Database Management Systems

Module 5: Transaction Management

School of Computer Engineering and Technology

Module 3- Transaction Management and Concurrency Control

- **Transaction Management(2)-ACID properties, transactions, schedules and concurrent execution of transactions,**
- Serializability(2) : View, Conflict , Cascade-less Schedule, Recoverable Schedule
- Concurrency control(2)- lock based protocol(simple,2 phase: Rigorous 2 phase, Strict 2 phase)
- Deadlocks(1): Prevention Techniques(Wait Die, Wound Wait), Detection Techniques
- Database Recovery(2),Failure classification Recovery and atomicity: Log-based recovery, Shadow paging

What is a Transaction

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

Set of Transaction Commands

- **Start**
- **Read**
- **Write**
- **Operation**
- **End**
- **Commit/Rollback**

Two main issues to deal with:

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

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)

Questions

- Write a Transaction to add 10% interest on current balance of bank account A,

Transaction to add 10% interest on current balance of bank account A
1. **read**(*A*) 2. $A := A + A*0.1$ 3. **write**(*A*)

Transaction Properties : A C I D

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

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)

The system should ensure that updates of a partially executed transaction are not reflected in the database

Transaction Properties : A C I D

Consistency requirement-

- ✔ The sum of A and B is unchanged by the execution of the transaction
- ✔ During transaction execution the database may be temporarily inconsistent.

Transaction to transfer $50 from account A to account B: A=1000, B=1000 A+B=?
1. read(A) 2. A := A – 50 3. write(A) 4. read(B) 5. B := B + 50 6. write(B)
After Transaction A=? , B=?, A+B=?

Database must be in consistent state before and after execution of transaction

Transaction Properties : A C I D

Isolation requirement-

if between steps 3 and 6 (of the fund transfer transaction) , 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).

Transaction to transfer $50 from account A to account B:
A=1000, B=1000

T1	T2
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*)	

In Transaction T2
A=? , B=?, A+B=?

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

Transaction Properties : A C I D

- **Durability requirement**-

  Once the transfer of the $50 has taken place, and transaction is committed , despite of failure database persists its state .

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)

Once transaction has completed, the updates made to the database by the transaction must persist even if there are software or hardware failures.

Transaction Properties : Recap

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 States

Active – the initial state; the transaction stays in this state while it is executing

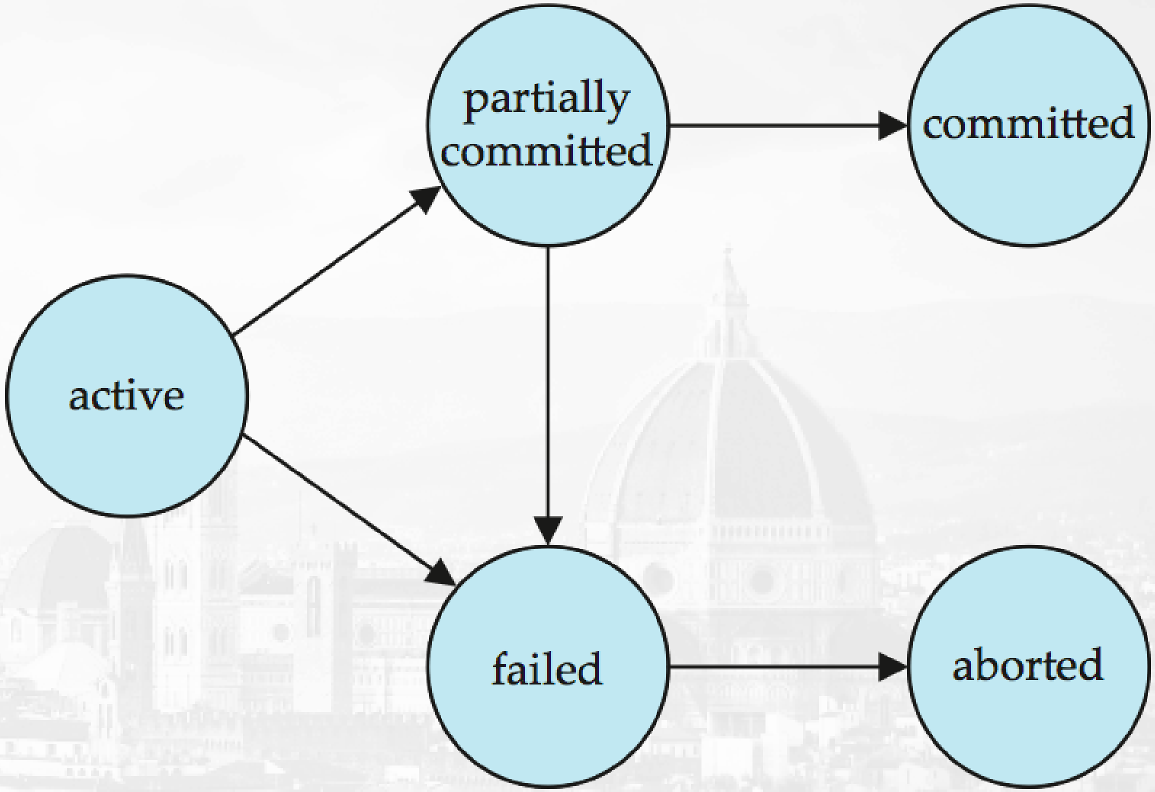
Partially committed – after the final statement has been executed.

