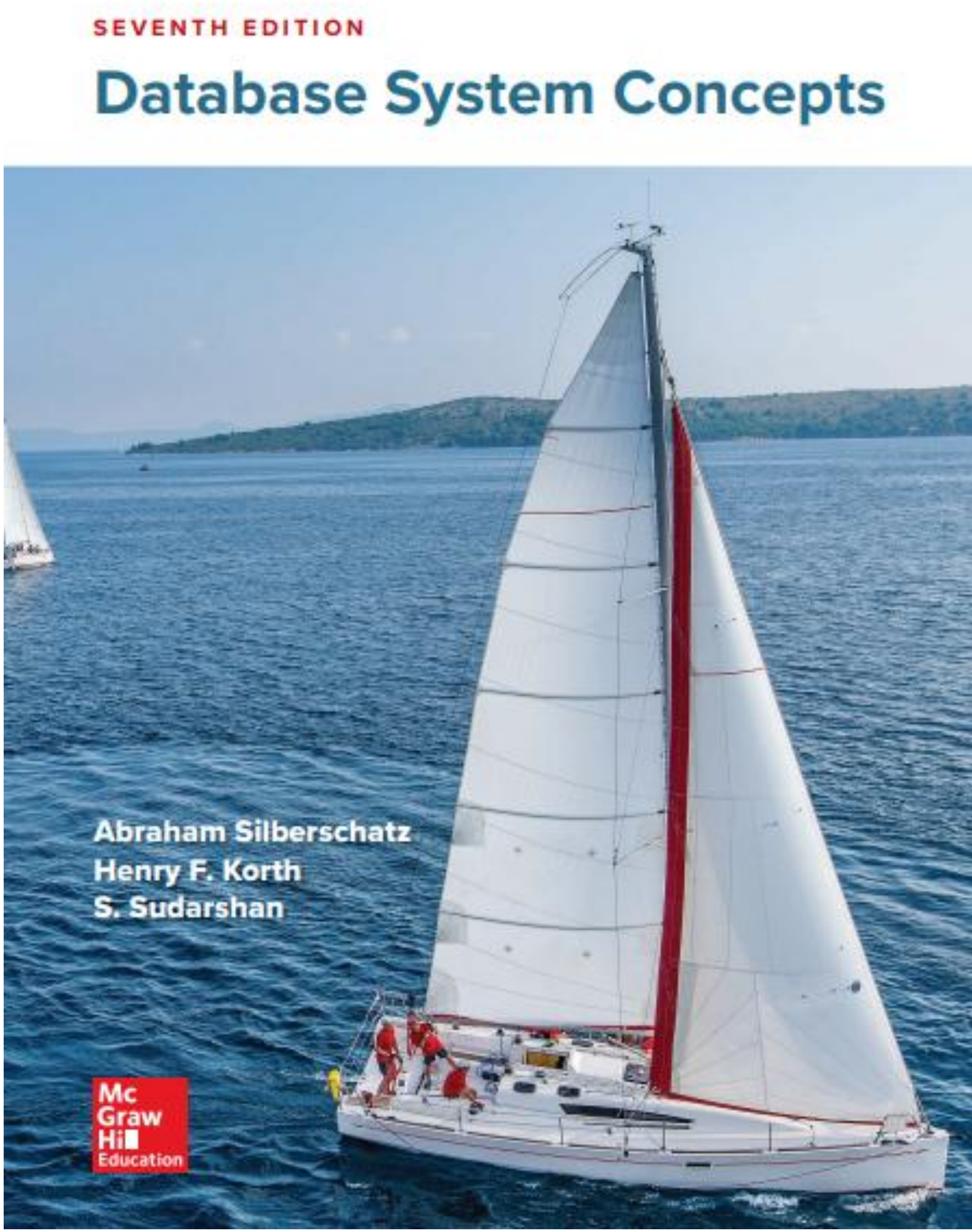


Advance Database -Lecture 3

By Dr. Taghinezhad

Mail:
a0taghinezhad@gmail.com
Website:
ataghinezhad.github.io

Chapter 18 : Concurrency Control

Database System Concepts, 7th Ed.

