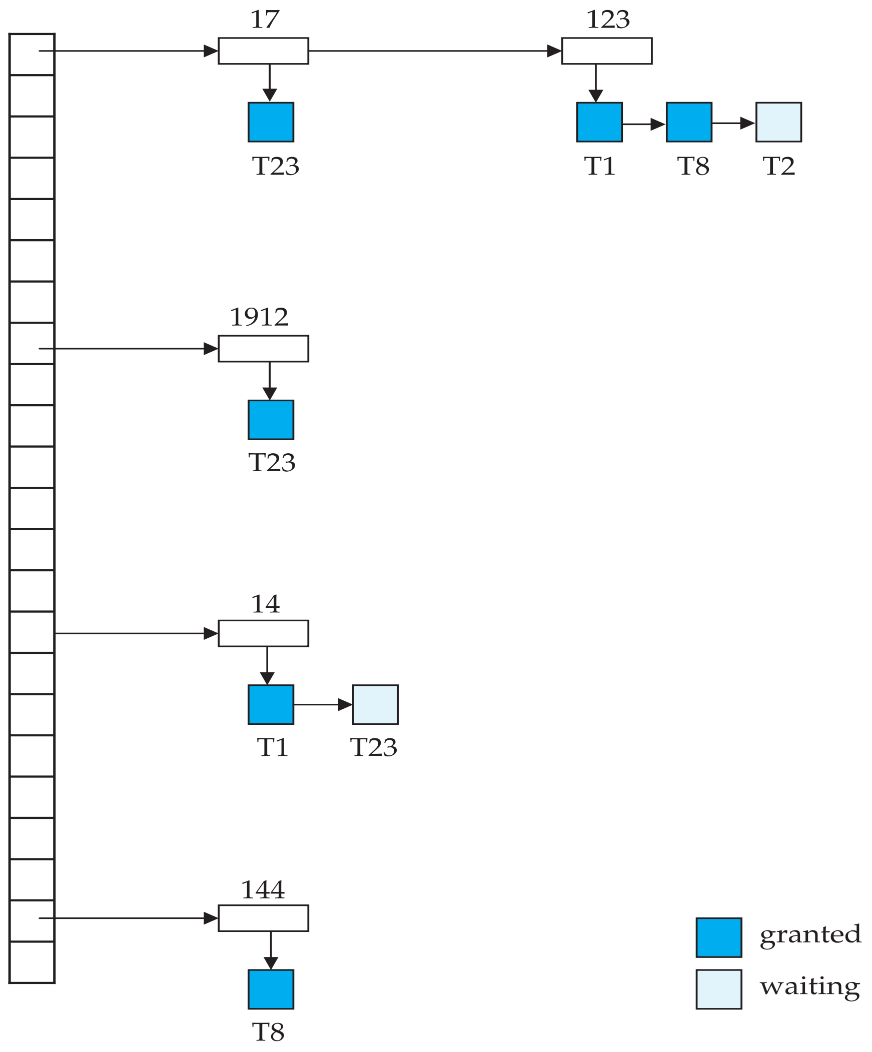
```
if $T_i$ has a lock-X on $D$ then
    write($D$)
else
    begin
        if necessary wait until no other transactions have any lock on $D$,
        if $T_i$ has a lock-S on $D$ then
            upgrade lock on $D$ to lock-X
        else
            grant $T_i$ a lock-X on $D$
        write($D$)
    end
```

- All locks are released after commit or abort

Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant message, or a message asking the transaction to roll back, in case of a deadlock
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