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
- kill the transaction

Committed – after successful completion.



Module 3- Transaction Management and Concurrency Control

- Transaction Management(2)-ACID properties, transactions, **schedules and concurrent execution of transactions**
- **Serializability(2) : View, Conflict , Cascade-less Schedule, Recoverable Schedule**
- Concurrency control(2)- lock based protocol(simple,2 phase: Rigorous 2 phase, Strict 2 phase)
- Deadlocks(1): Prevention Techniques(Wait Die, Wound Wait), Detection Techniques
- Database Recovery(2),Failure classification Recovery and atomicity: Log-based recovery, Shadow paging

Concurrent Transaction Executions

Advantage

- ✔ **increased processor and disk utilization**, leading to better transaction throughput

  E.g. 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.

Issues

Concurrent Transaction Executions may destroy the consistency of the database

Solution

- **Serializability**
- **Concurrency control schemes**
  - ✔ Mechanisms to achieve isolation.
  - ✔ To control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Serializability

- Schedules
- Serializability
- Types of Serializability
- Serializability Checking

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.

Schedules

Serial Schedule

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

Concurrent Schedule

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

Question : Identify the type of Given Schedule

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

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	

Question : Check the consistency property of given schedule [A=1000, B=1000]

S1

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

S2

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	

Question : Check the consistency property of given schedule [A=1000, B=1000]

S1

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

S2

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

Serializability

- **Basic Assumption –** Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency.

- A 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

- If a concurrent schedule is serializable it preserves the consistency of database.

Conflict Serializability

- We say that a schedule S is **conflict serializable** if it is **conflict equivalent** to a serial schedule
- If a schedule S(CS) can be transformed into a schedule S´(SS) by a series of swaps of non-conflicting instructions, we say that S and S´ are **conflict equivalent**.
- If a schedule S(CS) is **conflict equivalent then its Conflict Serializable.**
- **Conflict Serializable Schedules preserve consistency of database**

Conflict Serializability : Identify pairs of Conflict Instructions

What is Conflicting Instruction??

1. **If I1, I2 are of different Transaction**
2. **If I1, I2 access same data object**
3. **If one of them is write instruction**

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**

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

Conflict Serializability : Check the Conflict serializablity of given Schedule

Hint : If a schedule S(CS) can be transformed into a Serial schedule S´(SS) by a series of swaps of non-conflicting instructions, we say that S and S´ are conflict equivalent. And S is Conflict Serializable

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

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

Conflict Serializability : Check the Conflict serializablity of given Schedule

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

It is non-conflict Serializable

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 >.

Conflict Serializability : Check the Conflict serializablity of given Schedule

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

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

Precedence Graph : Method to check Conflict Serializability

A directed graph, called a precedence graph, from S. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.