Outline

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

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

1*.* **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.

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

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (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
- 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.

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 as above is *not sufficient* to guarantee serializability

- Suppose that the values of accounts A and B are \$100 and \$200,. If these two transactions are executed serially, order T1, T2 or the order T2, T1,
- Then transaction T2 will display the value \$300.

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($A + B$).

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

T_1	T_2	concurrency-control manage
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 With Lock Grants

- Same schedule but delayed unlocks:

T_1	T_2

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).

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)	

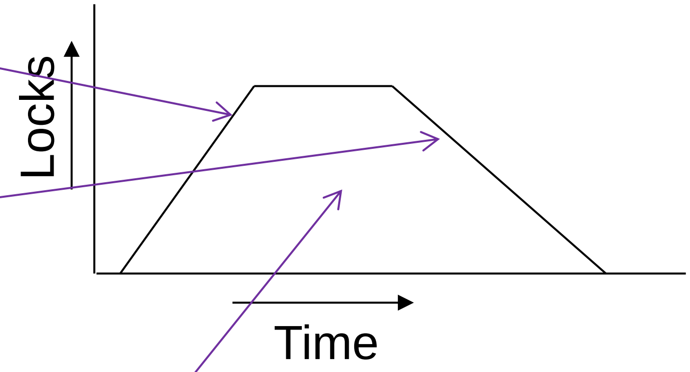
- Consider the partial schedule
- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back
    and its locks released.

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

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



Cascading rollback may occur under two-phase locking.

- Each transaction observes the two-phase locking protocol, but the failure of T5 after the **read(A)** step of T7 leads to cascading rollback of T6 and T7

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)

The Two-Phase Locking Protocol (Cont.)

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

The Two-Phase Locking Protocol (Cont.)

- **Two-phase** locking is **not** a **necessary** condition for **serializability**
  - There are **conflict serializable schedules** that cannot be obtained if the **two-phase locking protocol** is used.
- In the absence of extra information (e.g., **ordering** of **access** to **data**), two-phase locking is necessary 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.***

Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of **transactions** that **follow the protocol**
  - A **protocol ensures** serializability if **all legal schedules** under that **protocol are serializable**

Lock Conversions

- Two-phase locking protocol with **lock conversions**:
  - **Growing 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**)
  - **Shrinking Phase:**
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

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:

