Transaction Management
Concurrency Control

Minal Bhise

Distributed Databases Research Group

DA-IICT, Gandhinagar

minal_bhise@daiict.ac.in

Outline

Definition

ACID properties

Transaction states

Schedule: Serial, Interleaved, equivalent

Serializability

Concurrency Control Protocols:

Lock based Protocols: Two Phase Locking, Time stamp based, deadlock detection, prevention, recovery

Time Stamp based Protocols, Validation based Protocols

Crash Recovery Protocol

Algorithm for Recovery and Isolation Exploiting Semantics ARIES

Transaction

- **Transaction** is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all **committed** (applied to the database) or all **rolled back** (undone from the database)

- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database
- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes

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
  - Concurrent execution of multiple transactions

ACID properties

A tomicity: All actions in the transaction happen, or none happen

C onsistency: If each transaction is consistent, and the database starts consistent, it ends up consistent

I solation: Execution of one transaction is isolated from that of other transactions

D urability: If a transaction commits, its effects persist

Atomicity

- Transaction to transfer $50 from account A to account B. **t=0, A=B=100**
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database

Durability

Transaction to transfer $50 from account A to account B. **t=0, A=B=100**

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

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

Consistency

- The sum of A and B is unchanged by the execution of the transaction **(A+B=200)**
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Isolation