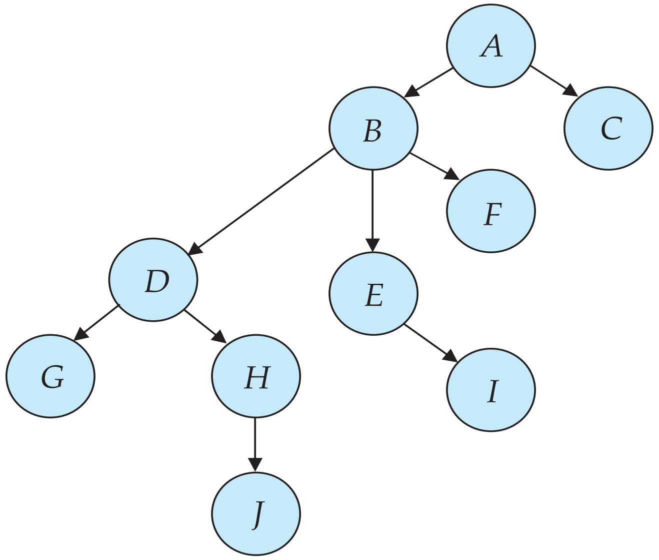
- Dark rectangles indicate granted locks, light ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



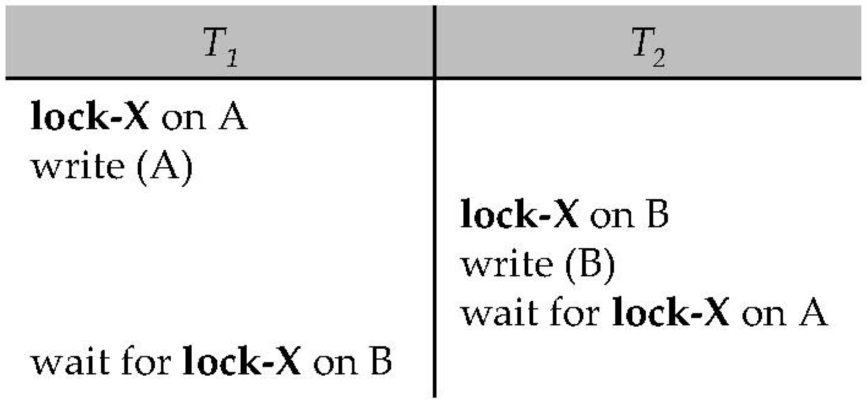
Graph-Based Protocols cont'd

- The tree protocol ensures conflict serialisability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    - shorter waiting times, and increase in concurrency
    - protocol is deadlock-free, no rollbacks are required
- Drawbacks
    - Protocol does not guarantee recoverability or cascade freedom
        - Need to introduce commit dependencies to ensure recoverability
    - Transactions may have to lock data items that they do not access.
        - increased locking overhead, and additional waiting time
        - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Deadlock Handling

- Consider the following schedule:

  T_1:write (A); T_2:write(B); T2:write(A); T1:write(B)
- Schedule with deadlock



T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling cont'd

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- The following schemes use transaction timestamps for the sake of deadlock prevention alone.
- wait-die scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- wound-wait scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

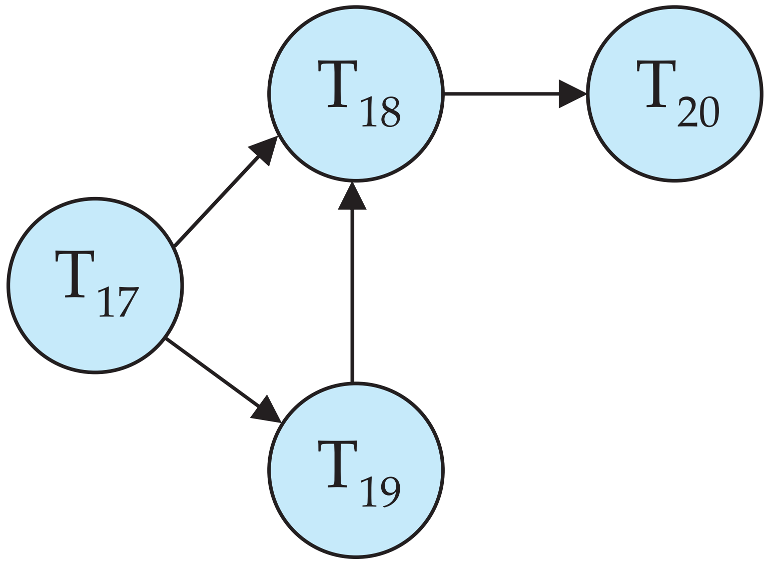
Deadlock prevention

- Both in *wait-die* and *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- But unnecessary rollbacks may occur in both schemes. Another approach is the Lock timeout-Based Schemes:
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement;
  - but starvation is possible. Also difficult to determine good value of the timeout interval.

