Transaction Processing and Concurrency Control

Subash Manandar

NCE

Transaction Concept

- A transaction is a sequence of operations that form a single unit of work.
- Every Statement in the database is considered as transaction.
- A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**:
  - Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.
- Commit a transaction if successful else rollback if errors.
- Transaction processing means dividing information processing up into individual, indivisible operations, called transactions, that complete or fail as a whole; a transaction can't remain in an intermediate, incomplete, state (so other processes can't access the transaction's data until either the transaction has completed or it has been "rolled back" after failure).
- Transaction processing is designed to maintain database integrity (the consistency of related data items) in a known, consistent state.

ACID Properties

- Atomicity
  - Either all operations of a transaction are reflected in the database or none of them .“all or nothing”
- Consistency
  - If the database was in a consistent state before the transaction started, it will be in a consistent state after the transaction has been executed.
- Isolation
  - If transactions are executed in parallel, the effects of an ongoing transaction must not be visible to other transactions.
- Durability
  - After a transaction finished successfully, its changes are persistent and will not be lost.

Transaction to transfer money from account A to B

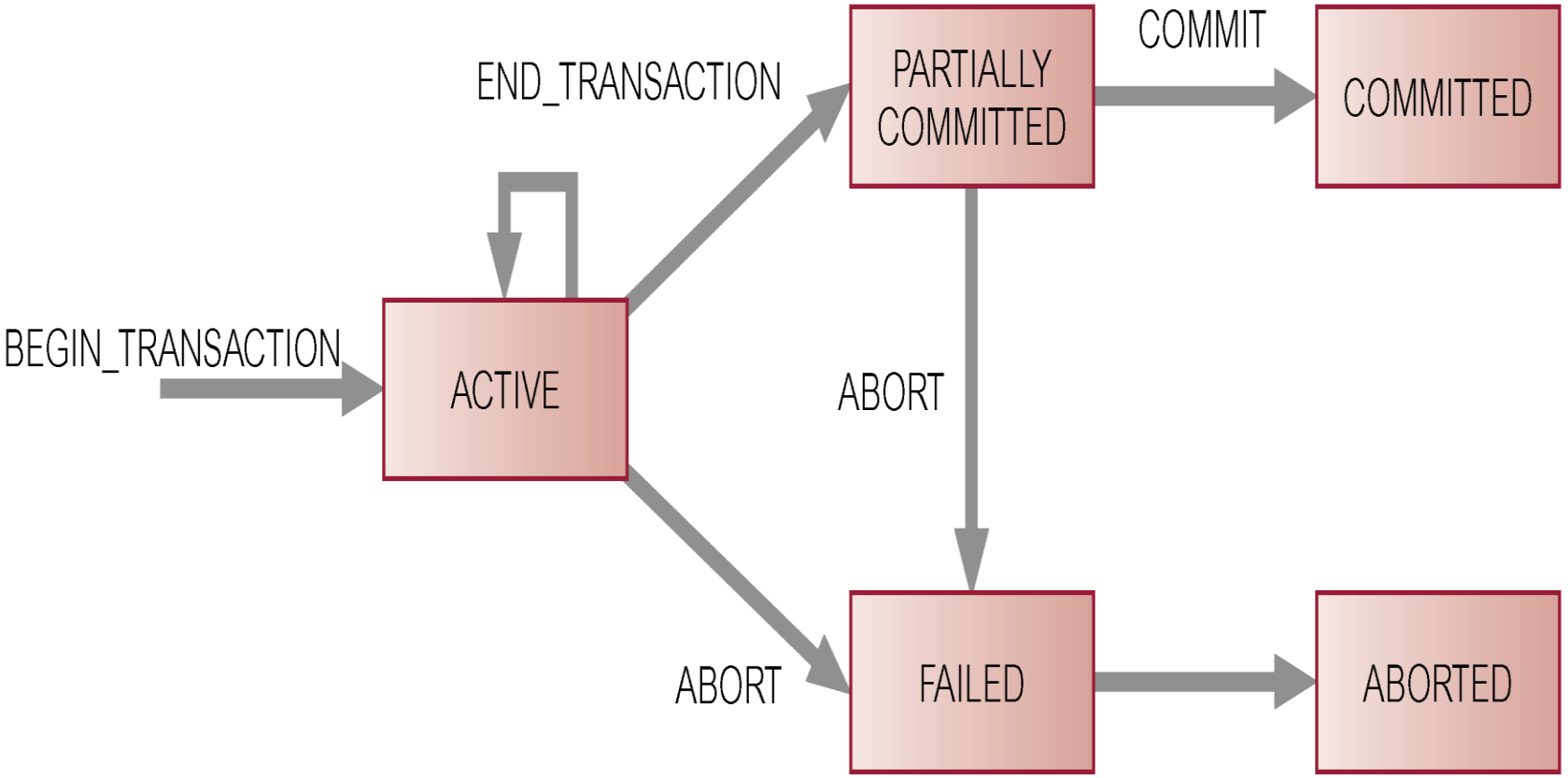
1. **Start transaction**
2. **Read(A)**
3. **A:=A-1000**
4. **Write(A)**
5. **Read(B)**
6. **B:=B+1000**
7. **Write(B)**
8. **Commit**