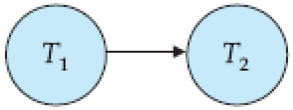
The set of vertices consists of all the transactions participating in the schedule.
The set of edges consists of all edges
$T_i \rightarrow T_j$ for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q)
2. Ti executes read(Q) before Tj executes write(Q)
3. Ti executes write(Q) before Tj executes write(Q)

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S equivalent to S, Ti must appear before Tj .
If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

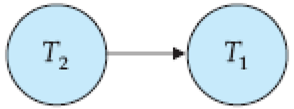
Precedence Graph : Example

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



(a)

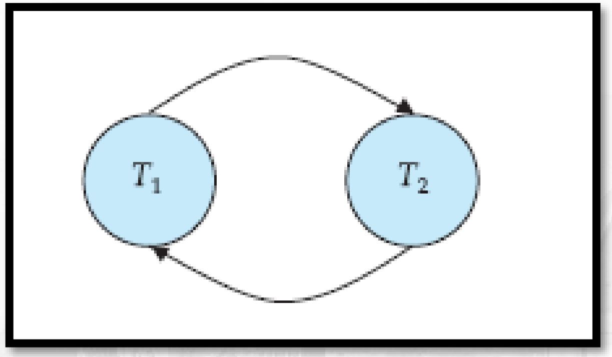
T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	



(b)

Precedence Graph : Check conflict Serializability

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



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 Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.
2. If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .
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'.

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

Sequence of First Read, R-W and Final Write should be same in both Schedules

View Serializability : Check the View of given Schedule

Hint: Sequence of First Read, R-W and Final Write should be same in both Schedules

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

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

Recoverable Schedule

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 **must** appear before the commit operation of T_j.

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

Is the Shown Schedule Is Recoverable??

The following schedule is not recoverable as T_9 commits immediately after the read(A) operation

How to Make it Recoverable??

T_8 commits must commit before T9

Cascading Rollback

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks.

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

Is the Shown Schedule held cascading rollback ??

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

How to Make it Cascadeless??

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.

Module 3- Transaction Management and Concurrency Control

- Transaction Management(2)-ACID properties, transactions, schedules and concurrent execution of transactions,
- Serializability(2) : View, Conflict , Cascade-less Schedule, Recoverable Schedule
- **Concurrency control(2)- lock based protocol(simple,2 phase: Rigorous 2 phase, Strict 2 phase)**
- Deadlocks(1) : Prevention Techniques(Wait Die, Wound Wait), Detection Techniques
- Database Recovery(2),Failure classification Recovery and atomicity: Log-based recovery, Shadow paging32

Lock-Based Protocols

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. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. ***shared*** (S) mode. Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to the concurrency-control manager by the programmer.

Transaction can proceed only after request is granted.

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.

Lock Compatibility

Lock-compatibility matrix

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.

	S	X
S	true	false
X	false	false

Lock-based Protocol : Convert Following in Lock-Based Protocol

read *(A)*;

read *(B)*;

display*(A+B)*

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

Pitfalls of Lock-Based Protocols

Consider the partial schedule

Neither T3 nor T4 can make progress — executing lock-X(B) causes T4 to wait for T3 to release its lock on B, while executing lock-S(A) causes T3 to wait for T4 to release its lock on A.

Such a situation is called a deadlock. To handle a deadlock one of T3 or T4 must be rolled back and its locks released.

T3	T4
lock-X(*B*) **read**(*B*) *B:- B-50* **write**(*B*)	
	lock-S(*A*) **read**(*A*) **lock-S**(*B*)
lock-X(*A*)	

The Two-Phase Locking Protocol

This protocol 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

Two Phase Locking Protocol : Convert Following in Two Phase Locking Protocol

read *(A)*;

read *(B)*;

display*(A+B)*

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

Strict Two Phase, Rigorous Two Phase

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 are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Convert Following in Strict Two Phase Locking Protocol

read *(A)*;
read *(B)*;
Write(B)
read(C)

display*(A+B+C)*
Commit

lock-S(A);
read (A);
lock-X(B);
read (B);
Write(B)
lock-S(C);
read(C)
unlock(A);
display(A+B+C)
unlock(C);
Commit
unlock(B);

