Chapter 15 : Concurrency Control

Database System Concepts, 6th Ed.

Chapter 15: Concurrency Control

- Isolation
  - The fundamental properties of a transaction
- When several transactions execute concurrently in the database
  - The isolation property may no longer be preserved
    - So, the system must control the interaction among the concurrent transactions
  - The control is achieved through mechanism called *concurrency control* schemes

Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability after it has executed is a little too late!
- *Goal – To develop concurrency control protocols that will* assure serializability
  - They will generally not examine the precedence graph as it is being created;
  - Instead a protocol will impose a discipline that avoids nonserializable schedules
- Require study of concurrency protocols in Chapter 15.
- *Tests for serializability help understand why a concurrency* control protocol is correct

Chapter 15: Concurrency Control

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

- Most frequently used schemes
  - Two-Phase Locking
  - Snapshot Isolation

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
  - To allow MULTIPLE transactions to READ a data item
  - But, limits WRITE access to just ONE transaction at a time
- Every transaction require a lock on data item Q in an appropriate mode depending upon the type of operations it will perform on the data item Q
- Transaction makes Lock requests to concurrency-control manager
- Transaction can proceed only after request is granted

Lock-Based Protocols

- Data items can be locked by transaction in two modes
  1. *Exclusive (X) mode*

     Data item Q can be both <u>read</u> as well as <u>written</u>

     X-lock is requested using **lock-X(Q)** instruction
  2. *Shared (S) mode*

     Data item can only be <u>read</u>

     S-lock is requested using **lock-S(Q)** instruction

- To unlock a data item Q using Unlock**(Q)** instruction

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may **be granted a lock on an item if the requested lock is COMPATIBLE with locks already held on the item by other transactions**

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**
- **ANY NUMBER of transactions can hold SHARED locks on an item**,
- But if any **transaction HOLDS AD 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,
  - Then only the lock is granted

Lock-Based Protocols

- **Banking Example**
- T1 transfers $50 from Acct B to Acct A
- T2 displays the total money in Acct A and B
- A=$100
- B=$200
- If schedule,
  - T1 □ T2, then A+B=300
  - T2 □ T1, then A+B=300

Transaction 1	**Transaction 2**
read (B)	read (A)
B=B-50	read (B)
write (B)	Display (A+B)
read (A)	
A=A+50	
write (A)	

- **Now, add lock information**

Lock-Based Protocols

- **Banking Example**
- T1 transfers $50 from Acct B to Acct A
- T2 displays the total money in Acct A and B
- A=$100
- B=$200
- If schedule,
  - T1 □ T2, then A+B=300
  - T2 □ T1, then A+B=300

Transaction 1	Transaction 2
lock-X(B)	**lock-S(A)**
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)	

Lock-Based Protocols

- **The transaction making a lock request cannot execute its next action until the concurrency control manager grants the lock**
- Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction

Shows points at which the concurrency-contro manager grants the locks

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

Schedule 1

Lock-Based Protocols

- Transactions are executed concurrently
- T2 displays $250
  - As T1 unlocked data item B too early
  - Inconsistent state

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Schedule 1

Lock-Based Protocols

- **Banking Example (UNLOCKING DELAYED)**
- T3 transfers $50 from Acct B to Acct A
- T4 displays the total money in Acct A and B
- A=$100
- B=$200
- No possibility of incorrect display of total $250
- Consistent state

Transaction 3	Transaction 4
lock-X(B)	
read (B)	
B=B-50	
write (B)	
lock-X(A)	
read (A)	
A=A+50	
write (A)	
unlock (B)	
unlock (A)	
	lock-S(A)
	read (A)
	lock-S(B)
	read (B)
	Display (A+B)
	unlock (A)
	unlock (B)

Pitfalls of Lock-Based Protocols

1. 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
  - **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A
- Such a situation is called a **deadlock**

Pitfalls of Lock-Based Protocols

- **Solution to deadlock**

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)	

  - To handle a deadlock one of the two transactions (T_3 or T_4) must ROLL BACK and release locks of that transaction
- The potential for deadlock exists in most locking protocols

Pitfalls of Lock-Based Protocols (Cont.)

2. **Starvation** is also possible, if concurrency control manager is badly designed
   - For example:
     - A sequence of transactions that each requests a S-lock on the data item and each transactions releases the lock a short while after it is granted, but T_x never gets the X-lock on the data item
       - The Transaction T_x may never make progress, and said to be starved
     - **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 T_x is repeatedly rolled back due to deadlocks
   - Concurrency control manager can be designed to prevent starvation

Prevent Starvation

- Avoid starvation of transactions by granting locks as shown
  - When a transaction *Ti requests a lock on a data item Q in a particular* mode *M, the concurrency-control manager grants the lock provided that:*

    1. There is no other transaction holding a lock on *Q in a mode that conflicts* with M

    2. There is no other transaction that is waiting for a lock on *Q and that made its lock request before Ti*
  - Thus, a lock request will never get blocked by a lock request that is made later

Lock-Based Protocols (Cont.)

- 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 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

Lock-Based Protocols (Cont.)

- Locking protocols
  - Restrict the set of possible schedules
  - The set of all such schedules is a proper subset of all possible serializable schedules
  - Allow only conflict serializable schedule

Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules
- Requires two phases
  - 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

Check Transaction for Two-Phase Locking Property

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

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

T_3:
lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4:
lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

- *Transactions T1 and T2 are not two phase, here mix of lock and unlock*
- *While,* Transactions ***T3 and T4 are two phase****, as all locks first and then all unlock*
  - *Clear boundary where the transaction has obtained its final lock (the end of its growing phase) is known as* ***lock point (a point of final lock)***

Check Transaction for Two-Phase Locking Property

T_3:	T_4:
lock-X(B);	lock-S(A);
read(B);	read(A);
$B := B - 50$;	lock-S(B);
write(B);	read(B);
lock-X(A);	display($A + B$);
read(A);	unlock(A);
$A := A + 50$;	unlock(B).
write(A);	
unlock(B);	
unlock(A).	

- *The unlock instructions do not need* to appear at the end of the transaction
- For example, in the case of transaction *T3,* we could **move the unlock(*B*)** *instruction to* ***just after the lock-X(A) instruction****,* and still retain the two-phase locking property

Two-Phase Locking Protocol

- The protocol assures serializability
  - 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 end of its growing phase)
    - Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions

Two-Phase Locking Protocol

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 protocol assures serializability, BUT NOT ENSURE FREEDOM FROM DEADLOCK

Two-Phase Locking Protocol

- To being serializable schedule, schedules should be cascadeless
- But, cascading rollback may occur

T_5	T_6	T_7
lock-x (A)		
read (A)		
lock-s (B)		
read (B)		
write (A)		
unlock (A)		
	lock-x (A)	
	read (A)	
	write (A)	
	unlock (A)	
		lock-s (A)
		read (A)

- Here in this schedule, the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7

The Two-Phase Locking Protocol (Cont.)

- Pitfalls
  - D*oes not* ensure freedom from deadlocks
  - Cascading roll-back is possible
- Solution to Two-Phase Locking Protocol (Used commercially)
  1. **Strict two-phase locking** is a modified protocol
     - Transaction must hold all its **Exclusive locks** till it commits/aborts
     - Preventing any other transaction from reading the data
  2. **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

Lock Conversions

- If, Two-phase locking
  - Then T_8 *must lock* a_1 *in exclusive* mode, therefore, any concurrent execution of both transactions amounts to a serial execution
- However, T_8 *needs an exclusive lock on* a_1 *only at* the end of its execution, when it writes a_1
  - *Thus, if* T_8 *could initially lock* a_1 *in* shared mode, and then could later change the lock to exclusive mode, **we could get more concurrency**, since T_8 *and* T_9 ***could access*** a_1 ***and*** a_2 ***simultaneously***
- Solution
  - A refinement of the basic two-phase locking protocol
    4. **Lock conversions**

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

Lock Conversions

- **A mechanism** for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock
- Conversion
  - From shared to exclusive modes by **upgrade**, and
  - From exclusive to shared by **downgrade**
- **Lock conversion cannot be** allowed arbitrarily, Rather
  - Upgrading can take place in only the growing phase
  - Downgrading can take place in only the shrinking phase

Lock Conversions

- 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
- But still relies on the programmer to insert the various locking instructions

Lock Conversions

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Incomplete Schedule with lock conversion

- *Transactions T_8 and T_9 can run concurrently under* the refined two-phase locking protocol (here shown only some of the locking instructions)
  - A transaction attempting to upgrade a lock on an item *Q may* be forced to wait
    4. **Enforced wait** occurs if *Q is currently locked by another* transaction in shared mode (T_8 last line in schedule)

Lock Conversions

- Like the two-phase locking protocol, two-phase locking with lock conversion generates
  - Only conflict-serializable schedules
    - Transactions can be serialized by their lock points
- If exclusive locks are held until the end of the transaction
  - Then, the schedules are cascadeless

Automatic Acquisition of Locks

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

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

Automatic Acquisition of Locks (Cont.)