- Atomicity
  - If the transaction fails after step 4 but before step 8, the updates on A should not be reflected in the database (rollback)
- Consistency
  - The sum of A and B should not be changed by the transaction
- Isolation
  - If another transaction is going to access the partially updated database between step 4 and 7, it will see an inconsistent database (with a sum of A and B which is less than it should be)
- Durability
  - Once the money has been transferred from A and B (commit), the effect of the transaction must persist.

Transaction State

- A transaction must be in one of the following states:
  - Active state:
    - It is initial state of transaction. Transaction is in this state while executing. A transaction goes into an active state immediately after it starts execution, where it can issue read and write operation.
  - Partially Committed:
    - when the transaction finish or ends it moves to the partially committed state (after the last statement has been executed).
  - Failed:
    - A transaction can got to the failed state if one of the check fails or if the transaction is aborted during its active state. The transaction may rolled back to undo the effect of its write operation on the database.
  - Aborted:
    - A transaction is in this state when after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
  - Committed:
    - A transaction reaches it commit point when all its operations that access the database have been executed successfully then the transaction enters in committed state.

State Transition diagram of a Transaction



Concurrent Executions

- Transaction processing system usually allow multiple transaction to run concurrently.
- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operation from the various transaction is known as a schedule.
- Allowing multiple transactions to run concurrently and allowing multiple transaction to update data concurrently cause several complications with consistency of the data.
- Advantages are:
  - **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.

Schedule

- Scheduler can be applied basically in two ways:
  - Serial execution of transaction
    - Each operation within a transaction can be executed atomically
    - Any serial execution of a set of transactions T_1 , …,T_n by different users is regarded as a correct result.
  - Parallel execution of transaction
    - Improves the throughput and resource utilization as well as the average response time
    - But too much parallelism can lead to wrong results (dirty reads, lost updates)
    - The scheduler has to choose the appropriate concurrency control scheme to avoid problems during parallel execution

Schedule

- When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction.
- The database system must control the interaction among the concurrent transaction to prevent them for destroying the consistency of the database.
- It is done through the concurrency control mechanism.
- A schedule S specifies the chronological order in which the operations of concurrent transactions are executed
  - A schedule for the transaction T_1 , …., T_2 must contain all operations of these transactions
  - The schedule must preserve the order of the operation in each individual transaction

Schedule (e.g. Serial Schedule)