Convert Following in Rigorous Two Phase Locking Protocol

read *(A)*;
read *(B)*;
Write(B)
read(C)

display*(A+B+C)*
Commit

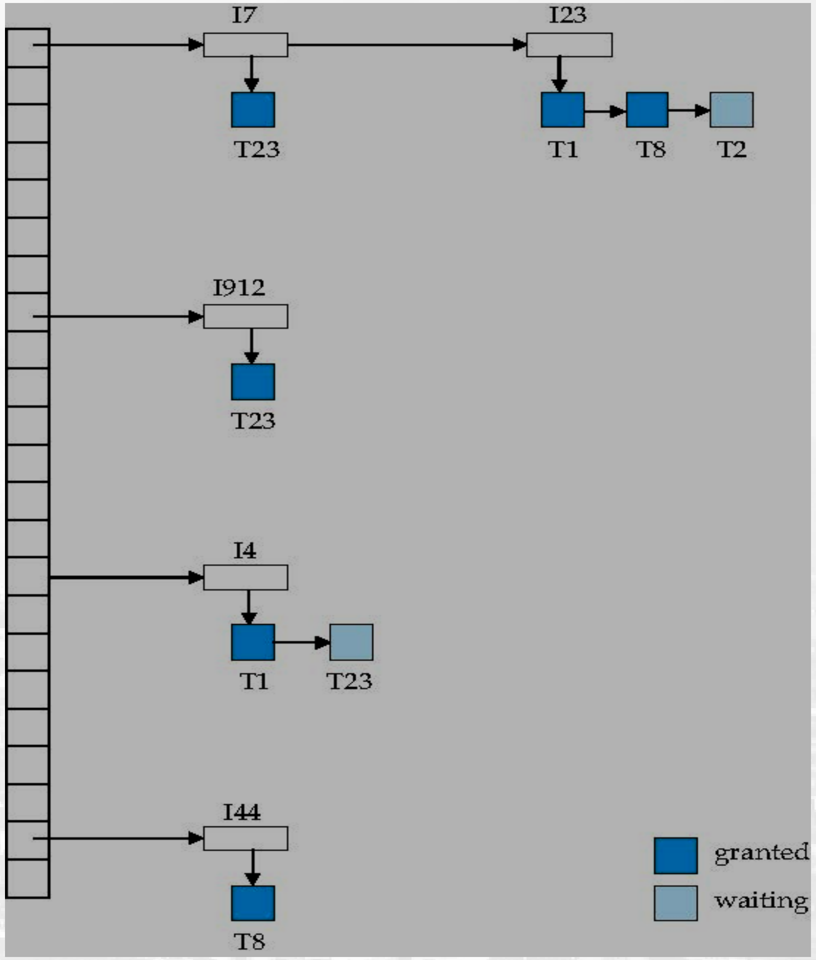
lock-S(A);
read (A);
lock-X(B);
read (B);
Write(B);
lock-S(C);
read(C)
display(A+B+C)
Commit
unlock(A);
unlock(B);
unlock(C);

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 messages (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 blue rectangles indicate granted locks; light blue 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



Module 3- Transaction Management and Concurrency Control

- Transaction Management(2)-ACID properties, transactions, schedules and concurrent execution of transactions,
- Serializability(2) : View, Conflict , Cascade-less Schedule, Recoverable Schedule
- Concurrency control(2)- lock based protocol(simple,2 phase: Rigorous 2 phase, Strict 2 phase)
- **Deadlocks(1) : Prevention Techniques(Wait Die, Wound Wait), Detection Techniques**
- Database Recovery(2),Failure classification Recovery and atomicity: Log-based recovery, Shadow paging

Deadlock

T1	T2
lock-X on *X* write (*X*)	
	lock-X on *Y* write (*X*) wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

Deadlock Handling

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.

wait-die scheme — non-preemptive

- Older transaction may wait for younger one to release data item. (older means smaller timestamp) 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.

Consider following Deadlock Scenario: Apply Wait Die

	T1	T2
1 2	**lock-X** on *X* write (*X*)	
3 4 5		**lock-X** on *Y* write (*X*) wait for **lock-X** on *X*
6	wait for **lock-X** on *Y*	

T2 Will not wait for T1,
It Die itself [Rollback T2]

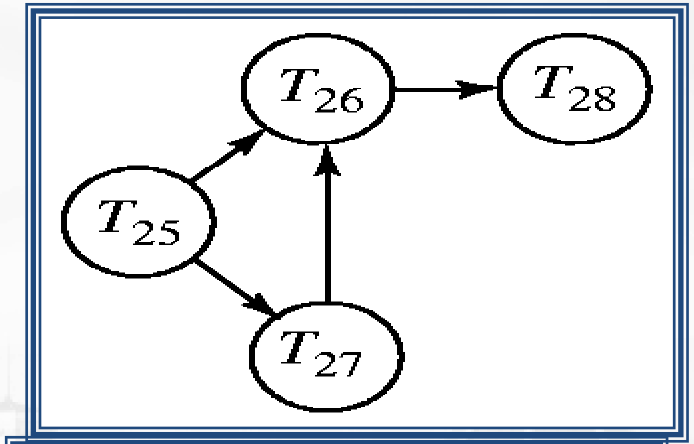
Consider following Deadlock Scenario: Apply Wound Wait