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

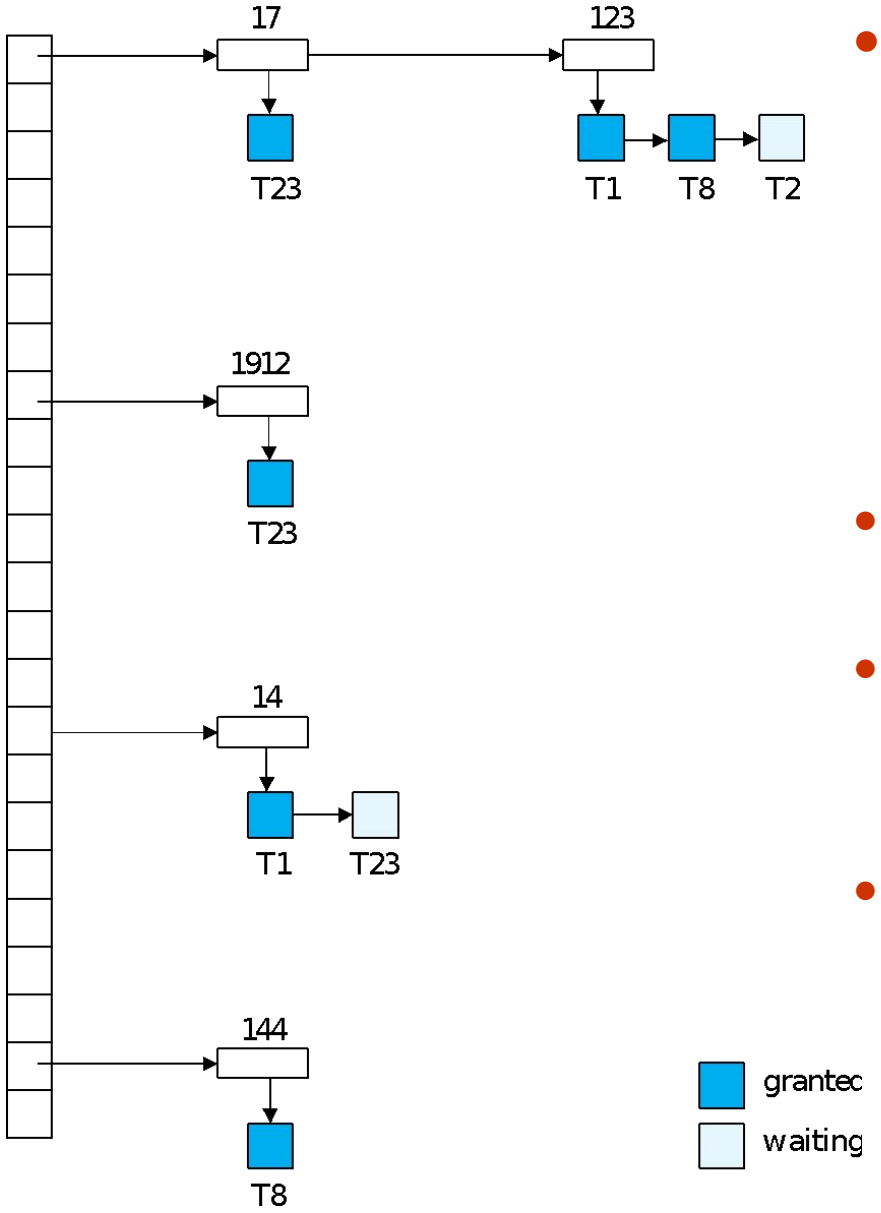
```
if T_i has a lock-X on D
  then
    write(D)
  else begin
      if necessary wait until no other trans. has any lock on D,
      if T_i has a lock-S on D
          then
              upgrade lock on D to lock-X
          else
              grant T_i a lock-X on D
    write(D)
    end;
```

- **All locks are released after commit or abort**

Implementation of Locking

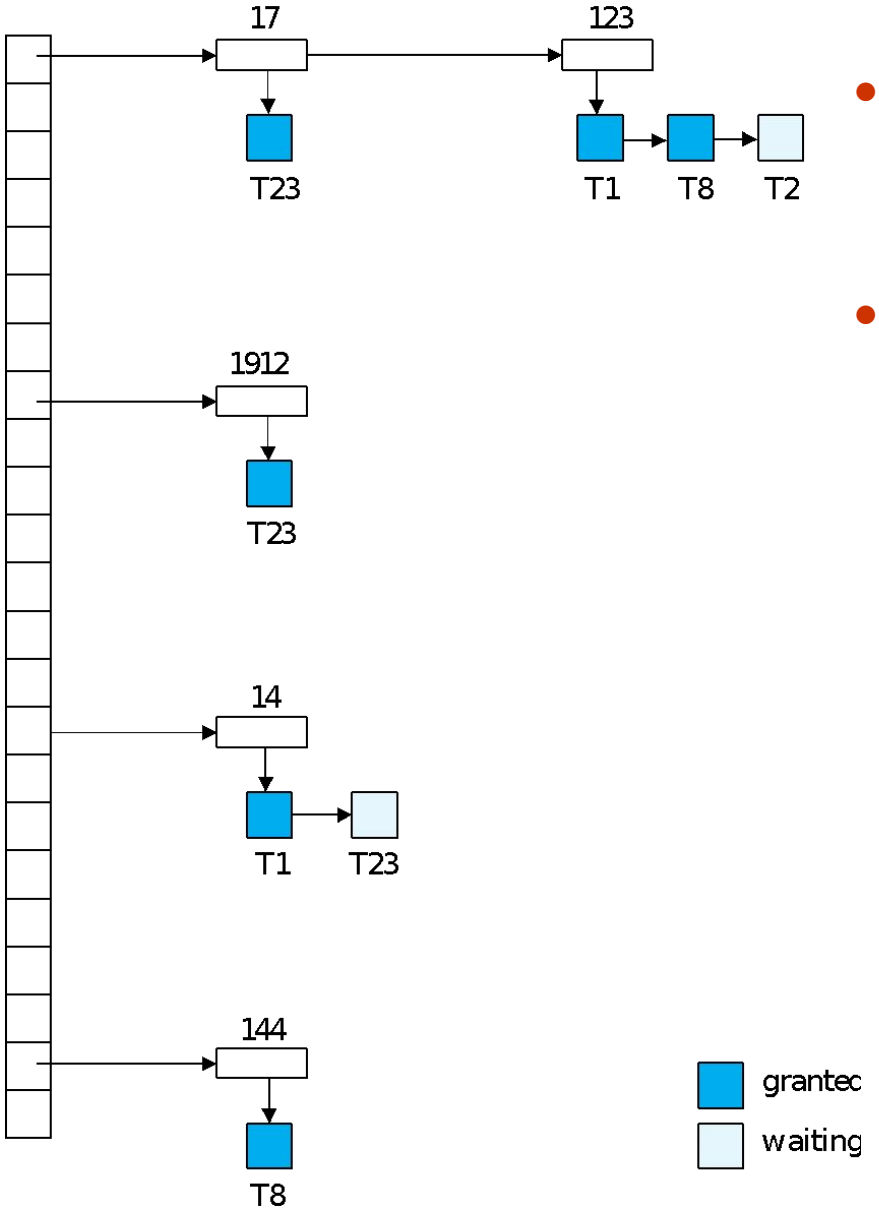
- **Lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
  - It replies, to a lock request by sending
    - A lock grant messages
    - A message asking the transaction to roll back (in case of a deadlock)
  - The requesting transaction waits until its request is answered
  - It maintains a data-structure called a **lock table** to record granted locks and pending requests

Lock Table



- Implemented as an **in-memory hash table indexed on the name of the data item being locked**
  - Data item maintains **link list of transaction record**, one for each request, **in the order in which the requests arrived**
- Data Items
  - 17, 1912, 14, 144 and 123
- Colored rectangles
  - Dark indicate granted locks
  - Light indicate waiting requests
- It also **records the type of lock** granted or requested (here not shown)

Lock Table

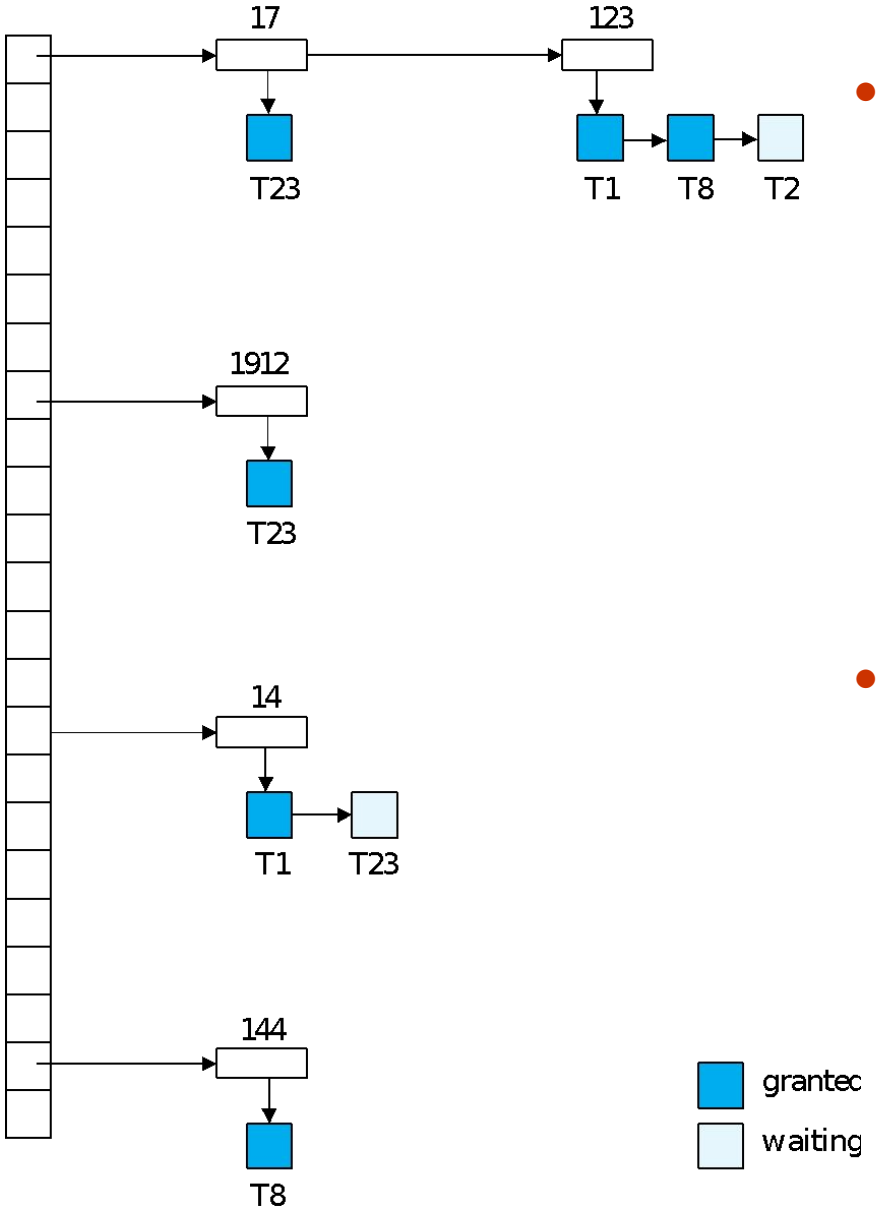


- Example: **Transaction T23** has been
  - Granted locks on items 1912 and 17
  - Waiting for a lock on item 14
- When **lock request arrives**

  Step 1. if link list is present
    - Adds a record to the end of the link list queue of requests for the data item,
    - **OR** creates a new link list

  Step 2. Either Grants,
    - If not currently locked, or If it is compatible with all earlier locks and ALL EARLIER REQUESTS HAVE BEEN GRANTED
    - **OR** Waits