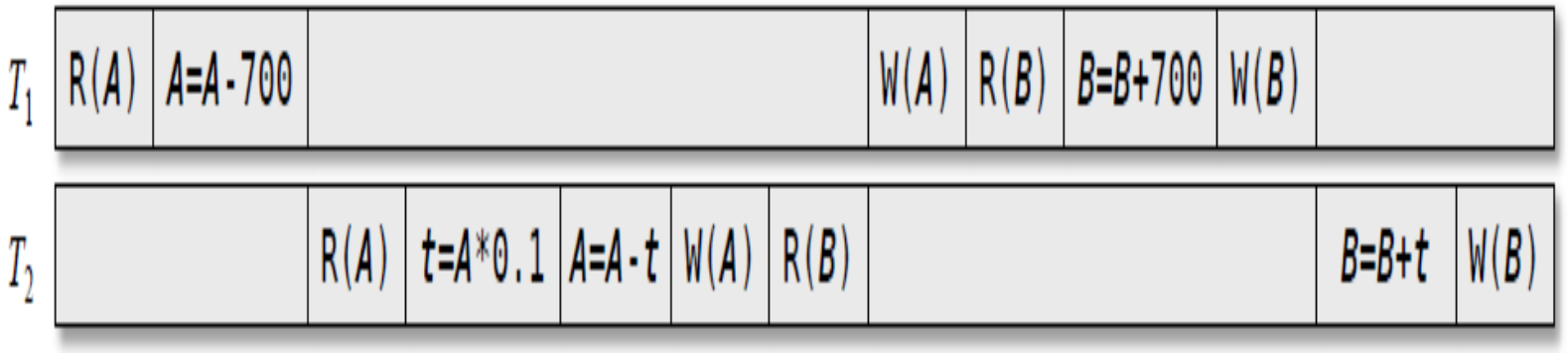
- Let transaction T_1 transfer Rs. 10000 from account A to B and T_2 transfer 10% of the balance from A to B.
- Critical are the read (R) and write (W) operation
- T1: read (A);
  A:=A-10000; // A:=20000-10000
  write (A);
  read (B);
  B:=B+10000; //B:=2000+10000=12000
  write (B);
- T2: read (A);
  temp=A*0.1; //temp:=10000*.1=1000
  A:=A - temp;//A:=10000-1000=9000
  write (A);
  read (B);
  B:=B + temp; // B:= 12000+1000=13000
  write (B);

E.g. Parallel Schedule

- T1: read (A)
  A:=A-10000 // A:=20000-10000
  write (A)
- T2: read (A)
  temp=A*0.1; //temp:=10000*.1=1000
  A:=A-temp;
  write (A);
- T1: read (B)
  B:=B+10000 //B:=2000+10000=12000
  write(B)
- T2: read (B)
  B:=B + temp ; // B:= 12000+1000=13000
  write (B)

E.g. Parallel Schedule

- this schedule *does not preserve the sum of A and B and therefore leads to problems*



Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- The main objective of Serializability is to search non-serial schedules that allow transaction to execute concurrently without interfering one another transaction and produce the result database state that could be produced by a serial execution.
- We can conclude that if a non serial schedule is correct if it produces the same results as some serial execution. Such a schedule that is equivalent to a serial schedule is said to be serializable.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- Ordering of read and write is important
- Rules
  - If two transactions only read data item they do not conflict and order is not important.
  - If two transactions either read or write completely separate data items, they do not conflict and order is not important.
  - If one transaction writes a data item and another reads or writes same data items, order of execution is important.

Example

T1	T2	T3
R(a)		
	R(b)	
		R(c)
W(a)		
	W(b)	
		W(c)
commit	commit	commit

Schedule 1

T1	T2	T3
R(a)		
W(b)		
Commit		
	R(b)	
	W(b)	
	Commit	
		R(c)
		W(c)
		Commit

Schedule 2

Here, the actions of the transactions in schedule 1 are not executed as same as in 2, but at the end 1 gives the same result as 2.
Thus, it is considered as Serializable.