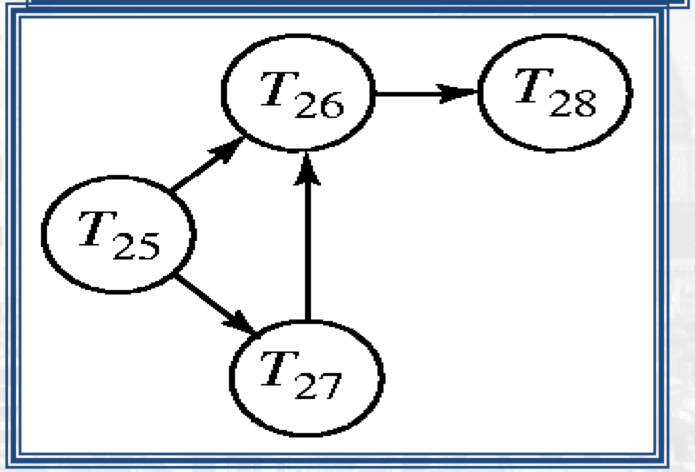
	T1	T2
1 2	**lock-X** on *X* write (*X*)	
3 4 5		**lock-X** on *Y* write (*X*) wait for **lock-X** on *X*
6	wait for **lock-X** on *Y*	

T1 Forces T2 to Rollback.

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.**





Module 3- Transaction Management and Concurrency Control

- Transaction Management(2)-ACID properties, transactions, schedules and concurrent execution of transactions,
- Serializability(2) : View, Conflict , Cascade-less Schedule, Recoverable Schedule
- Concurrency control(2)- lock based protocol(simple,2 phase: Rigorous 2 phase, Strict 2 phase)
- Deadlocks(1) : Prevention Techniques(Wait Die, Wound Wait), Detection Techniques
- **Database Recovery(2),Failure classification Recovery and atomicity: Log-based recovery, Shadow paging**

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. It is assumed that non-volatile storage contents are not corrupted.

- **Disk failure:** a head crash or similar failure destroys all or part of disk storage

Recovery Systems

Suppose transaction Ti transfers $50 from account A to account B

Two updates: subtract 50 from A and add 50 to B

Transaction Ti requires updates to A and B to be output to the database.

A failure may occur after one of these modifications have been made but before both of them are made.