Lock Table



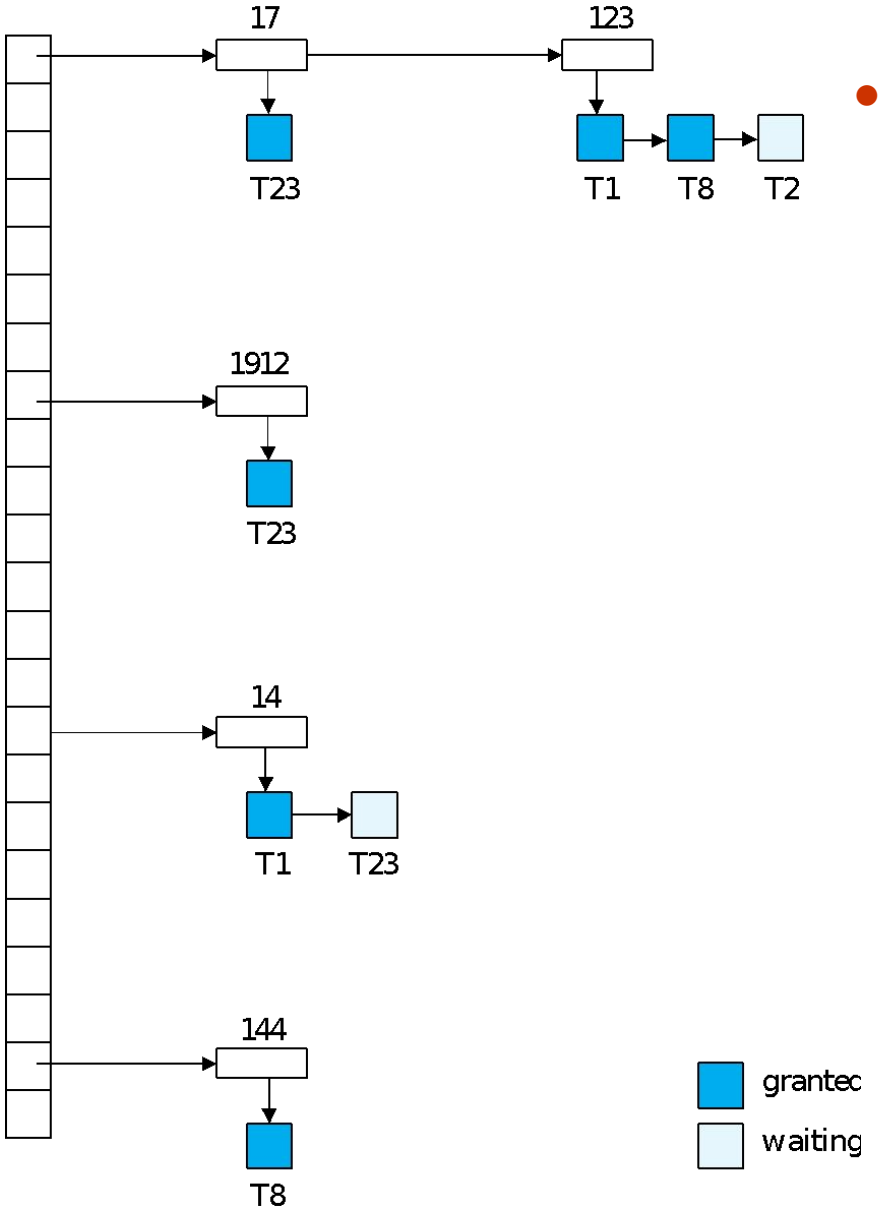
- When **unlock request arrives**

  Step 1. Deletes the record for that data item in the linked list corresponding to that transaction

  Step 2. Also it checks, following requests are checked to see if they can now be granted. If it can, grants the request and process the record following it and so on

- If transaction **aborts**
  - Deletes all waiting or granted requests of the transaction
  - Lock manager may keep a list of locks held by each transaction, to implement this efficiently (that helps in Recovery Ch. 16))

Lock Table

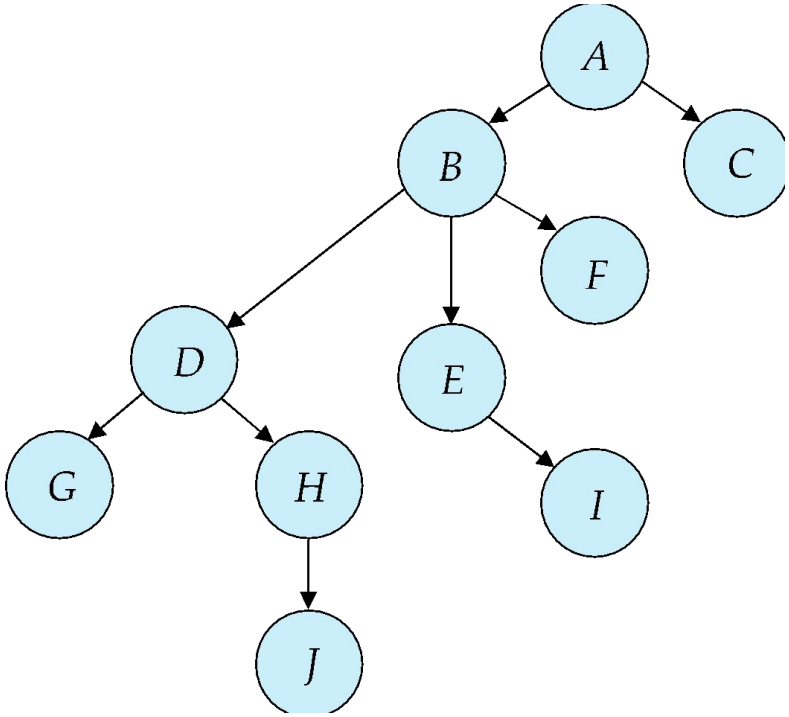


- Advantages
  - Freedom from starvation for lock requests
    - As request can never be granted while a request received earlier is waiting to be granted
  - Deadlock can occur
    - But, Deadlock can be resolved (later section)

Graph-Based Protocols

- An alternative to two-phase locking
- Based on PRIOR KNOWLEDGE
  - About the order in which the database items will be accessed
- 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, called a ***database graph***
    - **Considering Graphs as Rooted Trees**
- A simple kind of graph protocol
  - The ***tree-protocol***
    - Considering *o*nly **exclusive locks (lock-x)**

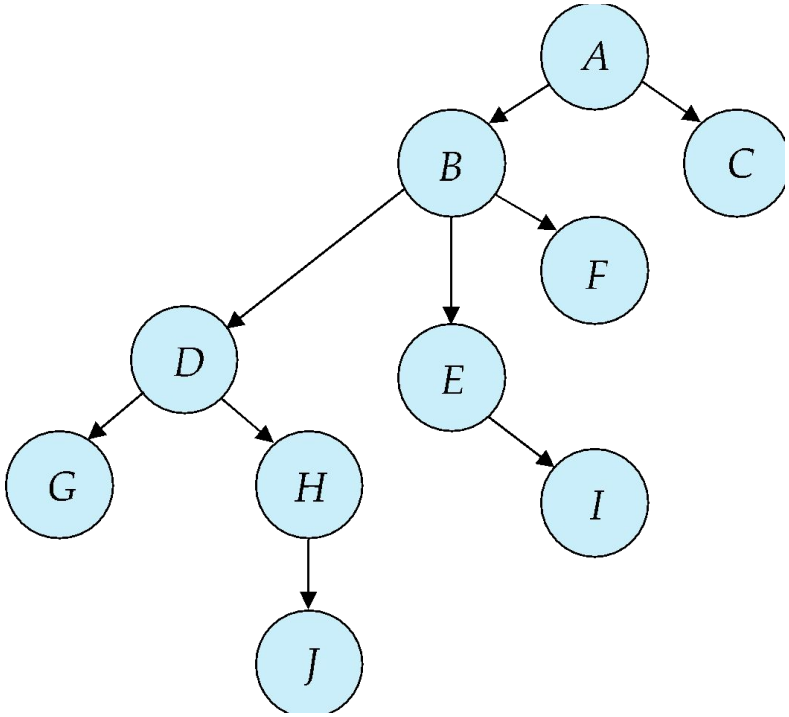
Tree Protocol



- Transaction T_i can lock a data item by following rules:

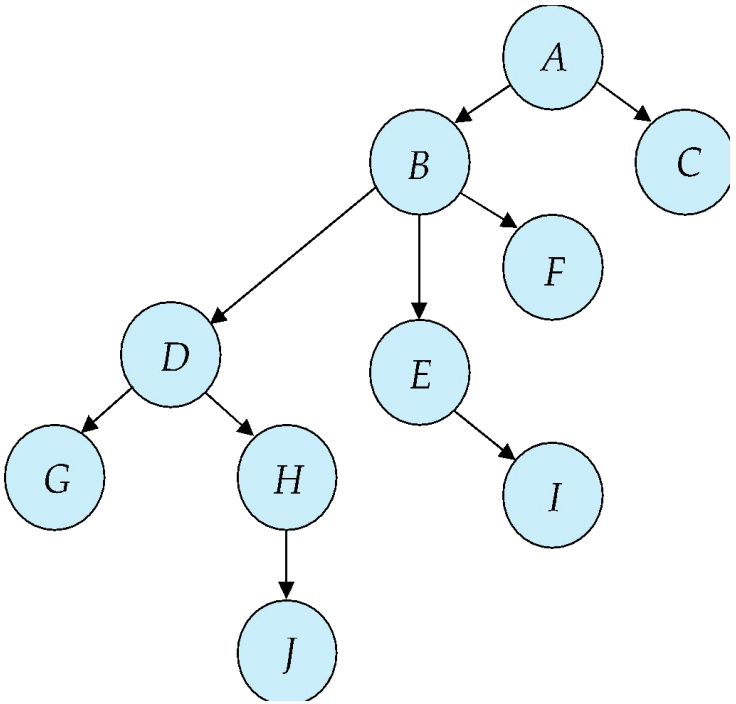
1. The FIRST **lock** by T_i may be on ANY data item
2. Subsequently, a data item Q can be locked by T_i only **if the parent of Q is currently locked by T_i**
3. Data items may be **unlocked at any time**
4. A data item that has been locked and unlocked by T_i **cannot subsequently be relocked** by T_i

Tree Protocol



- Four Transactions
  - T_{10} : lock-x(B), lock-x(E), lock-x(D), unlock(B), unlock(E), lock-x(G), unlock(D), unlock (G)
  - T_{11} : lock-x(D), lock-x(H), unlock(D), unlock (H)
  - T_{12} : lock-x(B), lock-x(E), unlock(E), unlock(B)
  - T_{13} : lock-x(D), lock-x(H), unlock(D), unlock (H)

Tree Protocol



Serialized Schedule

T_{10}	T_{11}	T_{12}	T_{13}
lock-x (*B*)			
	lock-x (*D*)		
	lock-x (*H*)		
	unlock (*D*)		
lock-x (*E*)			
lock-x (*D*)			
unlock (*B*)			
unlock (*E*)			
		lock-x (*B*)	
		lock-x (*E*)	
	unlock (*H*)		
lock-x (*G*)			
unlock (*D*)			
			lock-x (*D*)
			lock-x (*H*)
			unlock (*D*)
			unlock (*H*)
		unlock (*E*)	
		unlock (*B*)	
unlock (*G*)			

- Transaction T_{10} holds locks on two disjoint subtrees

Graph-Based Protocols (Cont.)