Conflict Serializability

- 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:
- 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

Instruction I_i	Instruction I_j	Result
Read(Q)	Read(Q)	No Conflict
Read(Q)	Write(Q)	Conflict
Write(Q)	Read(Q)	Conflict
Write(Q)	Write(Q)	Conflict

Conflict Serializability

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

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

Schedule A

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

Schedule B

Conflict Serializability

- Example of a schedule that is not conflict serializable:

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

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

View Serializability

- Two schedules are said to be view equivalent if the following three conditions hold:
  1. For each data item *Q*, if transaction *Ti* reads the initial value of *Q* in schedule *S*, then transaction *Ti* must , in schedule *S'* also read the initial value of *Q*.
  2. For each *Q*, if *Ti* executes *read(Q)* in *S* and if value was produced by *write(Q)* operation executed by *Tj*, then the *read(Q)* of *Ti* must, in *S'* , also read value of *Q* produced by same *write(Q)* of *Tj*
  3. For each Q, the transaction that performs the final write(Q) operation in S must perform the final write(Q) operation in S'.
- View serializability:
  - Definition of serializability based on view equivalence.
  - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

View Serializability

- Schedule X is a view serializable schedule.
- It is view equivalent to the serial schedule<T3,T4,T5> since the one Read(Q) reads the initial value of Q in both schedules and T5 performs the final Write of Q in both schedules.
- Here, Writes of T4 and T5 are called blind writes.
- *Every conflict serializable schedule is also view serializable but not vice versa.*

T3	T4	T5
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)

Schedule X

T3	T4	T5
Read(Q)		
Write(Q)		
	Write(Q)	
		Write(Q)

Schedule Y

Testing of Serializability

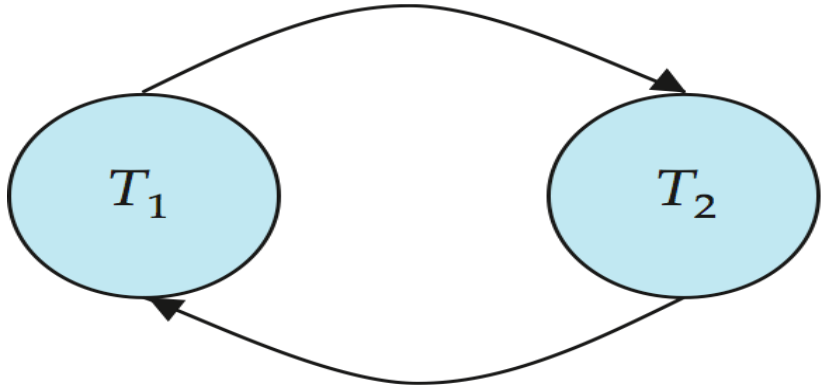
- For a testing of Serializability the simple and efficient method is to construct a directed graph called a precedence graph from S.
- Given a schedule S, a precedence graph is a directed graph G= (N, E) where
  - N=set of nodes
  - E=set of directed Edges
- Create as follows
  - Create a node for each transaction
  - A directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i.
  - A directed edge $T_i \rightarrow T_j$, if T_j writes the value into an item after it has been read by T_i.
- The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three condition holds
  - T_i executes write before T_j executes read
  - T_i executes read before T_j executes write
  - T_i executes write before T_j executes write

  If an edge $T_i \rightarrow T_j$ exits in the precedence graph for S, then in any serial schedule S' equivalent to S, T_i must appear before T_j .

Example

- *A precedence graph is said to be acyclic if there are no cycles in the graph.*
- *The precedence graph for serializable schedule S must be acyclic, hence it can be converted to a serial 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



Concurrency Control

- Different concurrency control schemes can be used to ensure the isolation property is ensured when multiple transactions are executed in parallel.
- The DBMS must guarantee that only serializable recoverable schedule are generated and it also guarantees that no effect of committed transaction is lost, and no effect of aborted (roll back) transaction remains in the related database.
- Due to concurrent execution it is very hard to preserve the isolation property. To ensure it, the system must control the interaction among the concurrent transactions.
- The control is achieved through one of the mechanism called concurrency control schemes.

Lock Based Protocol

- 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-compatibility matrix

	S	X
S	true	false
X	false	false

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

Lock Based Protocol

- Example of a transaction performing locking:

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

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- 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.
- The locking protocol must ensure serializability.