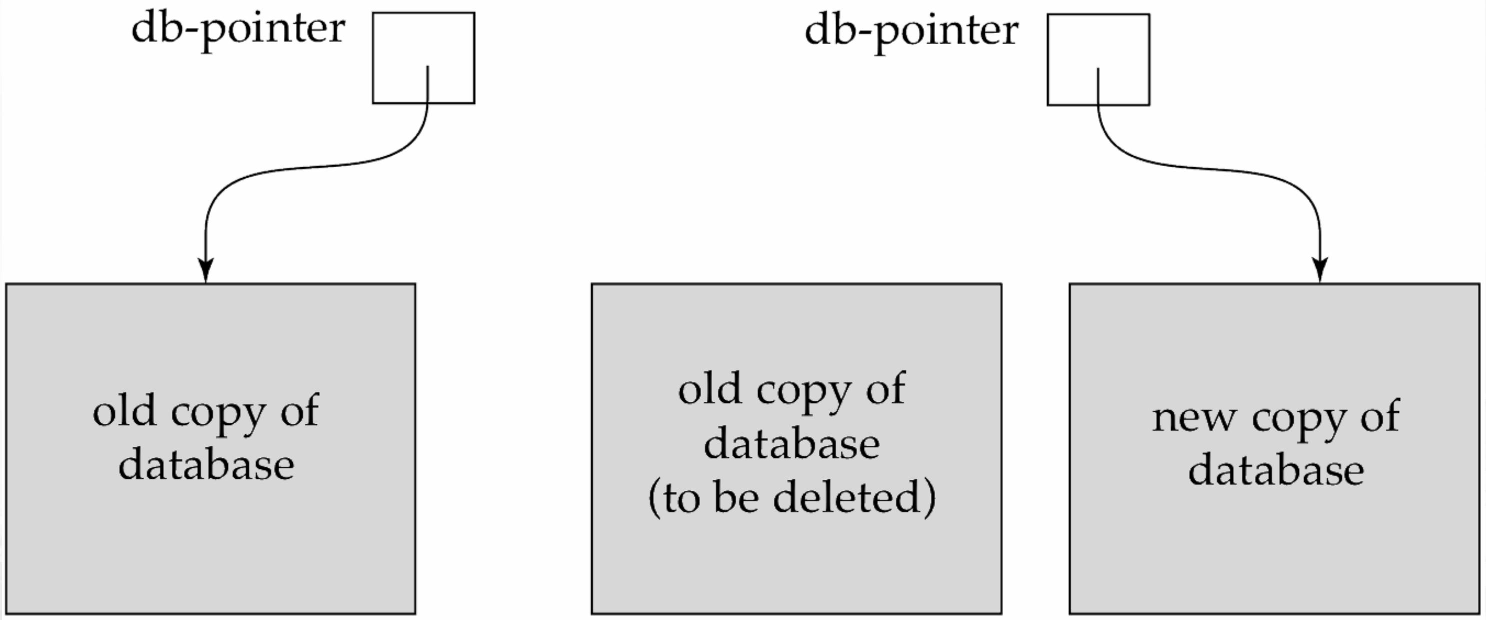
Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state

Not modifying the database may result in lost updates if failure occurs just after transaction commits

Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery & Atomicity : Shadow Paging



(a) Before update

(b) After update

Log-Based Recovery

- A **log** is a sequence of **log records**. The records keep information about update activities on the database. The **log** is kept on stable storage
- 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**> is written.
- Two approaches using logs
  - ✔ Immediate database modification
  - ✔ Deferred database modification.

What is Log Record for Following Transaction

A=100, B=100, C=100

read *(A)*;
read *(B)*;
B=B+10
Write(B)
Read(c)

display*(A+B+C)*
Commit

<T_i **start**>

<T_i, *B, 100, 110*>

<*Ti commit*>

Immediate 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

A=100, B=100, C=100

read *(A)*;
read *(B)*;
B=B+10
Write(B)
Read(c)
display*(A+B+C)*
Commit

Log

<T_i **start**>

<T_i, B, 100, 110>

<Ti commit>

Disk

B=110

Immediate Database Modification : Recovery

- **Recovery procedure has two operations instead of one :**
  - **undo(Ti)** restores the value of all data items updated by Ti to their old values, going backwards from the last log record for Ti
  - **redo(Ti)** sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti
- **When recovering after failure:**
  - Transaction Ti needs to be undone if the log contains the record < Ti start>, but does not contain <Ti Commit> the record .
  - Transaction Ti needs to be redone if the log contains both the record < Ti start> and the record <Ti Commit> .
- **Undo operations are performed first, then redo operations.**

Immediate Database Modification : Recovery Example : Guess the Undo, Redo Action

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read**(A)	T_1: **read**(C)
$A := A - 50$	$C := C - 100$
write(A)	**write**(C)
read(B)	
$B := B + 50$	
write(B)	

(a)	(b)	(c)
<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) **undo**(T_0): B is restored to 2000 and A to 1000.

(b) **undo**(T_1) and **redo**(T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) **redo**(T_0) and **redo**(T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

Deferred Database Modification

The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit

A=100, B=100, C=100

read *(A)*;
read *(B)*;
B=B+10
Write(B)
Read(c)
display*(A+B+C)*
Commit

Log

<T_i **start**>

<T_i, B, 100, 110>

<Ti commit>

Disk

B=110

Deferred Database Modification

- 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
  - No undo operation is required here
  - Only redo operation is performed on the data items
- 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

Deffered Database Modification : Recovery Example : Guess the Undo, Redo Action

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read**(A)	T_1: **read**(C)
$A := A - 50$	$C := C - 100$
write(A)	**write**(C)
read(B)	
$B := B + 50$	
write(B)	