- Tree Protocol
  - Ensures conflict serializability
  - Advantages over two-phase locking
    - Free from deadlock, so no rollbacks are required
    - Unlocking may occur earlier
      - Leads to shorter waiting times
      - Increases concurrency

Graph-Based Protocols (Cont.)

- Tree Protocol Drawbacks
  - Transactions may have to lock data items that they do not access
    - **Increased locking overhead**, and **additional waiting time**
    - **Potential decrease in concurrency**
  - Protocol does not guarantee recoverable or cascadelessness schedules
    - To achieve this, protocol can be modified by not permitting x-lock until the end of transaction
      - But, that reduces concurrency
    - To achieve recoverability, need to introduce **commit dependencies**

Graph-Based Protocols (Cont.)

- **Commit dependencies**
  - For each data item with an uncommitted write, record which transaction performed the last write to the data item
  - When transaction T_i reads uncommitted data, record the commit dependency of T_i transaction that performed the last write to the data item
  - Transaction T_i is not permitted to commit until the commit of all transactions on which it has a commit dependency
  - If any of these transactions aborts, T_i must also be aborted

Deadlock Handling

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

- 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 Handling Methods

1. Deadlock Prevention

- Ensures that the system will *never* enter into a deadlock state

2. Deadlock Detection and Recovery

- Allow the system to enter a deadlock state and then try to recover

- Utilize, If the probability that the system would enter a deadlock state,
  - Relatively high
    - Deadlock Prevention
  - Low
    - Deadlock Detection and Recovery

Deadlock Prevention

1. ***Deadlock Prevention Schemes based on***
   A. The **ordering of lock** requests or all locks to be acquired together
   B. Using transaction **rollback** instead of waiting for a lock

Deadlock Prevention

A. Deadlock Prevention Schemes based on ordering and all locks together

- 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)
  - Disadvantages
    - Transactions may have to lock data items that they do not access and thus **increased locking overhead**, additional waiting time and decrease in concurrency
    - Protocol does not guarantee recoverable or cascadelessness schedules

Deadlock Prevention

A. *Deadlock Prevention Schemes based on ordering and all locks together*

- Require that each transaction locks all its data items before it begins execution (**predeclaration**)
  - Disadvantages
    - Before the transaction begins, it is **hard to predict, what data items need to be locked**
    - Data item utilization may be very low, as many of data items may be locked but **unused for a long time**

Deadlock Prevention

B. Deadlock Prevention Schemes using Rollbacks

- Uses the concept of **Preemption**
  - When a transaction Tx requests a lock that transaction Ti holds, the lock granted to Ti may be preempted by rolling back of Ti and granting of the lock to Tx
  - To decide whether to roll back or wait, assigns transaction timestamp based on a
    1. Counter

       Time stamp is the value of the counter when the transaction enters the system

       Incremented after a new timestamp has been assigned
    2. System clock

       Value of the clock when the transaction enters the system

Deadlock Prevention

B. Deadlock Prevention Schemes using Rollbacks

- If the transaction is rollback, it retains the old timestamp when restarted
- Schemes
  - **wait-die and wound-wait**

More Deadlock Prevention Strategies

- Transaction timestamp based schemes for the deadlock prevention
  - **wait-die** scheme — non-preemptive
    - Allows the **older transaction to wait but kills the younger one**
    - If $TS(T_i) < TS(T_j)$: T_i is older than T_j − then, T_i is allowed to **wait**
      - Older transaction may wait for younger one to release data item
    - If $TS(T_i) > TS(t_j)$: T_i is younger than T_j − then T_i **die**s and T_i is restarted later with a random delay but with the same timestamp
      - Younger transactions never wait for older ones; they are rolled back instead
    - Transaction may die several times before acquiring needed data item

More Deadlock Prevention Strategies

- Transaction timestamp based schemes for the deadlock prevention
  - **wait-die** scheme — non-preemptive
    - Example
    - Transaction T_{22}, T_{23}, T_{24} have time-stamps 5, 10 and 15 respectively
      - (5 is older timestamp, 15 younger timestamp)
      - If T_{22} requests a data item held by T_{23} then T_{22} will wait
      - If T_{24} requests a data item held by T_{23}, then T_{24} will be rolled back

More Deadlock Prevention Strategies

- Transaction timestamp based schemes for the deadlock prevention
  - **wound-wait** scheme — preemptive
    - Older transaction *wounds* (forces to abort or release the item by rollback) of younger transaction instead of waiting for it
    - If $TS(T_i) < TS(T_j)$, then
      - T_i forces T_j to be rolled back − that is T_i **wounds** T_j
      - T_j is restarted later with a random delay but with the same timestamp
    - Younger transactions may wait for older ones
    - If $TS(T_i) > TS(T_j)$, then T_i is forced to **wait** until the data item is available

More Deadlock Prevention Strategies

- Transaction timestamp based schemes for the deadlock prevention
  - **wound-wait** scheme — preemptive
    - Example
    - Transactions T_{22}, T_{23}, T_{24} have time-stamps 5, 10 and 15 respectively
      - If T_{22} requests a data item held by T_{23}, then data item will be preempted from T_{23} and T_{23} will be rolled back
      - If T_{24} requests a data item held by T_{23}, then T_{24} will wait

Deadlock prevention (Cont.)

- **Time-stamp based Schemes**
  - Wait-Die „
    - Wait: If Ti is older than Tj, then Ti waits
    - Die: If Ti is younger than Tj, then abort Ti
  - Wound-Wait „
    - Wound: If Ti is older than Tj, then abort Tj
    - Wait: If Ti is younger than Tj, then Ti waits
  - Rolled back transactions are restarted with its original timestamp
  - Older transactions thus have precedence over newer ones, and starvation is hence avoided

Deadlock prevention (Cont.)

- 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

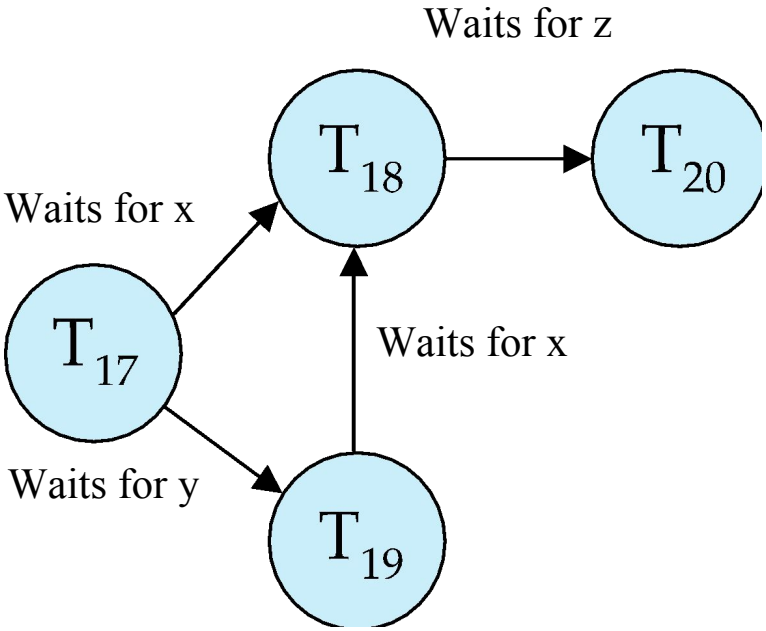
Deadlock Detection & Recovery

- Aborting a transaction is not always a practical approach
- Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance
  - Method "wait-for graph“
    - Suitable for only those systems where transactions are lightweight having fewer instances of resource
    - In a bulky system, deadlock prevention techniques may work well

Deadlock Detection

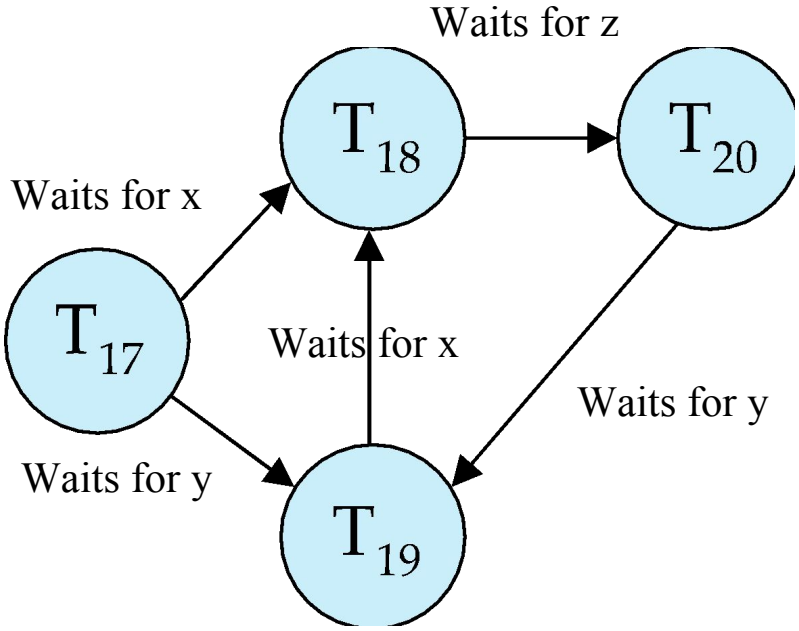
- ***wait-for graph***
  - Consists of a pair $G = (V,E)$,
    - V is a set of vertices (all the transactions in the system)
    - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$
  - If $T_i \rightarrow T_j$ is in E, Directed edge from T_i to T_j,
    - Implying that T_i is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

Deadlock Detection (Cont.)