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

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

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits

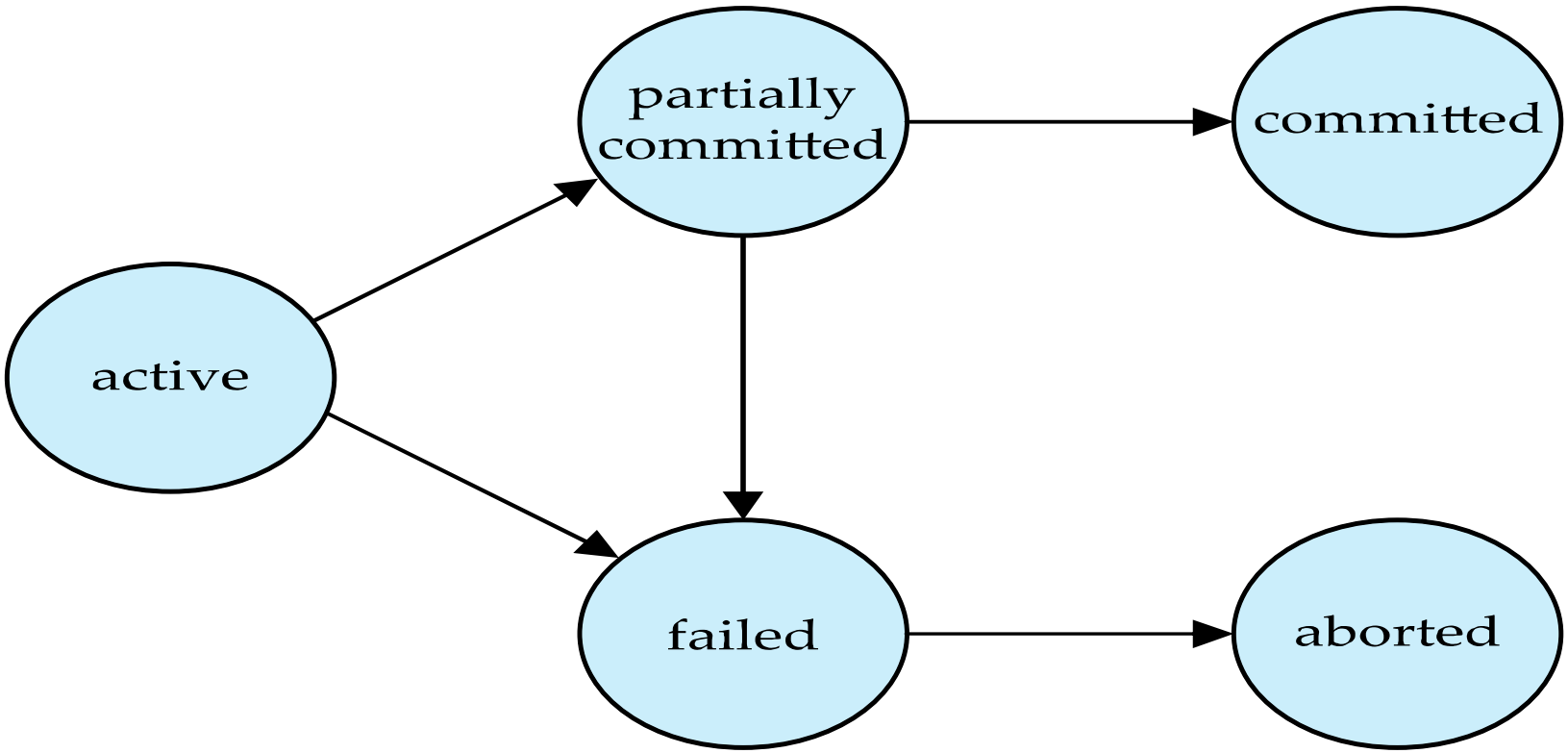
ACID Properties

- **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.

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.

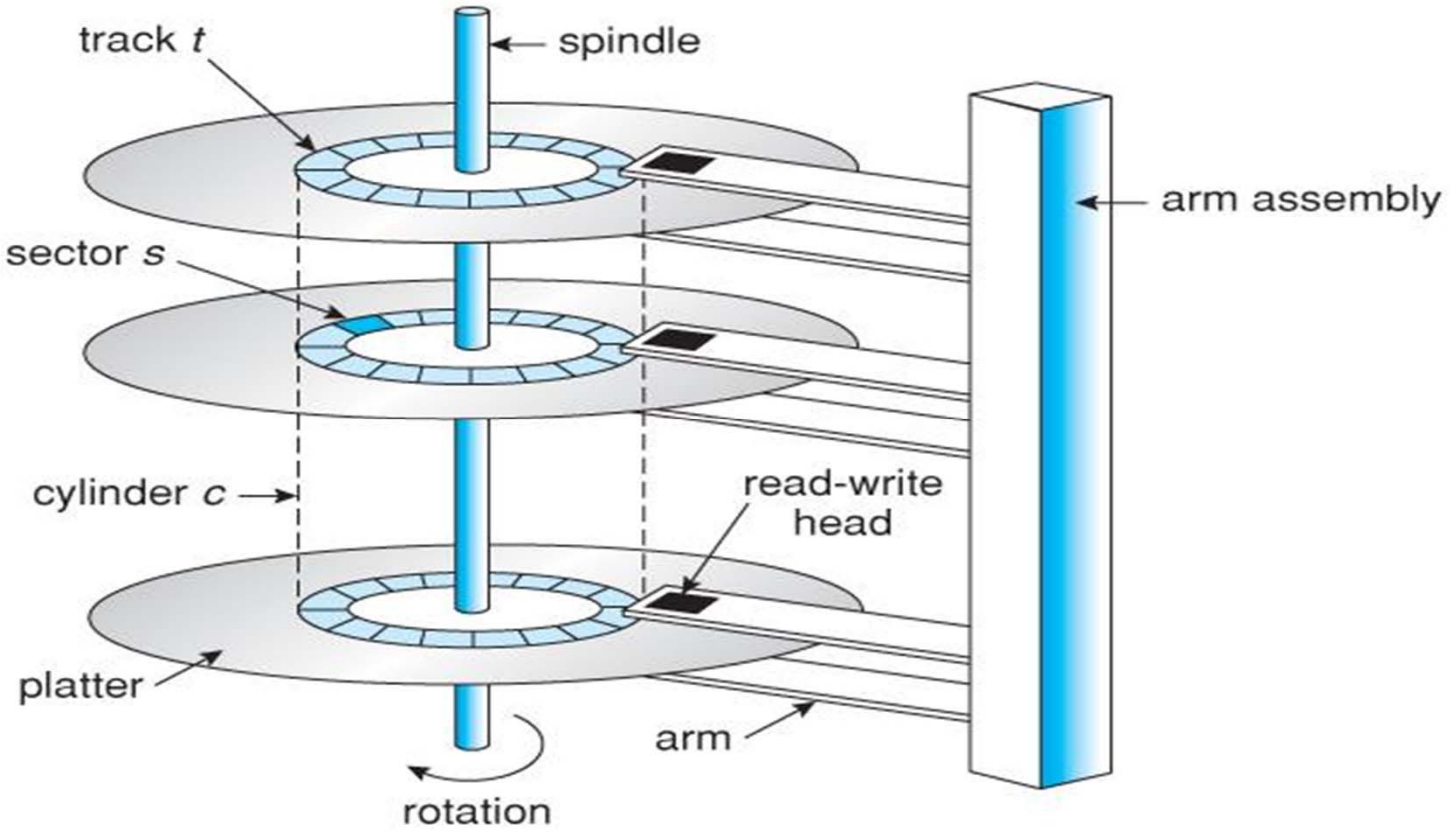
Transaction State



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
  - **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.
- **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