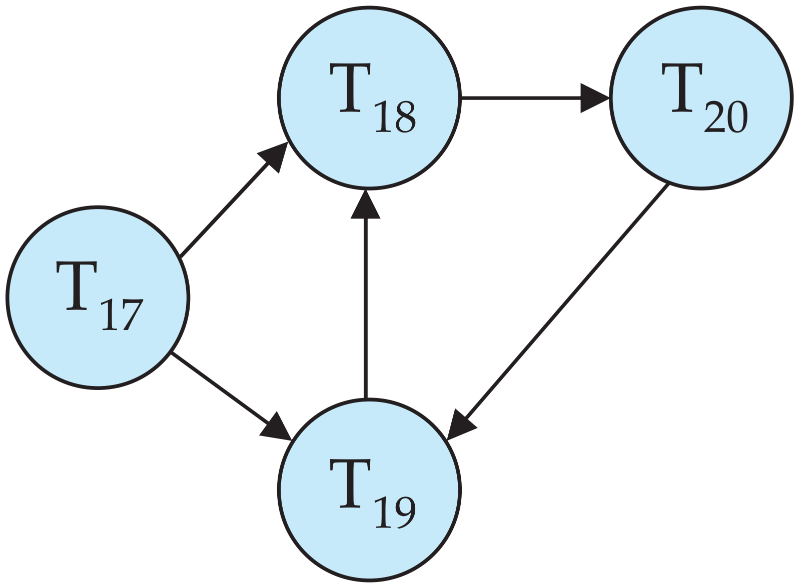
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge (T_i -> T_j) is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection cont'd



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected, three actions need to be taken :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation.

End of Lecture

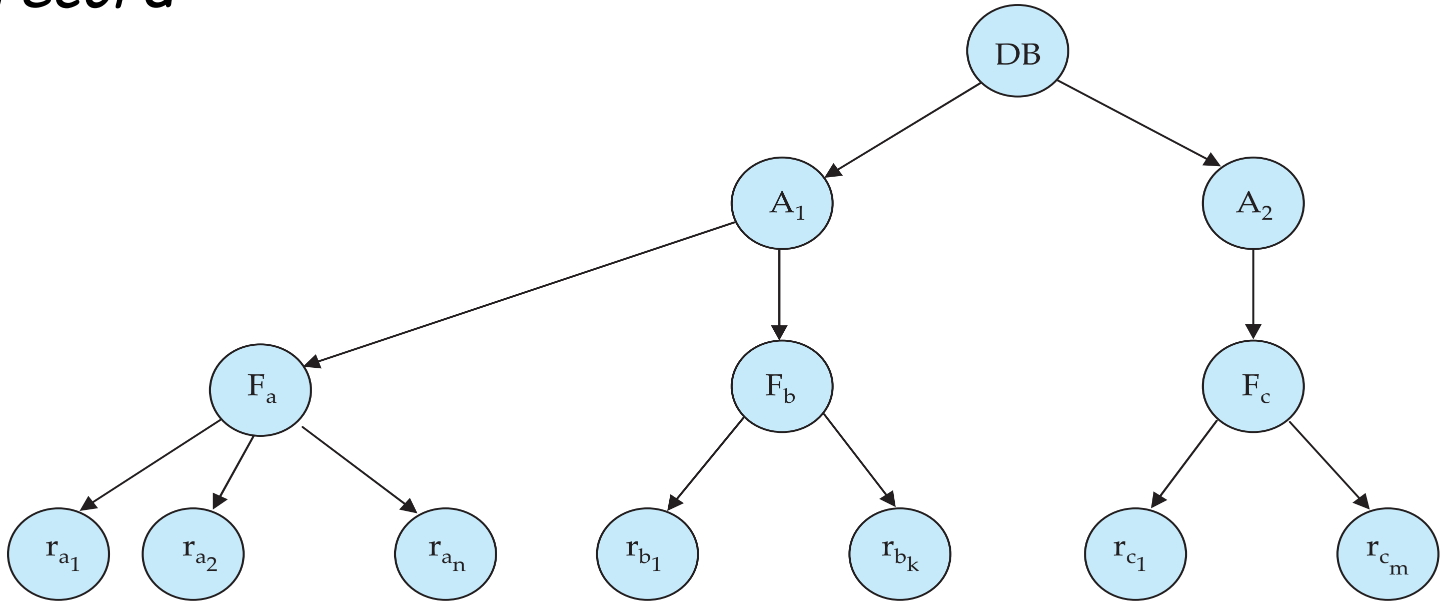
- Summary
  - Concurrency control, Lock-based lock protocol, 2PL, strict 2PL, Graph-based protocols, Deadlock prevention and detection, Starvation, etc.
- Reading
  - Textbook 6th edition, chapter 15.1, 15.2, 15.3
  - Textbook 7th edition, chapter 18.1, 18.2, 18.3