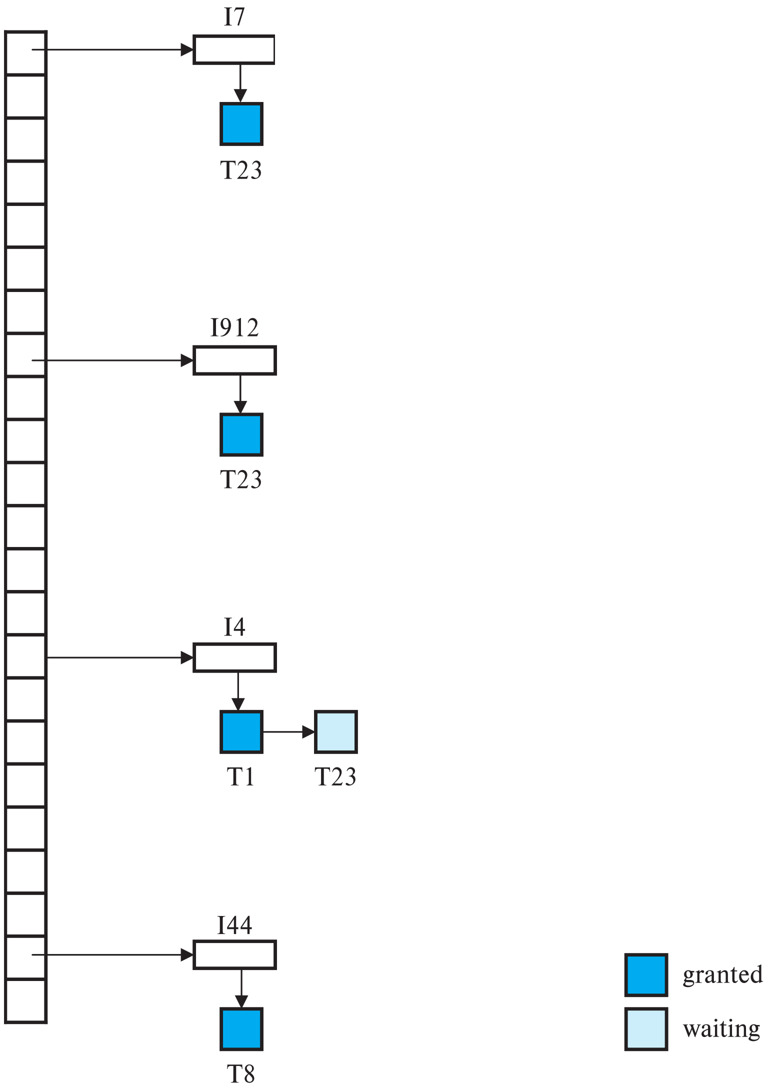
if T_i has a **lock-X** on D
  then
    write(D)
  else begin
    if necessary wait until no other trans. has any lock on D,
    if T_i has a **lock-S** on D
      then
        upgrade lock on D to **lock-X**
      else
        grant T_i a **lock-X** on D
      write(D)
  end;

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

Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can **send lock** and **unlock** requests as **messages**
- 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 an in-memory data-structure called a **lock table** to record **granted** locks and **pending** requests

Lock Table



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

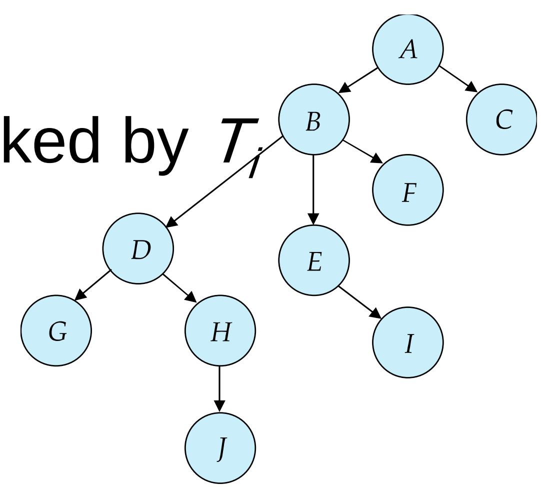
Graph-Based Protocols

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

Tree Protocol

- In the tree protocol, the only lock instruction allowed is **exclusive locks (lock-X).**
- Each transaction $\boldsymbol{T_i}$ can lock a data item at most **once**, and must observe the following rules:

1. The **first lock** by T_i may be on **any data item (the root of data items used)**. **Subsequently**, **a data Q can be locked by $\boldsymbol{T_i}$** only if the **parent of Q is currently locked** by $\boldsymbol{T_i}$.
2. Data items may be unlocked at any time.
3. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