(a)	(b)	(c)
$\langle T_0$ **start**$\rangle$	$\langle T_0$ **start**$\rangle$	$\langle T_0$ **start**$\rangle$
$\langle T_0, A, 1000, 950\rangle$	$\langle T_0, A, 1000, 950\rangle$	$\langle T_0, A, 1000, 950\rangle$
$\langle T_0, B, 2000, 2050\rangle$	$\langle T_0, B, 2000, 2050\rangle$	$\langle T_0, B, 2000, 2050\rangle$
	$\langle T_0$ **commit**$\rangle$	$\langle T_0$ **commit**$\rangle$
	$\langle T_1$ **start**$\rangle$	$\langle T_1$ **start**$\rangle$
	$\langle T_1, C, 700, 600\rangle$	$\langle T_1, C, 700, 600\rangle$
		$\langle T_1$ **commit**$\rangle$

If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) **redo**(T_0) must be performed since $\langle T_0$ **commit**$\rangle$ is present

(c) **redo**(T_0) must be performed followed by **redo**(T_1) since $\langle T_0$ **commit**$\rangle$ and $\langle T_1$ **commit**$\rangle$ are present

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.

Solution

Streamline recovery procedure by periodically performing checkpointing

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 Ti that started before the checkpoint, and transactions that started after Ti .

- *Scan backwards from end of log to find the most recent <**checkpoint> record***

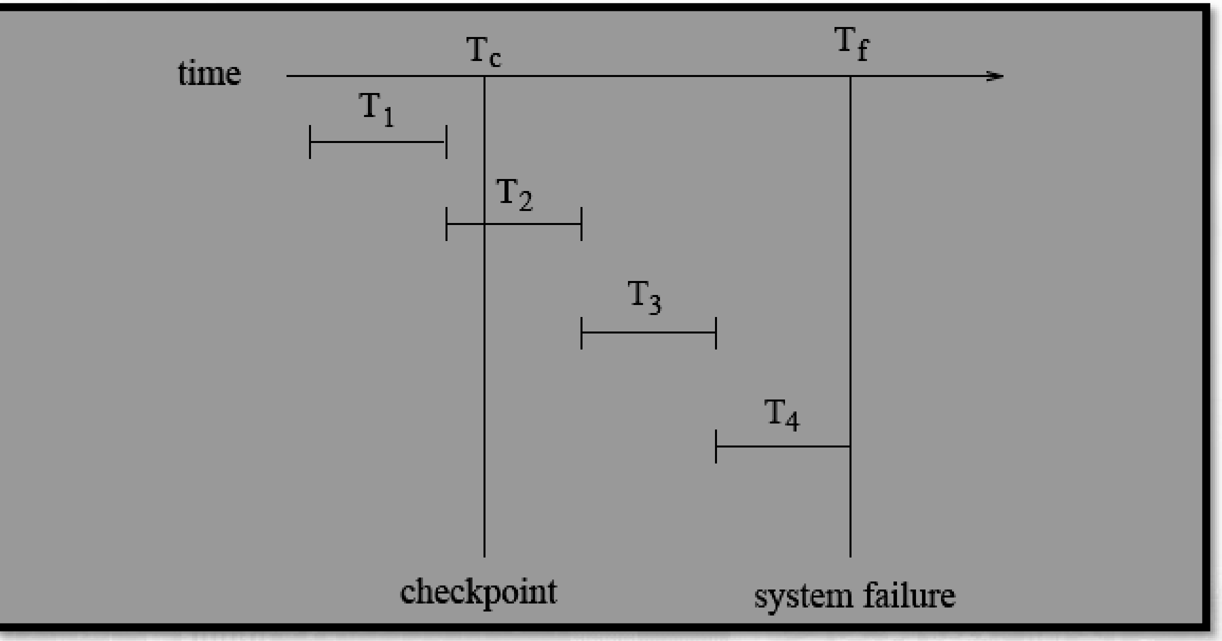
- *Continue scanning backwards till a record <Ti **start> is found.***

- *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.

- *For all transactions (starting from Ti or later) with no <Ti **commit>, or <Ti abort> found in log execute undo(Ti). (Done only in case of*** immediate modification.)

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

Checkpoints Example : Consider the scenario and suggest recovery



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

Transaction Management

Refer :

Abraham Silberschatz, Henry F. Korth and S. Sudarshan, Database System Concepts ,McGraw Hill

MIT-WPU

।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Thank You!