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

- The levels, starting from the coarsest (top) level are
  - *database*
  - *area*
  - *file*
  - *record*



Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- Intention locks are put on all the ancestors of a node before that node is locked explicitly.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.
  - A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  - A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  - T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  - T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- Lock granularity escalation: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Database Development and Design (CPT201)

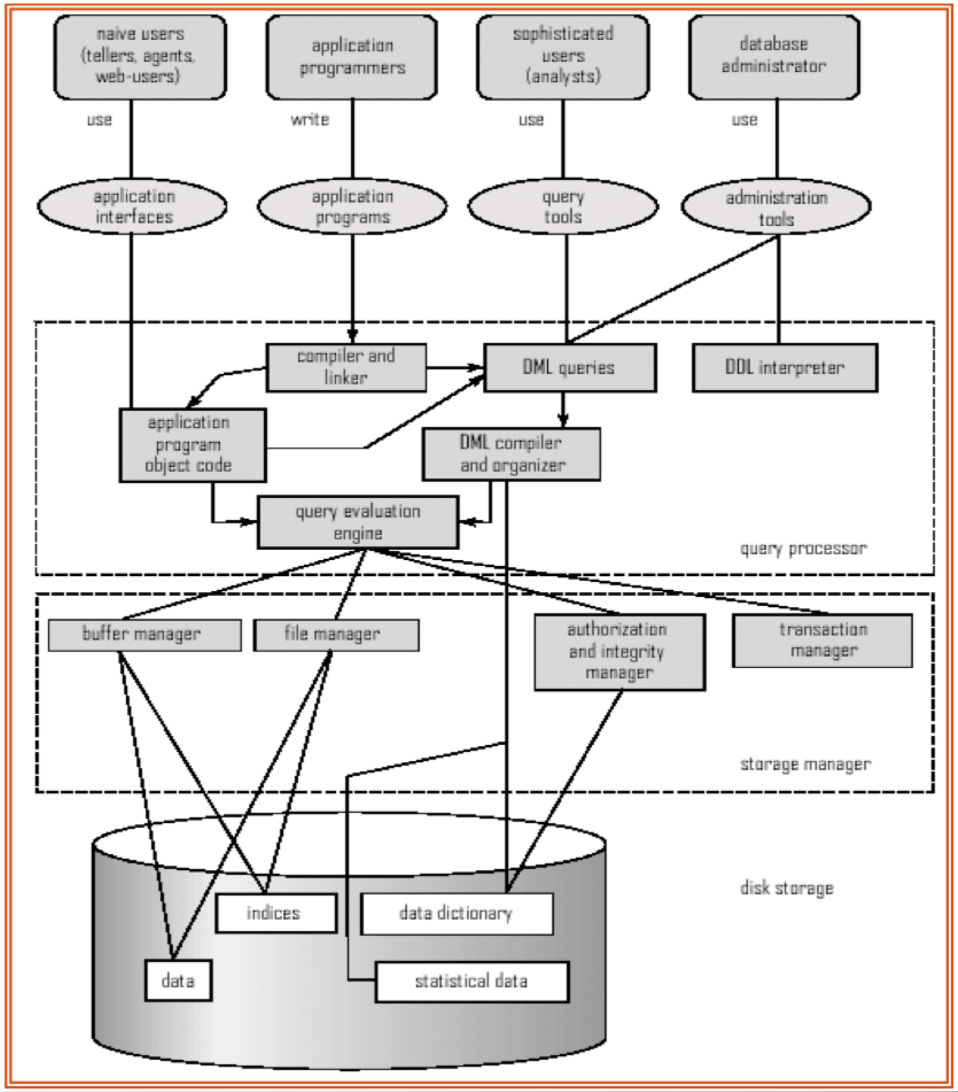
Lecture 6c: Transaction Management – Failure Recovery