Wait-for graph without a cycle

T_{17} waiting for T_{18} and T_{19}
T_{19} waiting for T_{18}
T_{18} waiting for T_{20}



Wait-for graph with a cycle

One more, T_{20} waiting for T_{19}
T_{18}, T_{19} and T_{20} are **deadlocked**

Deadlock Recovery

- When deadlock is detected, actions : For the rollback
  1. Selection of a victim -- determine which transaction (made a victim) to rollback to break deadlock with minimum cost factors
     a. How long the transaction has computed, and how much longer the transaction will compute before it completes
     b. How many data items used by the transaction
     c. How many more data items needs for the completion
     d. How many transactions will be involved in rollback

Deadlock Recovery

- When deadlock is detected, actions : For the rollback
  1. Selection of a victim
  2. 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 requires to manage extra information
  3. Starvation -- if same transaction is always chosen as victim, not allowing it to complete
     - Pick victim only for finite number of times and use number of rollbacks as cost factor

Multiple Granularity

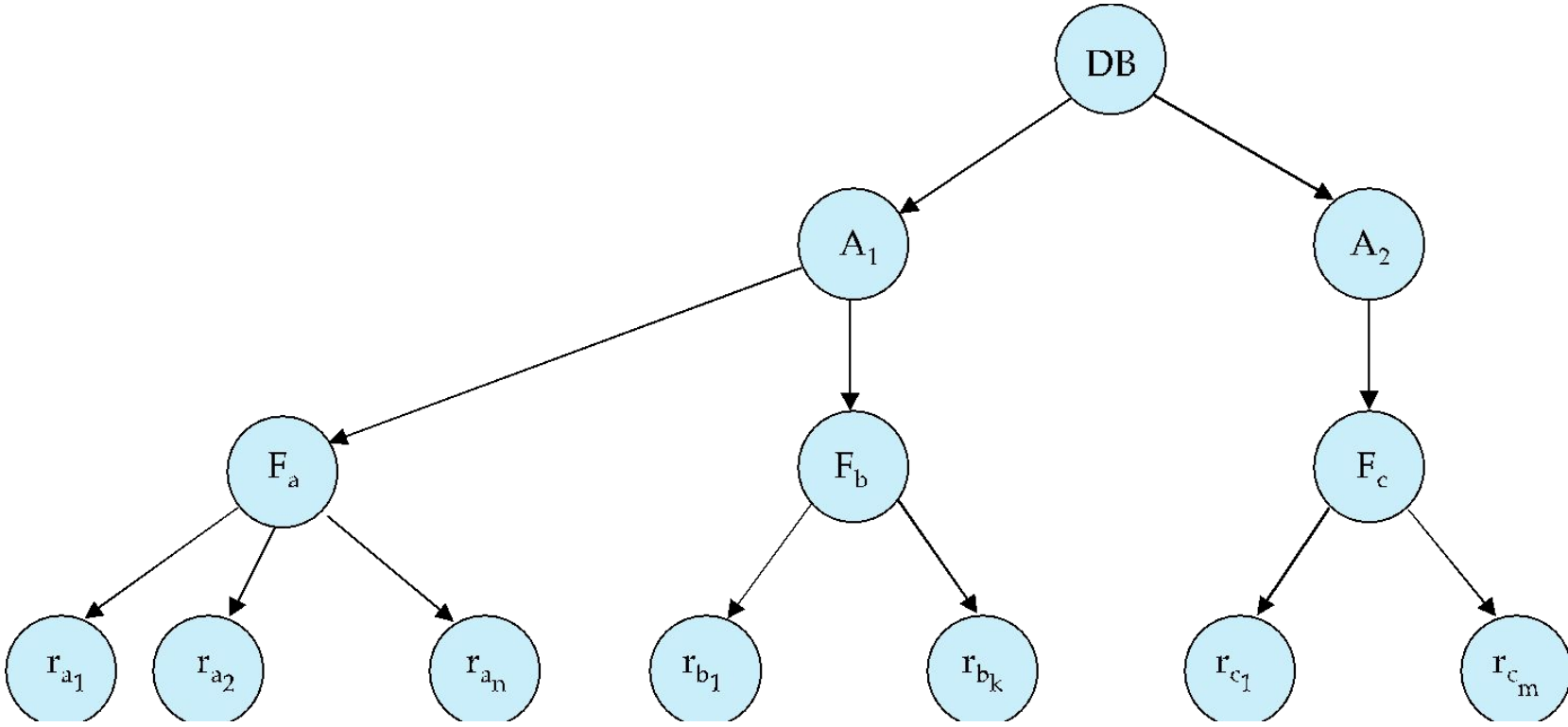
- Till this, used individual data items as the synchronization unit
- Advantageous, if consider groups of several data items and treat them as one individual synchronization unit
  - If locks of all items of database is needed
    - Time consuming, if locking each item individually in the database
    - Better, if using a single lock for the entire database
  - If locks of only few data item is needed
    - Not to lock entire database, otherwise concurrency will lost

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**

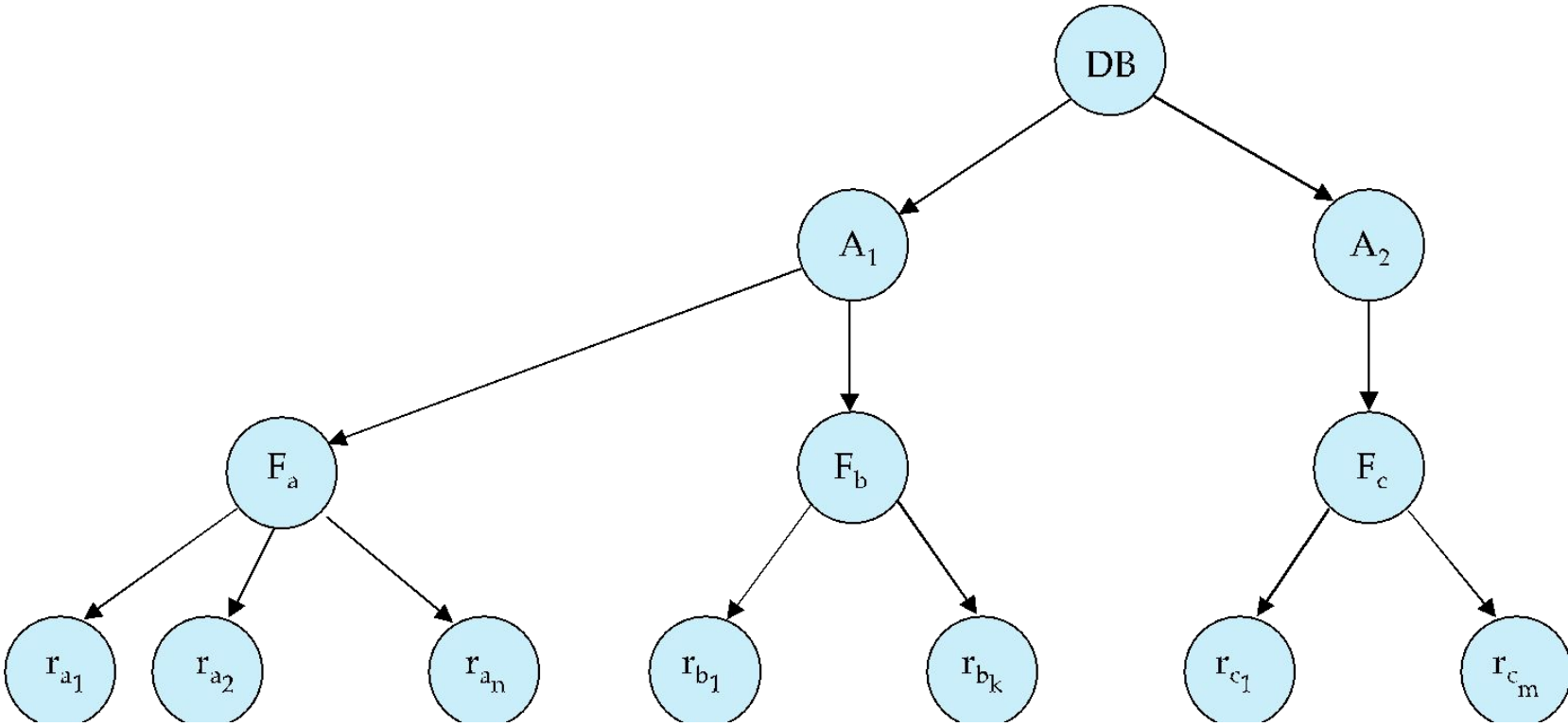
Multiple Granularity

- If transaction Ti gets an explicit lock on file Fc in X-mode
  - Then, it has an implicit lock in X-mode on all the records belonging to that file
  - Does not require to lock the individual records of Fc explicitly