Disk Structure



[1] http://casem3.blogspot.com/2016/10/magnetic-disk-primary-computer-storage.html

Schedules

- 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

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- **At t=0, A=1000, B=2000, A+B= 3000, A=855, B=2145**
- A serial schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A serial schedule where T_2 is followed by T_1
- **At t=0, A=1000, B=2000, A+B= 3000, A=850, B=2150**

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	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
- **At t=0, A=1000, B=2000, A+B= 3000, A=855, B=2145**

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

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

Schedule 4

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

At t=0, A=1000, B=2000, A+B= 3000, A=950, B=2100

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

Serializability

- 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.

Simplified view of transactions

- simplified schedules consist of only **read** and **write** instructions.

- **read(*X*)** transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.
- **write(*X*)** transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.

Conflicting Instructions

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

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

- 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

Conflict Serializability

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

- a schedule that is not conflict serializable:

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

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

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability

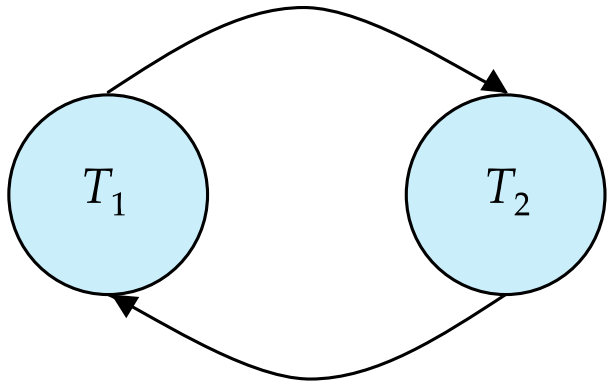
- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

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

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

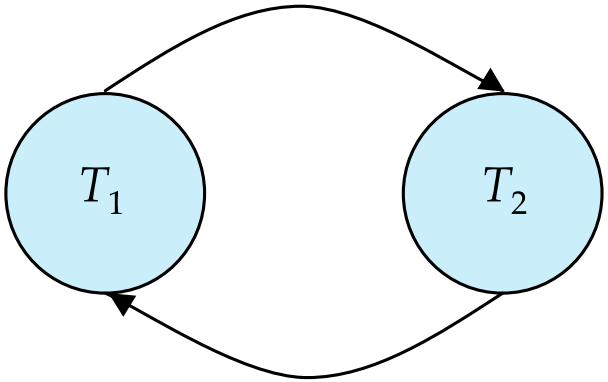
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a directed graph where the vertices are the transactions (names)
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed
- Example of a precedence graph



Example

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



Precedence Graph

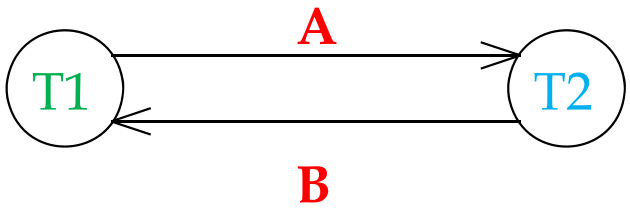
- This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $Ti \rightarrow Tj$ for which one of three conditions holds:
  - 1. *Ti* executes 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 $Ti \rightarrow Tj$ exists in the precedence graph, then, in any serial schedule *S*' equivalent to *S*, *Ti* must appear before *Tj*.

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

- **One node per transaction; edge from *Ti* to *Tj* if actions of Ti precedes and conflicts with one of Tj's actions**
- **The cycle in the graph G (V,E) reveals the problem. The output of T1 depends on T2, and vice-versa**



- **Schedule is conflict serializable if and only if its precedence graph is acyclic**

Recoverable Schedules

- 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

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

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

Cascading Rollbacks

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

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

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

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

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

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - serializable, 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
- Testing a schedule for serializability *after* it has executed is a little too late!
- to develop concurrency control protocols that will assure serializability.