Dr. Wei Wang
Department of Computing

Learning Outcomes

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery

DBMS Components revisited



Failure Classification

- **Transaction failure**:
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drivers use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to *A* and B to the database.
  - A failure may occur after one of these modifications has been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Non-volatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - usually approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies. Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    - Write the information onto the first physical block.
    - When the first write successfully completes, write the same information onto the second physical block.
    - The output is completed only after the second write successfully completes.

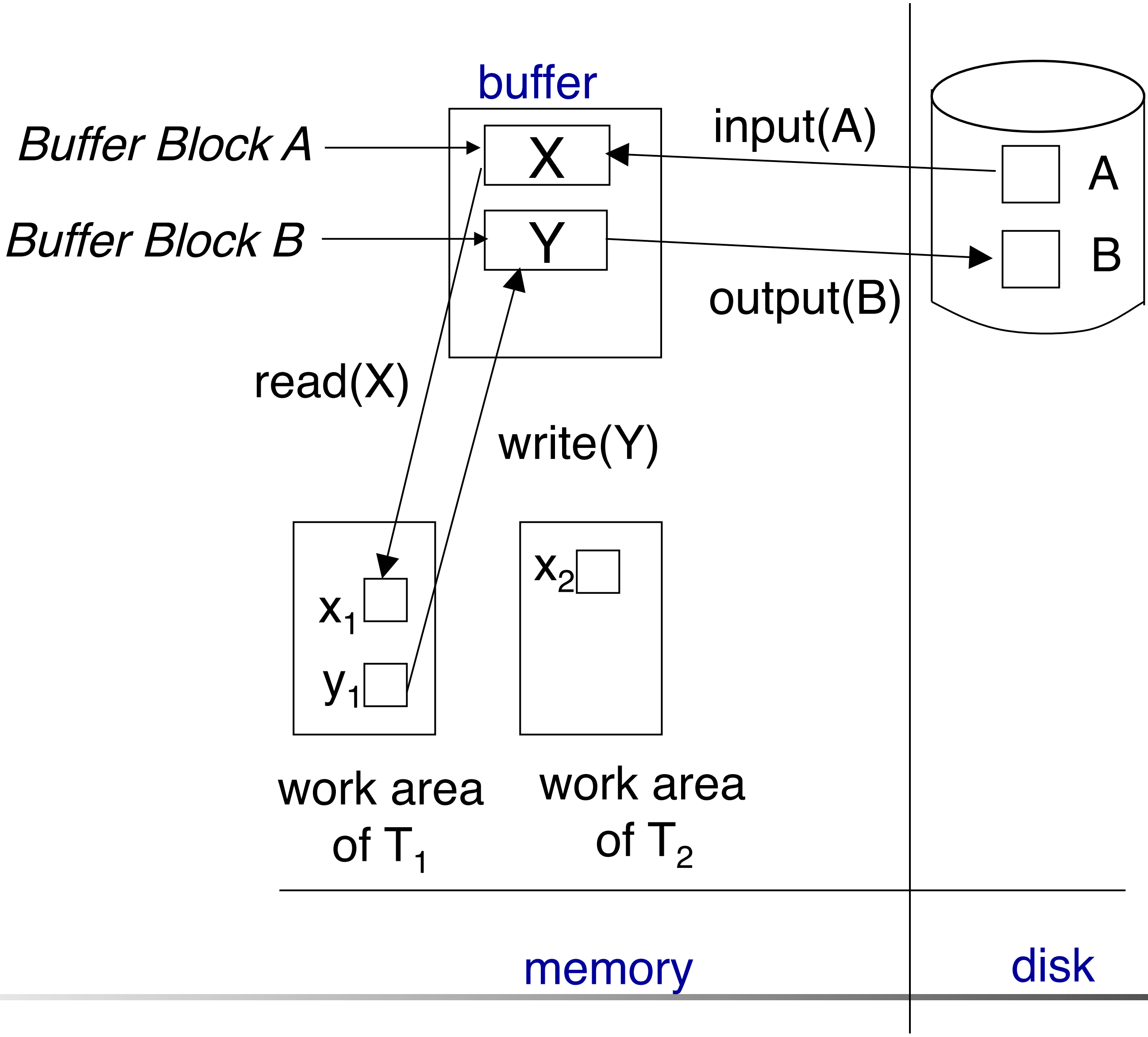
Stable-Storage Implementation cont’d

- Copies of a block may differ due to failure during output operation. To recover from failure:
  - First find inconsistent blocks:
    - *Expensive solution*: Compare the two copies of every disk block.
    - *Better solution*:
      - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
      - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
      - Used in hardware RAID systems (Redundant Arrays of Independent Disks).
  - If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block (or vice verse).

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Data Access cont'd

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Recovery and Atomicity

- To ensure atomicity despite of failures, we first output information describing the modifications (e.g., logs) to stable storage without modifying the database itself.
- log-based recovery mechanisms
  - key concepts
  - actual recovery algorithm

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.
- When transaction T_i starts, it registers itself by writing a
  <T_i **start**> log record
- *Before* T_i executes **write**(X), a log record
  <T_i, X, V_1, V_2>
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commit**> or <T_i **abort**> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate and Deferred Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits.
  - Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Output of updated blocks to stable storage can take place at any time before or after transaction commit
  - Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate and Deferred Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted, i.e. using the* strict two-phase locking protocol.
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

- Undo of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **old** value V_1 to X
- Redo of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **new** value V_2 to X (again)
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the values of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - each time a data item X is restored to its old value V, a special log record $\langle T_i , X, V \rangle$ is written out
    - when undo of a transaction is complete, a log record $\langle T_i$ **abort**$\rangle$ is written out.
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No additional logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
- Note that if transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - such a redo redoes all the original actions including the steps that restored old values
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery algorithm greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time. Assume that the failure occurs immediately after the last statement.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

- (a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out
- (b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and *C* is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.
- (c) redo (T_0) and redo (T_1): *A* and B are set to 950 and 2050 respectively. Then *C* is set to 600.