Multiple Granularity

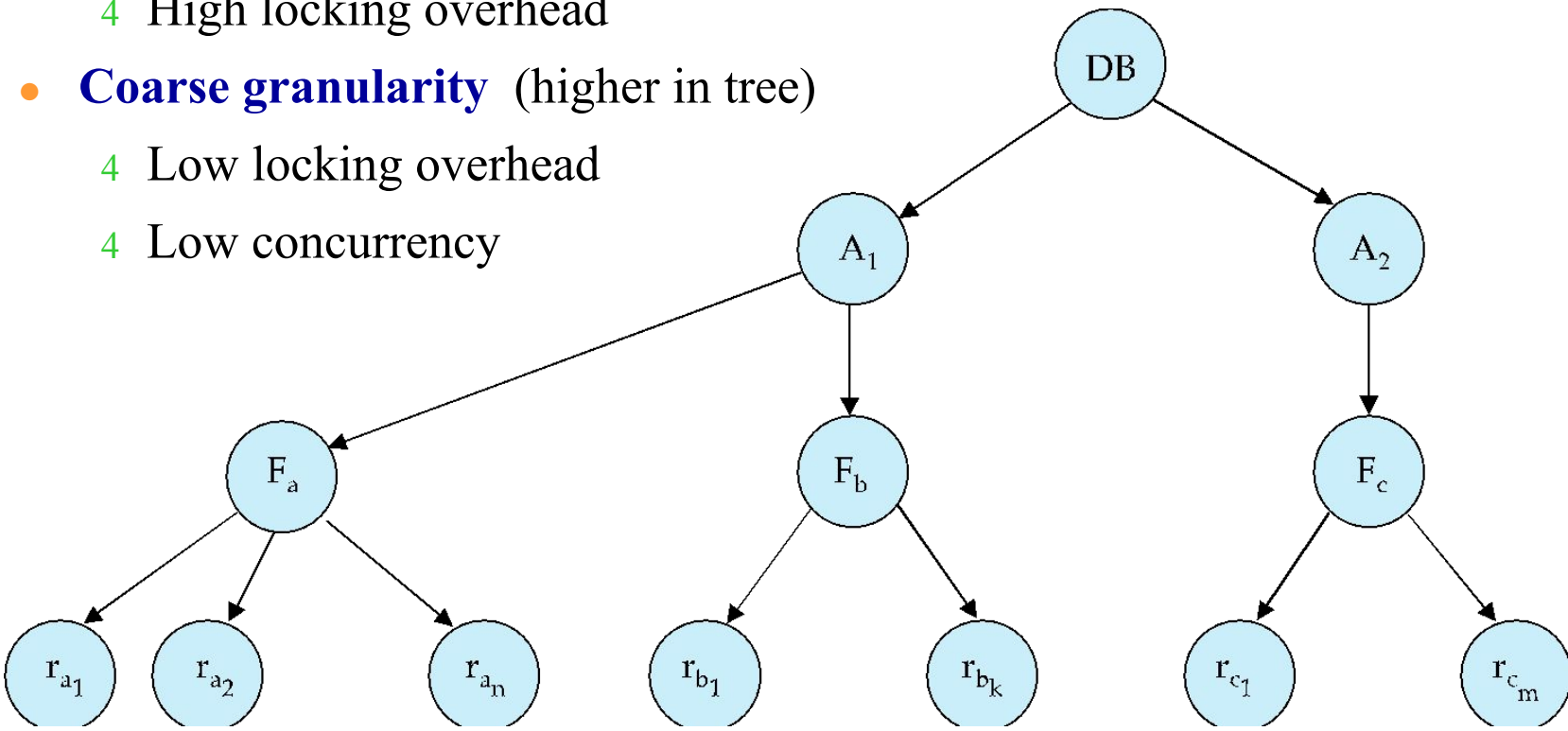
- If Transaction Tj wishes to lock record rb6 of file Fb, since Ti has locked Fb explicitly, it follows that rb6 is also locked (implicitly)
  - Whether to lock or not, Tj must traverse the tree from the root to record rb6. if any node in that path is locked in an incompatible mode, then Tj must be delayed



Multiple Granularity

- 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

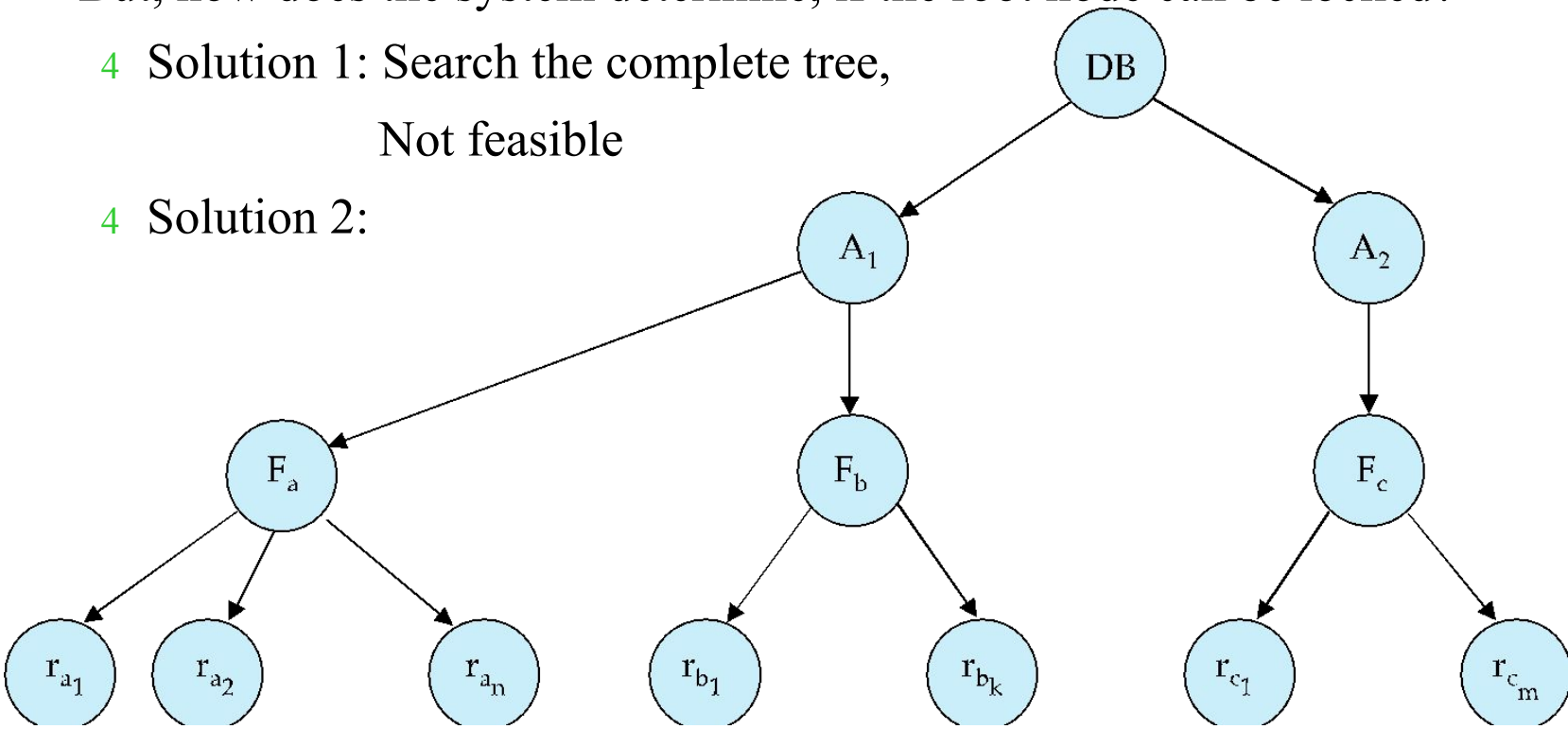
The levels, starting from the coarsest (top) level are
Database, Area , File, Record



Multiple Granularity

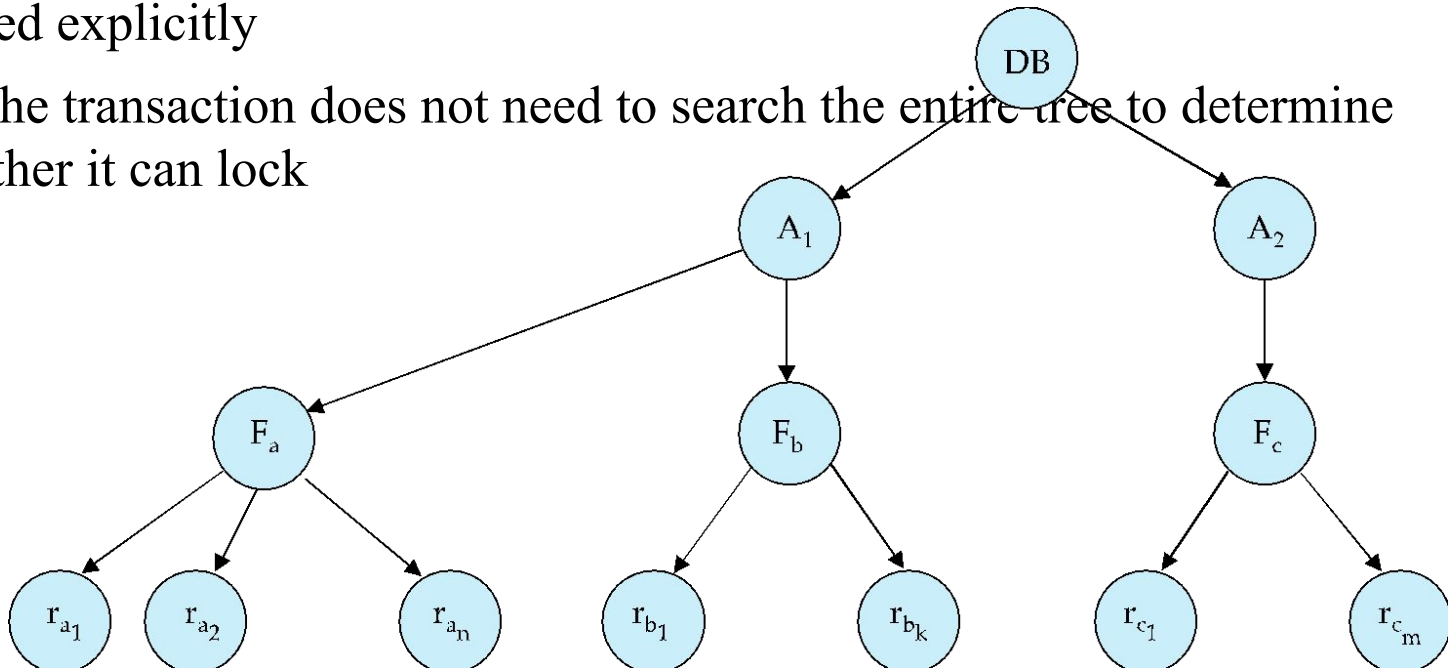
- If Transaction Tk wishes to lock the entire database, it is simply must lock the root of the hierarchy
  - But, Tk should not succeed in locking the root node, since Ti is currently holding a lock on part of the tree (e.g. if, Fb)
  - But, how does the system determine, if the root node can be locked?
    - Solution 1: Search the complete tree,

      Not feasible
    - Solution 2:



Multiple Granularity

- But, how does the system determine, if the root node can be locked?
  - Solution 1: Search the complete tree, Not feasible
  - Solution 2: Introduced New class of lock – INTENSION lock modes
    - If a node is locked in an intention mode, explicit locking is done at a lower level of the tree
    - Intention locks are put on all the ancestors of a node before that node is locked explicitly
    - So, the transaction does not need to search the entire tree to determine whether it can lock