Concurrency Control

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Concurrency Control Protocols

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

Lock Based CC

- Only serializable, recoverable schedules are allowed, no actions of committed transactions are lost while undoing aborted transactions
- Use of locks
- Locking protocol is a set of rules to be followed by each transaction to ensure that net effect of interleaved transactions is identical to some serial execution

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 concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive lock 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.

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)	

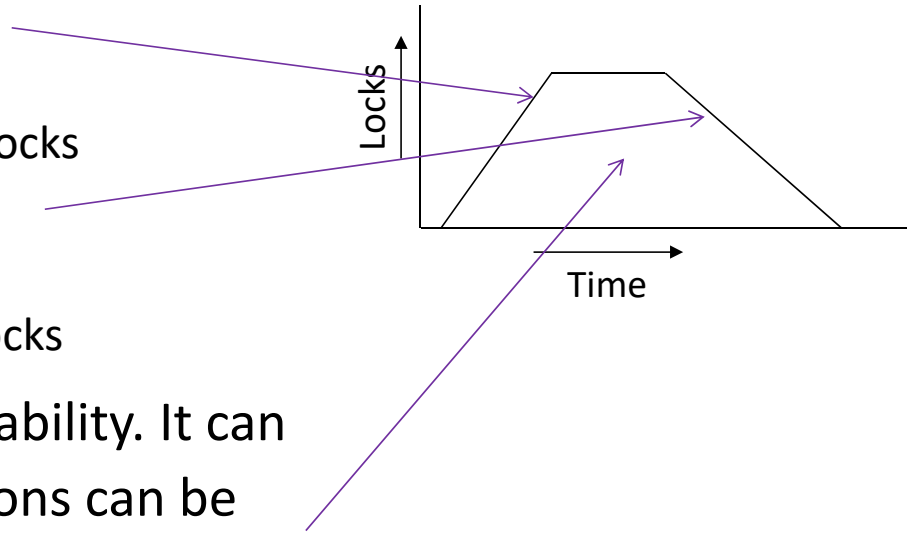
- Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*
- Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase
- To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlock

- 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 for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

- 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. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



The Two-Phase Locking Protocols

- 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.

The Two-Phase Locking Protocols

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking/ Two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

Locking in Commercial Databases

- When a transaction *Ti* issues a read(*Q*) operation, the system issues a lock-S(*Q*) instruction followed by the read(*Q*) instruction.
- When *Ti* issues a write(*Q*) operation, the system checks to see whether *Ti* already holds a shared lock on *Q*. If it does, then the system issues an upgrade(*Q*) instruction, followed by the write(*Q*) instruction. Otherwise, the system issues a lock-X(*Q*) instruction, followed by the write(*Q*) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Lock Management

- Lock and unlock requests are handled by the lock manager
- Maintains Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
- Descriptive entry in transaction table for each transaction
- Entry contains pointer to a list of locks held by the transaction
- This list is checked before requesting a lock to ensure that a transaction does not request the same lock twice

Pitfalls of Lock-Based Protocols

- Starvation is also possible if concurrency control manager is badly designed.
- A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
- Identification of deadlocks using timeout mechanism: transaction waiting for too long…assuming it is a deadlock …abort it
- Blocking : blocked transactions may hold other locks that force other transactions to wait
- Aborting and restarting: wastes the work done

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 (predeclaration).
    - 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).

Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **$TS(T_1) < TS(T_2)$**
- **wait-die scheme**
  - 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**
  - 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

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits

  - Fewer rollbacks in wound-wait

- If a transaction re-starts, make sure it has its original timestamp

Wait-Die Scheme

- Lower the timestamp, higher the priority
- If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
- Lower priority transactions can never wait for higher priority transactions
- To ensure that no transaction is repeatedly aborted, when a transaction is aborted and hence restarted, it will be given the original timestamp it had
- Only a transaction requesting a lock can be aborted
- As the transaction grows older, it tends to wait for more and more younger transactions. The conflicting younger transaction may be repeatedly aborted. But a transaction having all the locks will never be aborted.