Sample Transactions with Locks

- T1: lock-X(B) T2: lock-S(A)

T1	T2
read(B)	read(A)
B = B -50;	unlock(A)
write(B);	lock-S(B)
unlock(B);	read(B);
lock-X(A);	unlock(B);
read(A);	display(A+B);
A = A + 50;	
write(A);	
unlock(A);	

Schedule for Transactions: non-serializable schedule

T1	T2	concurrency-control manager
lock-X(B)		
		grant-X(B, T1)
Read(B		
B = B - 50;		
write(B);		
unlock(B);		
	lock-S(A)	
		grant-S(A, T2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T2)
	read(B);	
	unlock(B);	
	display(A+B);	
lock-X(A);		
		grant-X(A, T2)
read(A);		
A = A + 50;		
write(A);		
unlock(A);		

Pitfalls of Lock Based Protocol

- Consider the partial schedule
- 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 must be released.

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)	

- The potential for deadlock exists in most locking protocols.
- **Starvation** is also possible if 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.

Two Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability.
  - It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking.

Two Phase Locking Protocol

- **Strict two-phase locking**.
  - Here a transaction must hold all its exclusive locks till it commits/aborts.
  - No cascading rollback
- **Rigorous two-phase locking** is even stricter:
  - Here *all* locks (shared and exclusive) are held till commit/abort.
  - No cascading rollback (of course)
  - In this protocol transactions can be serialized in the order in which they commit.

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

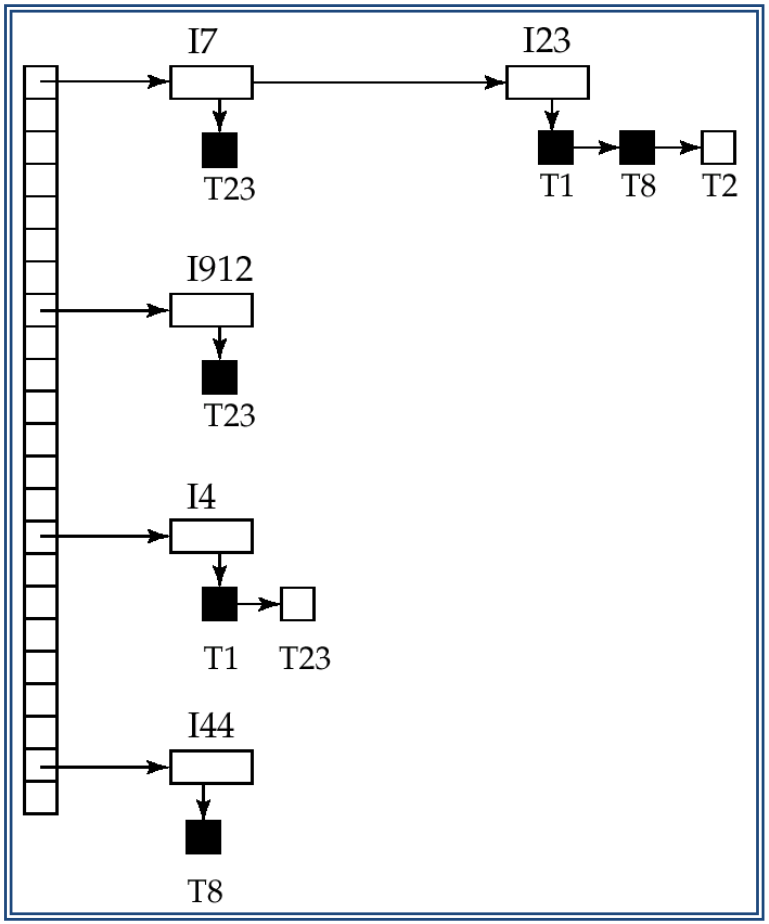
2PL with Lock Conversion

- The original lock mode with (lock-X, lock-S)
  - assign lock-X on a data D when D is both read and written
- 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 serializability.
- The refined 2PL gets more concurrency than the original 2PL
- *Strict two phase locking and rigorous two phase locking with lock conversion are used extensively in commercial database systems.*