Intention Lock Modes

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

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

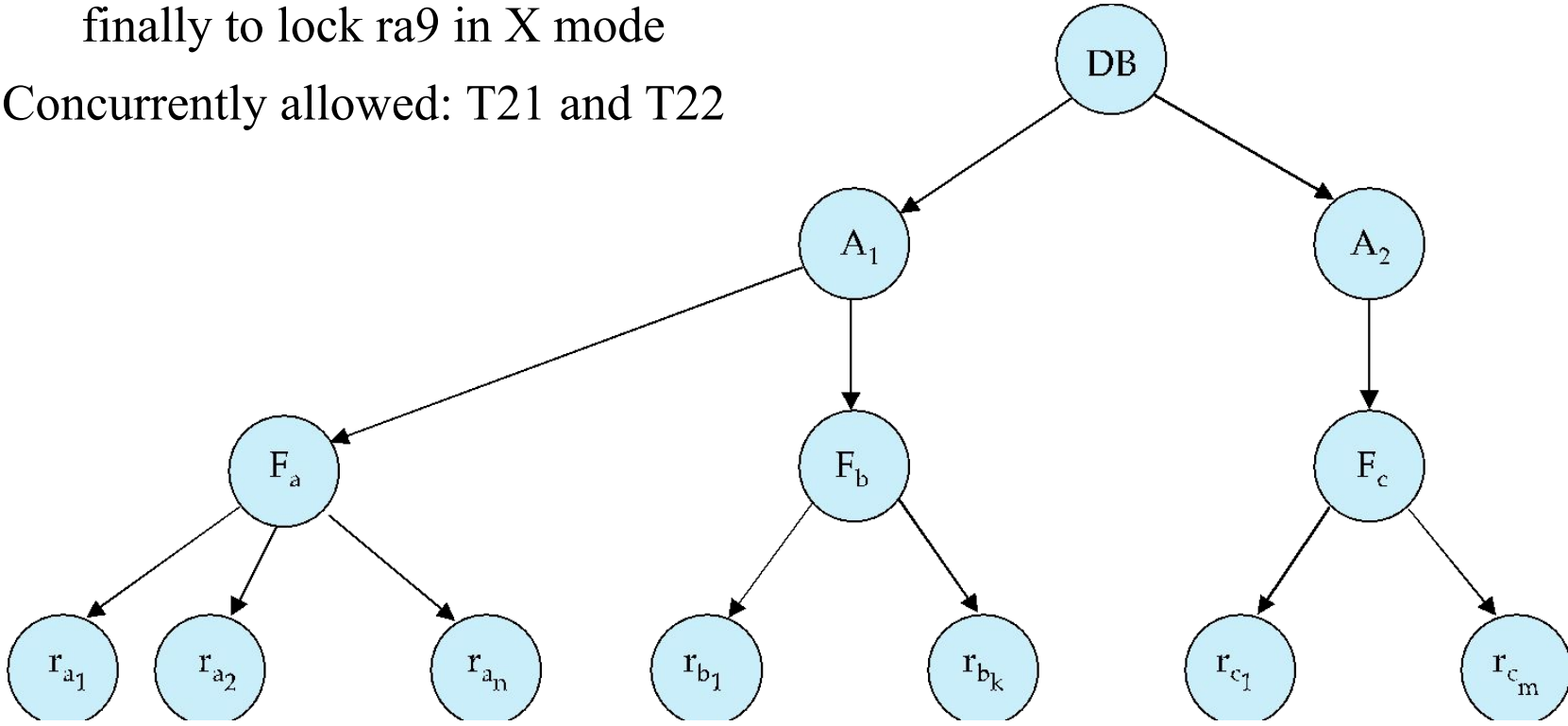
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  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.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Intention Lock Modes

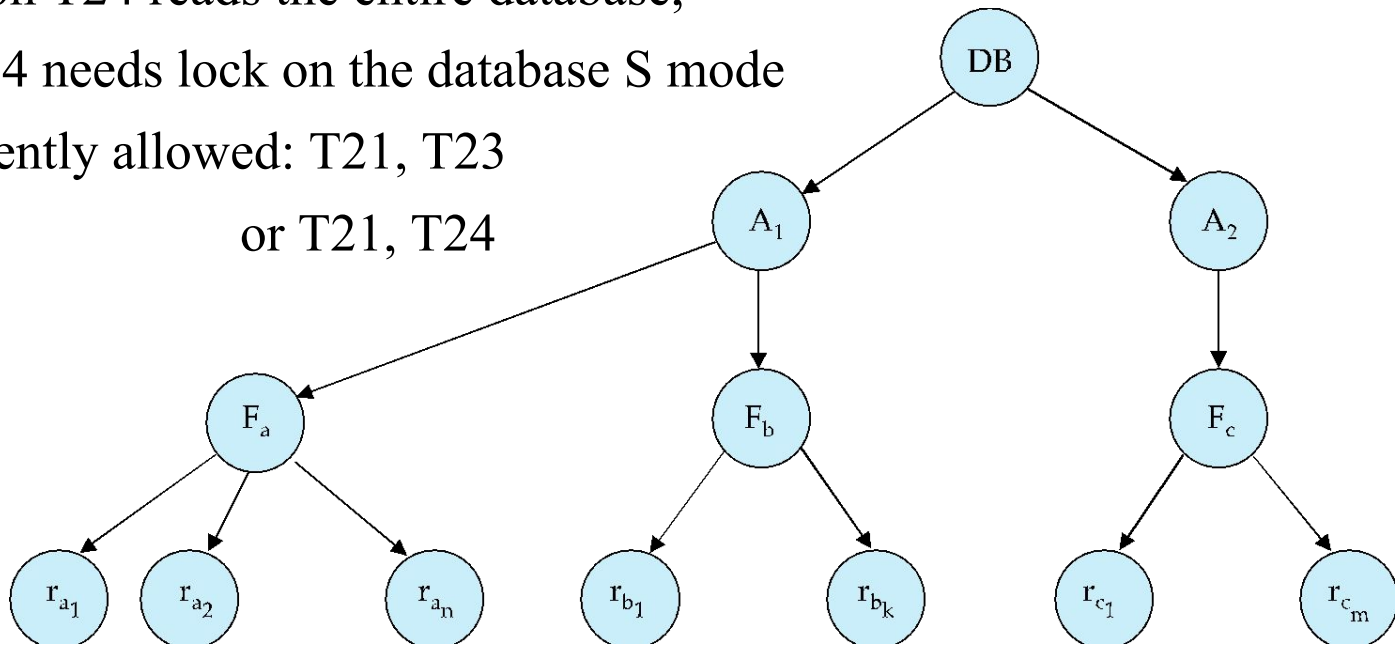
- If Transaction T21 reads a record ra2 in file Fa,
  - Then T21 needs to lock the database area A1 and Fa in IS mode and finally to lock ra2 in S mode
- If Transaction T22 modifies a record ra9 in file Fa,
  - Then T22 needs to lock the database area A1 and Fa in IX mode and finally to lock ra9 in X mode
- Concurrently allowed: T21 and T22



Intention Lock Modes

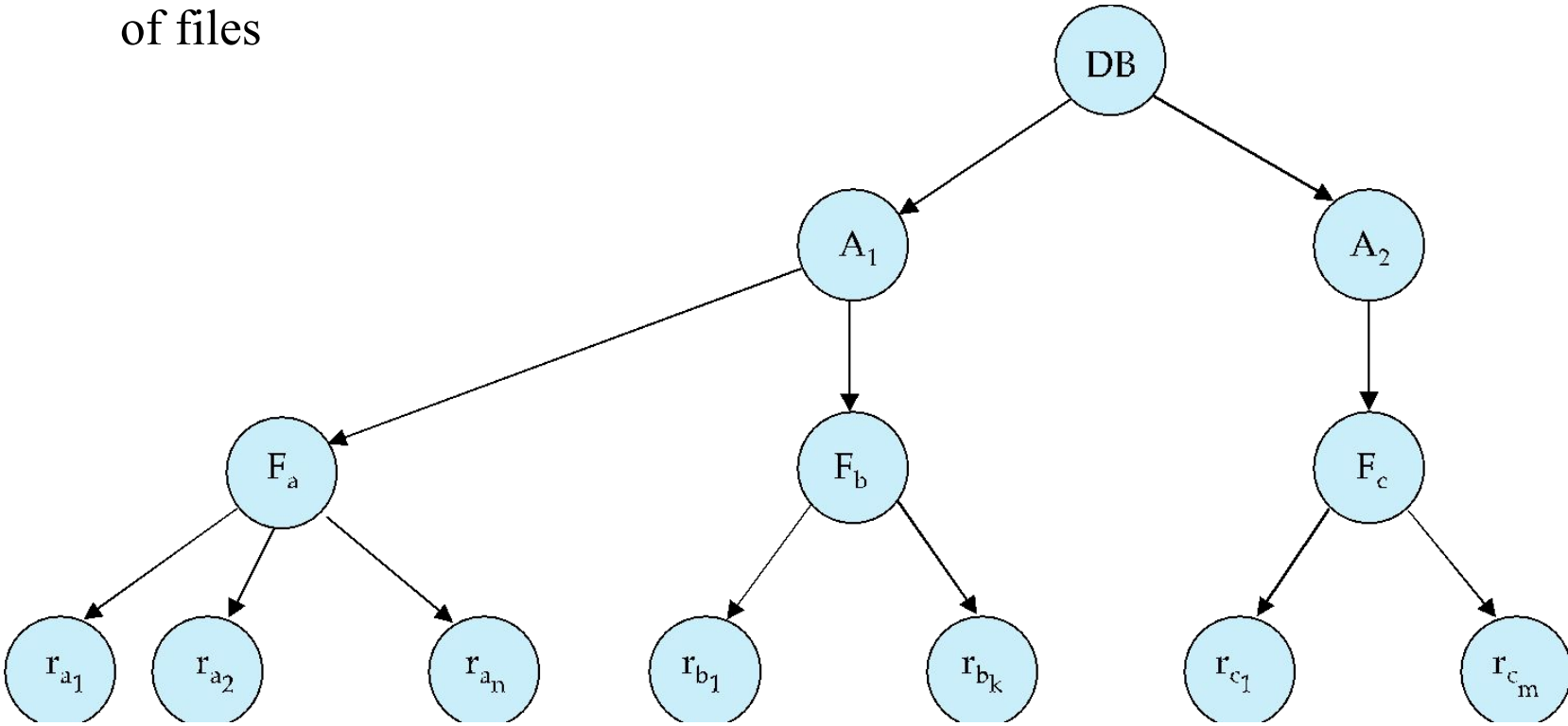
- If Transaction T21 reads a record ra2 in file Fa,
  - Then T21 needs to lock the database area A1 and Fa in IS mode and finally to lock ra2 in S mode
- If Transaction T23 reads all the records in file Fa,
  - Then T23 needs to lock the database area A1 in IS mode and and finally to lock Fa in S mode
- If Transaction T24 reads the entire database,
  - Then T24 needs lock on the database S mode
- Not concurrently allowed: T21, T23

  or T21, T24



Intention Lock Modes

- Enhances concurrency and reduces lock overhead
- Useful in applications, that include a mix of
  - Short transactions that access only a few data items
  - Long transactions that produce reports from an entire file or set of files



View Serializability

Check for Conflict Serializability

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

YES, Conflict Serializable
Serial Schedule: <T1, T2>

View Serializability

- Let *S1* and *S2* be two schedules with the same set of transactions. *S1* and *S2* are **view equivalent** if the following three conditions are met, for each data item *A,*
  1. Schedules S1 and S2 are view equivalent, If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  2. If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  3. If Ti writes final value of A in S1, then Ti also writes final value of A in S2

View Serializability (Cont.)