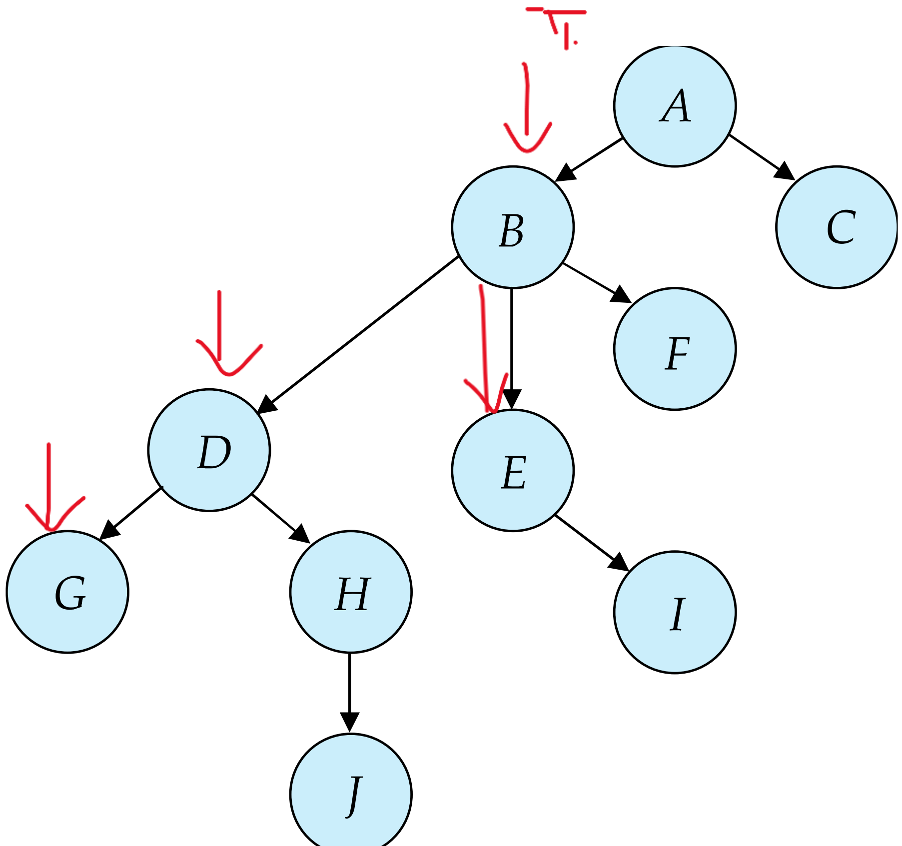
Tree Protocol

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

- One possible schedule in which these four transactions participated appears in Figure
- During its execution, transaction T10 holds locks on two disjoint subtrees. Observe that the schedule of Figure is conflict serializable.
- It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock

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)			

Graph-Based Protocols (Cont.)

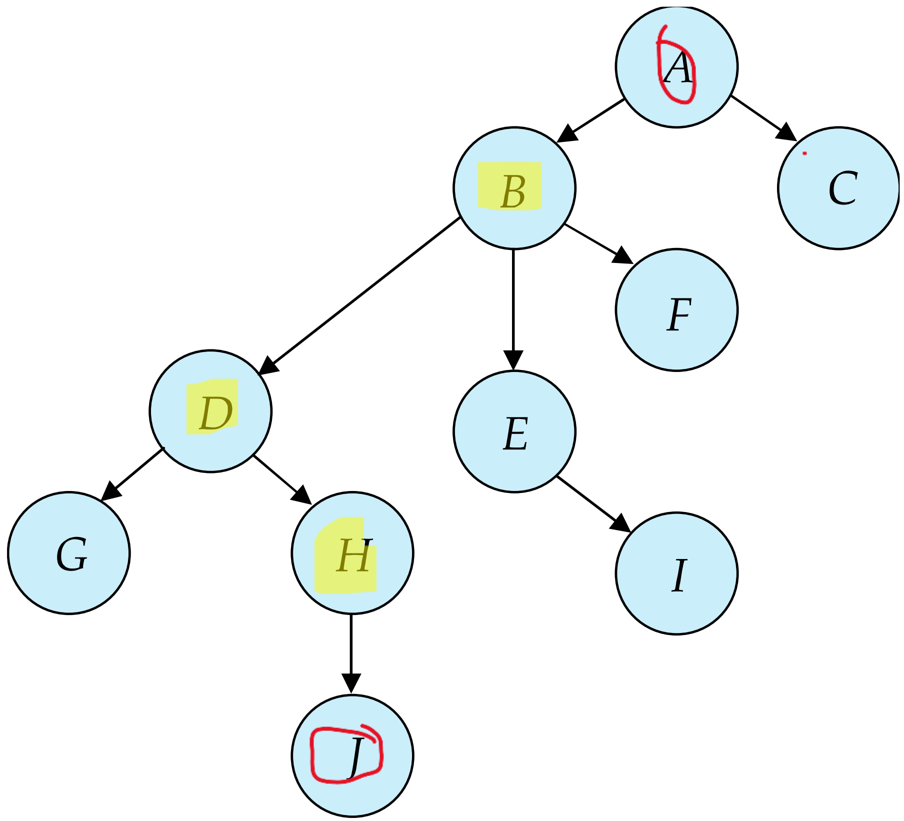
- The tree protocol **ensures conflict serializability** as well as **freedom** from **deadlock**.
- **Unlocking** may occur **earlier** in the **tree-locking** protocol than in the **two-phase locking protocol**.
  - **Shorter waiting times**, and **increase** in **concurrency**
  - Protocol is **deadlock-free**, **no rollbacks** are required