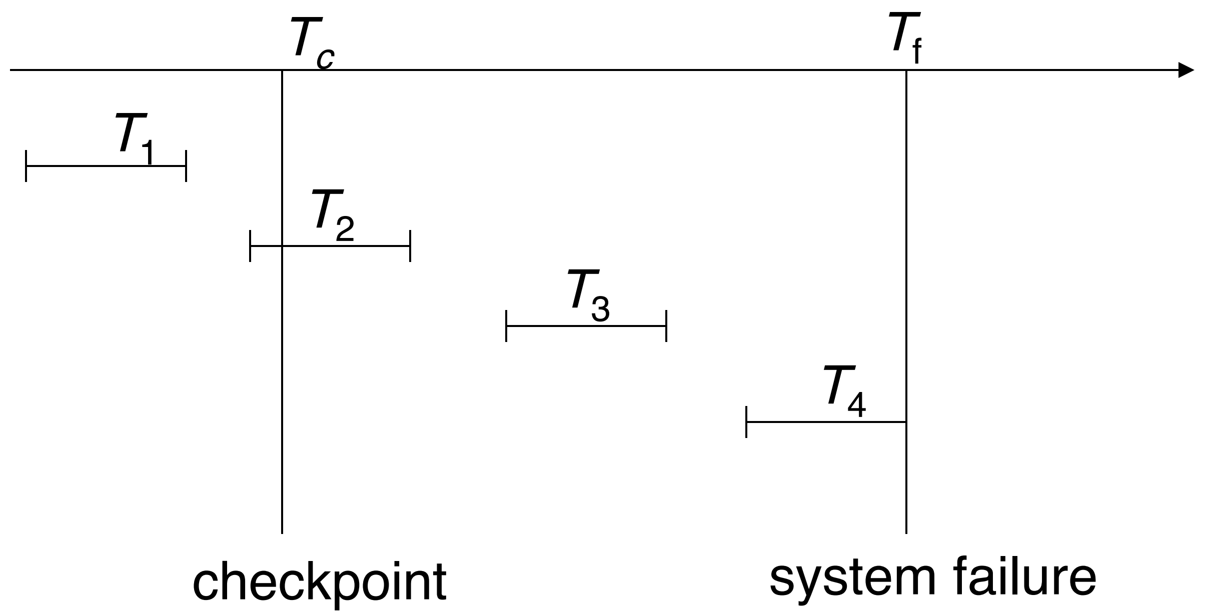
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - processing the entire log is time-consuming if the system has run for a long time
  - we might unnecessarily redo transactions which have already output their updates to the database long time ago.
- Streamline recovery procedure by periodically performing **checkpointing**
  - Output all log records currently residing in main memory onto stable storage.
  - Output all modified buffer blocks to the disk.
  - Write a log record <**checkpoint** *L*> onto stable storage where *L* is a list of all transactions which are active at the time of checkpointing.
  - All updates are stopped while doing checkpointing.

Checkpoints cont'd

- During recovery we need to consider only the most recent transactions that started before the checkpoint but not finished, and transactions that started after checkpoint.
    - Upon failure, scan backwards from end of log to find the most recent <**checkpoint** *L*> record
    - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
    - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage (no need to consider them).
- Some earlier part of the logs may be needed for undo operations
    - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
    - Parts of log prior to the earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone (but all instructions in T4 up to the failure point need to be redone)

Recovery Algorithm

- Logging (during normal operation)
  - $<T_i$ **start**$>$ at transaction start
  - $<T_i, X_j, V_1, V_2>$ for each update, and
  - $<T_i$ **commit**$>$ at the end of transaction
- Transaction rollback (during normal operation)
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$
    - perform the undo by writing V_1 to X_j,
    - write a log record $<T_i , X_j, V_1>$
      - such log records are called **compensation log records**
  - Once the record $<T_i$ **start**$>$ is found stop the scan and write the log record $<T_i$ **abort**$>$