- **Example: Check View Serializabiltiy:** W3(Z), R2(X), W2(Y), R1(Z), W3(Y), W1(Y)
  - To check for view serializability of a schedule, you must create all possible combinations of the Transactions
  - Here three transactions, then you need to check for these combinations:
    < T1, T2, T3 >
    < T1, T3, T2 >
    < T2, T1, T3 >
    < T2, T3, T1 >
    < T3, T1, T2 >
    < T3, T2, T1 >
- Now the schedule is view serializable if:
  1. *A Tx reads an initial data in a Schedule, the same Tx also should read the initial data in one of the transaction combination.*
     - Here, at least T2 must occur first, though it actually doesn't matter also because no one else writes X, so we still keep all our Tx combinations

View Serializability (Cont.)

- **Example:** W3(Z), R2(X), W2(Y), R1(Z), W3(Y), W1(Y)

2. *A Tx reads a data after another Tx has written in a Schedule, the same Tx also should read the data after another Tx has written it in one of the transaction combination.*

< T1, T2, T3 >
< T1, T3, T2 >
< T2, T1, T3 >
< T2, T3, T1 >
< T3, T1, T2 >
< T3, T2, T1 >

4 Now, this means that **T1 must occur after T3 because T1 reads Z after T3 writes it.** So we **remove all where T1 is before T3**

⇨

- < T1, T2, T3 >
- < T1, T3, T2 >
- < T2, T1, T3 >

⇨

< T2, T3, T1 >
< T3, T1, T2 >
< T3, T2, T1 >

View Serializability (Cont.)

- **Example:** W3(Z), R2(X), W2(Y), R1(Z), W3(Y), W1(Y)

3. *A Tx writes the final value for a data in a Schedule, the same Tx should also write the final data in one of the transaction combination.*

4 Here, **T1 must occur last after T3 or T2** because if not T3 or T2 will overwrites Y that T1 writes in our schedule. So we **remove** < T3, T1, T2 >

< T2, T3, T1 >
< T3, T1, T2 > ⇨
< T3, T2, T1 >

⇨ < T2, T3, T1 >
< T3, T2, T1 >

- So two combinations left satisfy the view serializability, they are:

< T2, T3, T1 >

< T3, T2, T1 >

View Serializability (Cont.)

- **Example:** R1(X),R2(Y),R2(Y),W2(X),W3(Y),R1(X)

1. *A Tx reads an initial data in a Schedule, the same Tx also should read the initial data in one of the transaction combination.*

< T1, T2, T3 >
< T1, T3, T2 >
< T2, T1, T3 >
< T2, T3, T1 >
< T3, T1, T2 >
< T3, T2, T1 >

- Here, at least **T1 must occur first, before T2**, so we remove
- ⇨ <T2, T1, T3>
- <T2, T3, T1>
- <T3, T2, T1>

⇨

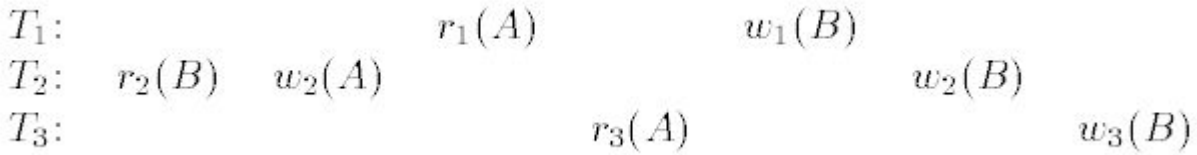
< T1, T2, T3 >
< T1, T3, T2 >
< T3, T1, T2 >

View Serializability (Cont.)

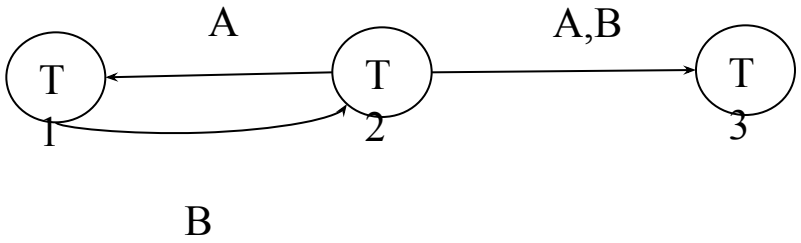
- **Example:** R1(X),R2(Y),R2(Y),W2(X),W3(Y),R1(X)
  2. *A Tx reads a data after another Tx has written in a Schedule, the same Tx also should read the data after another Tx has written it in one of the transaction combination*
     - **T1 reads X after T2 writes,** means that **T2 should occur before T1**
       - But wait a minute, we've just said that **T1 should occur before T2** on the previous condition
       - Because a cycle in the graph also caused
       - Need T1 before T2 and at the same time, need T2 before T1
       - Because of this, **none of the combinations can satisfy these two conditions**, so it is **not view serializable**.

View Serializability (Cont.)

- **Example:**



- **Conflict Serializable?**



- **Conflict is NOT resolved, so it is NOT Conflict Serializable**
- **View Serializable?**
  - **<T2, T1,T3>**
- **YES, view serializable schedule and schedule T2,T1,T3 is view equivalent schedule for the given schedule**

Insert and Delete Operations

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple

Insert and Delete Operations

- Insertions and deletions can lead to the **phantom phenomenon**
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)

    and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)

    (conceptually) conflict in spite of not accessing any tuple in common
  - If only tuple locks are used, non-serializable schedules can result
    - e.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction

Insert and Delete Operations (Cont.)

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information
  - The conflict should be detected, e.g. by locking the information
- One solution
  - Associate a data item with the relation, to represent the information about what tuples the relation contains
  - Transactions **scanning the relation acquire a shared lock** in the data item,
  - Transactions **inserting or deleting a tuple acquire an exclusive lock** on the data item (Note: locks on the data item do not conflict with locks on individual tuples)
- Above protocol provides very low concurrency for insertions / deletions

Caution

- Some statements cannot be rolled back
- Data definition language (DDL) statements, such as create or drop databases, create, drop, or alter tables or stored routines
- Design your transactions not to include such statements
- If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back in such cases by issuing a ROLLBACK statement

Implicit Commit

- Statements That Cause an Implicit Commit
- Implicitly end any transaction active in the current session, as if you had done a COMMIT before executing the statement
- Intent is to handle each such statement in its own special transaction because it cannot be rolled back anyway
- Transaction-control and locking statements are exceptions: If an implicit commit occurs before execution, another does not occur after
- ***Data definition language (DDL) statements that define or modify database objects***
  - ALTER DATABASE , CREATE DATABASE, CREATE INDEX, CREATE PROCEDURE, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, DROP DATABASE, DROP PROCEDURE, DROP TABLE, DROP TRIGGER, DROP VIEW, TRUNCATE TABLE

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 (Cont.)

- 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) ≤ **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)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i))

Timestamp-Based Protocols (Cont.)

- 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).

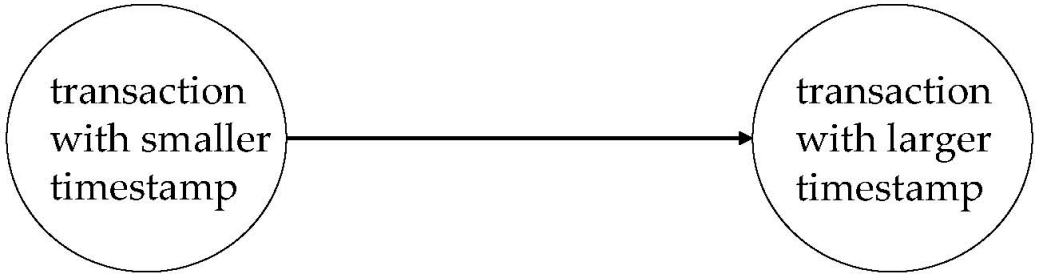
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of 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.

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- Determining such e f operations other than read and write.
  - Operation-conflicts, operation locks

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

End

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Time-stamp Ordering Protocol 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-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.
  - Allows some view-serializable schedules that are not conflict-serializable.

Index Locking Protocol

- Provides higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets
  - Every relation must have at least one index
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - Transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
    - Must update all indices to r
    - Must obtain X-locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Next-Key Locking

- Index-locking protocol to prevent phantoms required locking entire leaf
  - Can result in poor concurrency if there are many inserts
- Alternative: for an index lookup
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent

Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data
- Index-structures are typically accessed very often, much more than other database items
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained
    4. In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node

Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
- Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows. During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- 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 (Cont)

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

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- 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 Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Give logical "snapshot" of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) □ 0 R(Y)□ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) □ 0 R(Y) □ 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits of SI

- Reading is *never* blocked,
  - and also doesn't block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

Snapshot Isolation Anomalies

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

SI In Oracle and PostgreSQL

- **Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL's implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializabilty including handling predicate reads (coming up)

SI In Oracle and PostgreSQL

- Can sidestep SI for specific queries by using **select .. for update** in Oracle and PostgreSQL
  - E.g.,
    1. **select max**(orderno) **from** orders **for update**
    2. read value into local variable maxorder
    3. insert into orders (maxorder+1, …)
  - Select for update (SFU) treats all data read by the query as if it were also updated, preventing concurrent updates
  - Does not always ensure serializability since phantom phenomena can occur (coming up)
- In PostgreSQL versions < 9.1, SFU locks the data item, but releases locks when the transaction completes, even if other concurrent transactions are active
  - Not quite same as SFU in Oracle, which keeps locks until all
  - concurrent transactions have completed

Weak Levels of Consistency

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable:** is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**

Transactions across User Interaction

- Many applications need transaction support across user interactions
  - Can't use locking
  - Don't want to reserve database connection per user
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **select** r.balance, r.version **into** :A, :version
      from r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **update** r **set** r.balance = r.balance + :deposit
      where acctId = 23 **and** r.version = :version
- Equivalent to **optimistic concurrency control without validating read set**
- Used internally in Hibernate ORM system, and manually in many applications
- Version numbering can also be used to support first committer wins check of snapshot isolation
  - Unlike SI, reads are not guaranteed to be from a single snapshot

End of Chapter