Wound-Wait Scheme

- If Ti has higher priority, Tj aborts; otherwise Ti waits
- A transaction that has all the locks it needs may get aborted

Deadlock prevention

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, thus starvation is avoided
- **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

- Create a wait-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Schedule

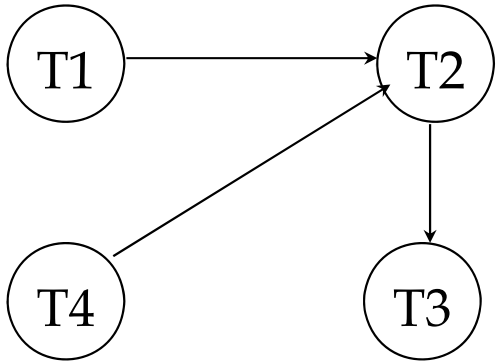
T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)

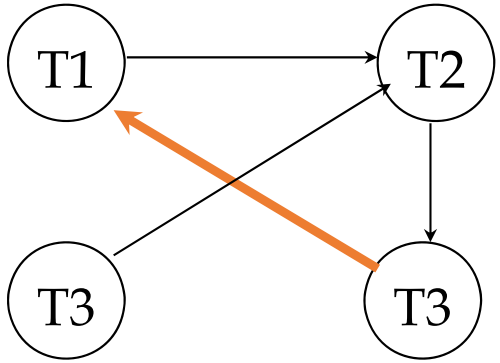
- S1(A), R1(A), X2(B), S1(B),W2(B), S3(C),R3(C), X2(C), X4(B), X3(A)

Deadlock Detection

Example:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)





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 Recovery

When deadlock is detected: Selection of victim, rollback, starvation

- **Selection of victim:**
- Select that transaction as victim that will incur minimum cost
    - How long it has computed and how long is still to go?
    - Number of data items already used by it
    - Number of data items to be used further by it
    - Number of transactions involved in rollback

Deadlock Recovery

- **Rollback** -- determine how far to roll back transaction

  Total rollback: Abort the transaction and then restart it.

  Partial rollback: More effective to roll back transaction only as far as necessary to break deadlock.

  - Needs to keep additional information like state of all running , sequence of lock request/grants and updates performed by transactions
  - Detection mechanism should decide which locks the selected transactions needs to release to break the deadlock
  - The selected transaction must be rolled back to the point where it obtained first of these locks, undoing all actions it took after that point

- **Starvation**
  - happens if same transaction is always chosen as victim.
  - Include the number of rollbacks in the cost factor to avoid starvation

Multiple Granularity

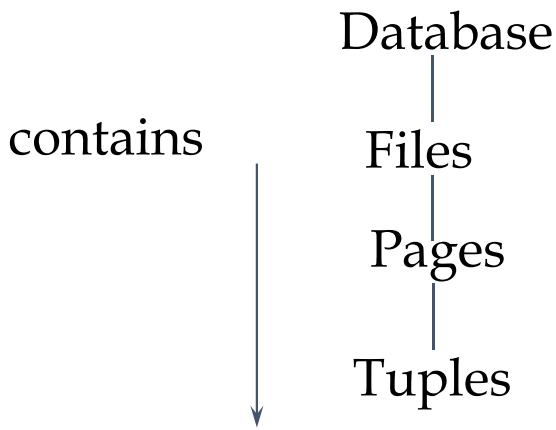
- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents 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