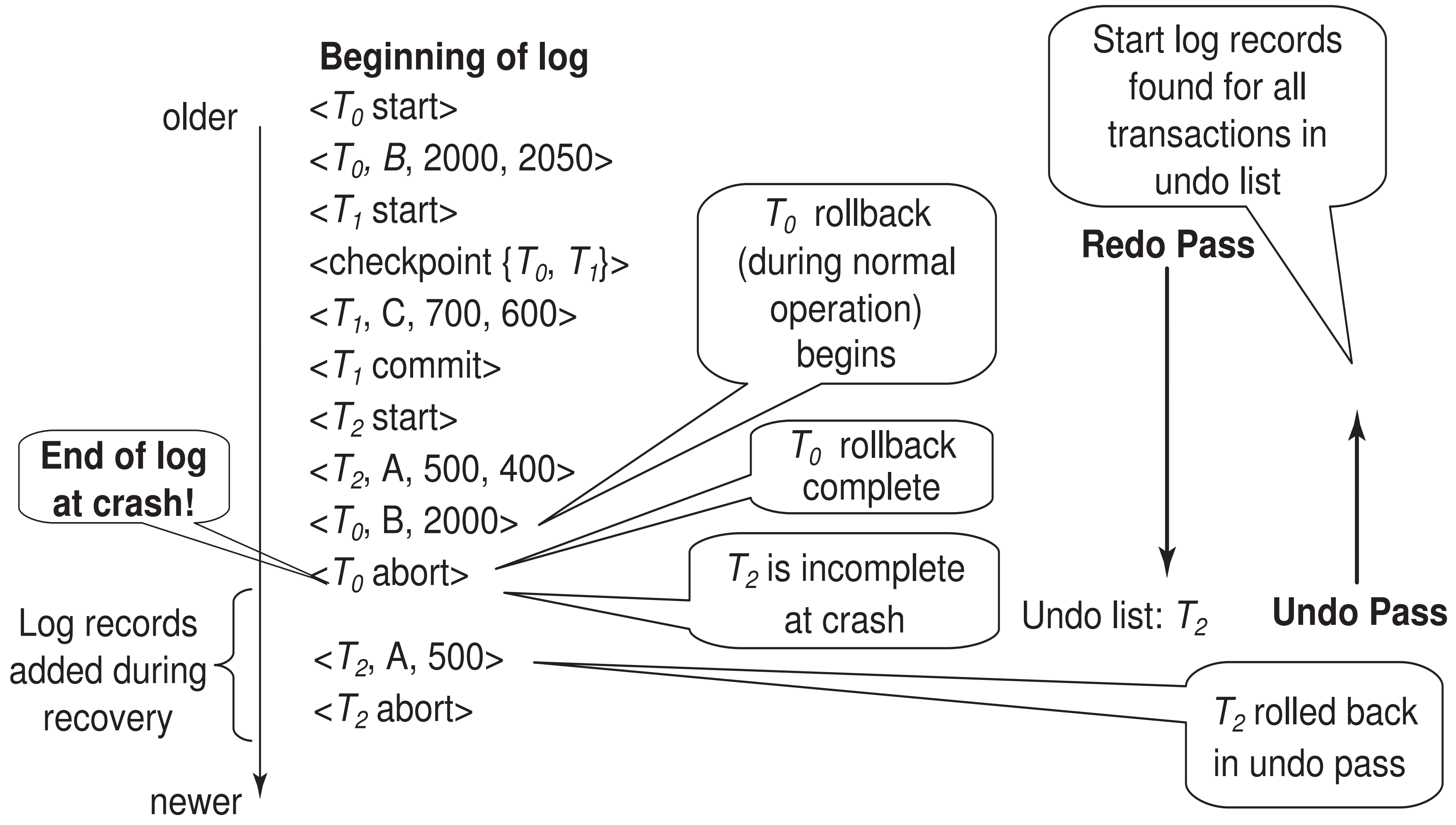
Recovery Algorithm cont'd

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete, at and after checkpoint
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  - Find last <**checkpoint** *L*> record, and set the undo-list to *L* (undo-list = L).
  - Scan forward from above <**checkpoint** *L*> record
    - Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
    - Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
    - Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recovery Algorithm cont'd

- **Undo phase:**
  - Scan log backwards from the failure point
    - Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list, perform same actions as for transaction rollback:
      - perform undo by writing V_1 to X_j.
      - write a log record $<T_i , X_j, V_1>$
    - Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,
      - Write a log record $<T_i$ **abort**$>$
      - Remove T_i from undo-list
    - Stop when undo-list is empty
      - i.e., $<T_i$ **start**$>$ has been found for every transaction in undo-list
  - After undo phase completes, normal transaction processing can commence again.

Example of Recovery



End of Lecture

- Summary
  - Failure Classification
  - Storage Structure
  - Recovery and Atomicity
  - Log-Based Recovery

- Reading
  - Textbook 6th edition, chapter 16.1, 16.2, 16.3, 16.4
  - Textbook 7th edition, chapter 19.1, 19.2, 19.3, 19.4