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



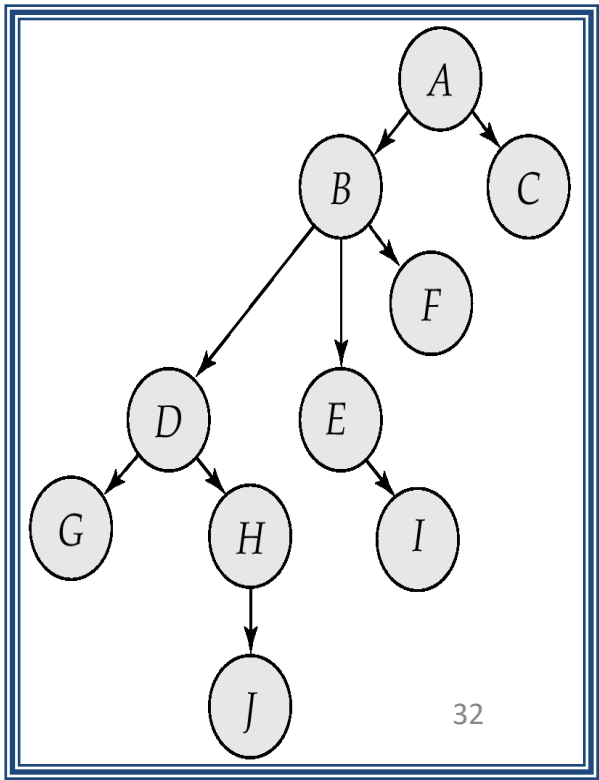
- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result 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 Protocol

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., 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 (DAG), called a *database graph*.

- The *tree-protocol* is a simple kind of graph protocol.
  - Only exclusive locks are allowed.
  - The first lock by T_i may be on any data item if there is no lock on the 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.

- in tree locking protocol a transaction may have to lock data items that it does not access.
- increased locking overhead and additional waiting time
- potential decrease in concurrency
- it is deadlock free so no rollback
- unlocking may occur earlier



Timestamp Based Protocol

- 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.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Timestamp Ordering Protocol

- Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q),
   - 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),
   - The **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

- Suppose that transaction T_i issues a **write**(Q).

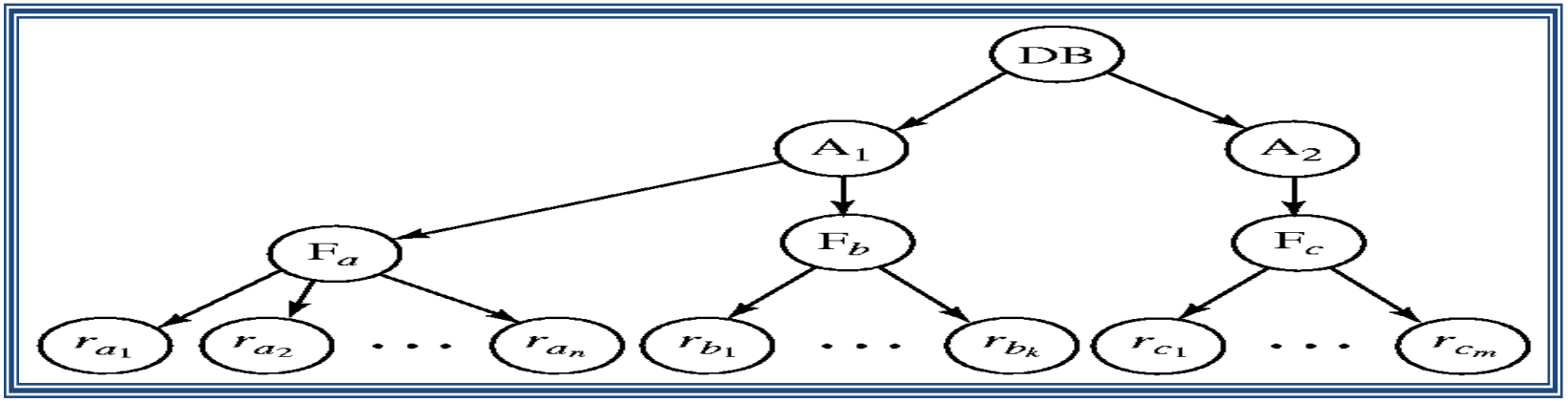
1. If TS(T_i) < R-timestamp(Q),
   - The value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the **write** operation is rejected, and T_i is rolled back.