Granularity

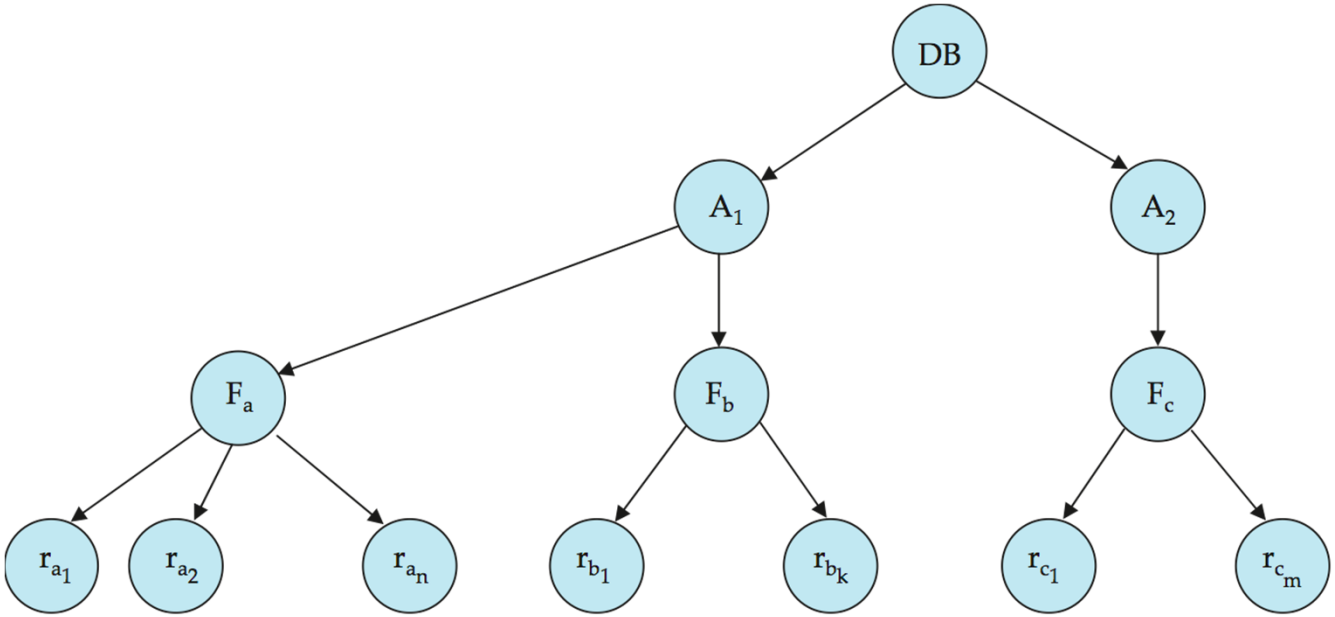
- Data items are of various sizes
- Defining hierarchy of data granularities where small granularities are nested within larger ones
- Tree
- A nonleaf node of multi granularity tree represents data associated with its descendants

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data "containers" are nested



Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are:

- *database*
- *file*
- *record*

Writer Starvation Problem

- If several READ requests are compatible, immediately grant the lock request. Such policy may cause writer starvation problem if there are a large number of read requests. If a reader was granted a read lock, then fellow readers can immediately join in. However, writers will be blocked out, until all readers have finished. In fact, some unlucky writers may get blocked indefinitely.
- A solution is to use FIFO (first in, first out) policy to queue up the requests. Only requests at the front of the queue can try to get the lock. However, concurrency and efficiency may be negatively impacted. Some opportunities for parallel access will be lost as queue processing is serial in nature.

Optimistic CC

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead
  - Deadlock detection/resolution
  - Lock contention for heavily used objects
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflits before transactions commit

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Timestamp-Based Protocols

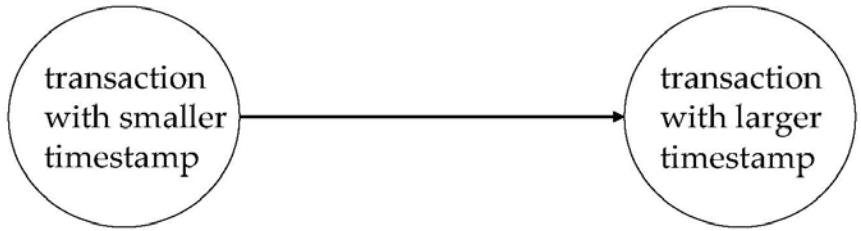
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q):
  1. If $TS(T_i) \leq$ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If $TS(T_i) \geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **mx**(R-timestamp(Q), $TS(T_i)$).

Timestamp-Based Protocols

- Suppose that transaction T_i issues **write** (Q):
  1. If TS(T_i) < R-timestamp (Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp (Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



  Thus, there will be no cycles in the precedence graph.
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.

Multiversion Timestamp Protocol

- Multiversion schemes keep old versions of data item to increase concurrency.
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- when a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering

- Suppose that transaction T_i issues a **read**(Q) or **wrie**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. TS(Ti) > R-timestamp (Qk), a new version of Q is created.
- Observe that
  - Reads always succeed.
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability.

MVCC: Implementation Issues

- Reading of data items also requires the updating of R-timestamp field (2 disk accesses)
- Conflicts are resolved though rollbacks rather than through waits (expensive)
- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Qk and Qj, and both versions have W-timestamp less than the timestamp of the oldest transaction in the system. Then the older of the 2 versions (Qk, Qj) will not be used again and can be deleted
  - the oldest active transaction has timestamp > 9, then Q5 will never be required again