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, (TS(T_i) $\geq$ R-timestamp(Q) and TS(T_i) $\geq$ W-timestamp(Q))
   - The **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

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



- The highest level is the entire database and then *area*, *file* and *record*.

Multiple Granularity

- 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 allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Multiple Granularity Locking Protocol

- Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. 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.
4. 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.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. 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.

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.
- Consider the following two transactions:

  T_1: write (X) T_2: write(Y)

  write(Y) write(X)

- Schedule with deadlock

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

Deadlock Handling

- To deal with deadlock, we can use:
  - Deadlock prevention protocol
    - Ensure that system will never enter a deadlock state
  - Deadlock detection and Recovery scheme
    - Try to recover system once it entered to deadlock state
- Both methods may result in transaction rollback.
- Prevention is used if probability of system entering a deadlock state is relatively high.
- Otherwise detection and recovery are more efficient.

Deadlock Prevention

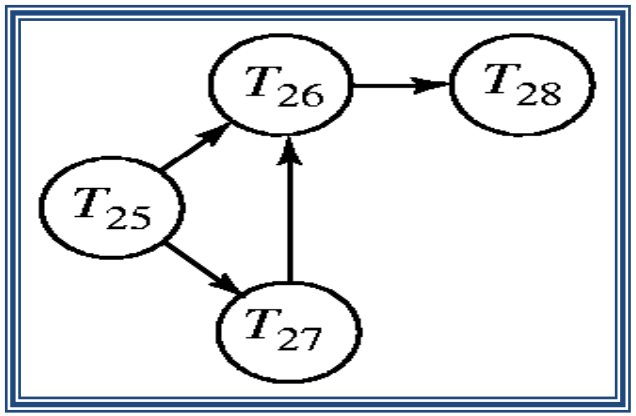
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items
    - require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
  - Timeout-Based Schemes :
    - a transaction waits for a lock only for a specified amount of time.
      - After the wait time is out and the transaction is rolled back. (No deadlock!)
    - simple to implement; but starvation is possible
    - Also difficult to determine good value of the timeout interval.

Deadlock Prevention

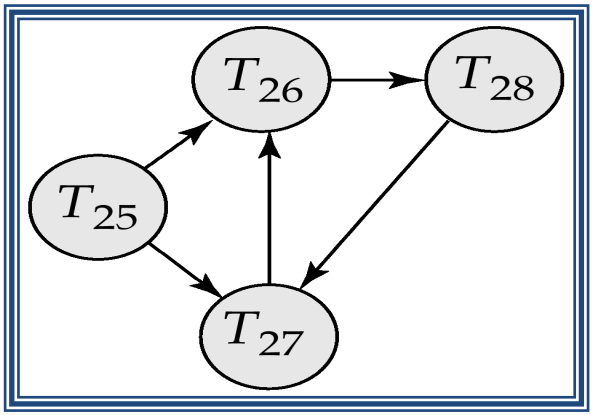
- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
  - **Wait-die** scheme — non-preemptive
    - Older transaction may wait for younger one to release data item.
    - Younger transactions never wait for older ones; they are rolled back instead.
    - A transaction 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.
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp.
  - Older transactions thus have precedence over newer ones in these schemes, and starvation is hence avoided.

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 held by T_j, then $T_i \rightarrow T_j$ 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.
- The system invokes a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected:
  - 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.
    - Partial rollback: More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim.
    - The system may include the number of rollbacks in the cost factor to avoid starvation

Starvation

- Problem of locking generates starvation.
- It occurs when a transaction cannot proceed for a definite period of time while other transaction in the system continue normally. This may occur if the waiting scheme for locked items is unfair giving priority to some transaction over other.
- It can be removed either by using FIFO queue or the longer the transaction waits the higher priority that it will get over another.