Graph-Based Protocols (Cont.)

- **Drawbacks**
  - Protocol does **not guarantee** **recoverability** or **cascade** freedom
    - Need to introduce **commit dependencies** to **ensure recoverability**
  - **Transactions** may have to **lock data items** that they **do not access**.
    - increased **locking overhead**, and additional **waiting** time
    - potential **decrease** in **concurrency**
- Schedules **not possible** under **two-phase locking** are **possible** under the **tree protocol**, and **vice versa**.

Graph-Based Protocols (Cont.)

- Drawback:
  - a transaction that needs to access data items **A and J** in the database graph of must lock not only A and J, but also data items **B, D**, **H**



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.

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)	

Deadlock Handling

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

More Deadlock Prevention Strategies

- **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 a lock

wait-die scheme, Example

- suppose that transactions **T14, T15, and T16** have timestamps **5, 10, and 15**, respectively.
- If T14 requests a data item held by T15, then what will happen?
- If T16 requests a data item held by T15, then what will happen?

More Deadlock Prevention Strategies

- **wound-wait** scheme — **preemptive**
  - **Older** transaction ***wounds*** (forces rollback) of **younger transaction** instead of waiting for it.
  - **Younger** transactions may **wait for older ones**.
  - **Fewer** rollbacks than ***wait-die*** **scheme**.
- In both schemes, a **rolled back transactions** is restarted with **its original timestamp**.
  - Ensures that **older** transactions have **precedence** over **newer ones**, and **starvation** is thus **avoided**.

Wound-die scheme, Example

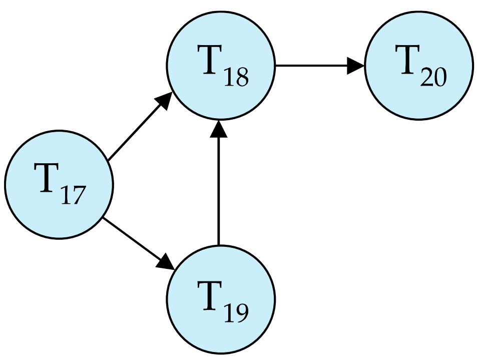
- suppose that transactions **T14, T15, and T16** have timestamps **5, 10, and 15**, respectively.
- if T14 requests a data item held by T15, then what happens?
- If T16 requests a data item held by T15, then what happens?

Deadlock prevention (Cont.)

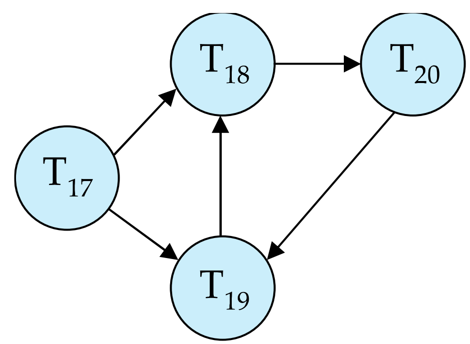
- **Timeout-Based Schemes**:
  - A transaction **waits** for a lock only for **a specified amount** of **time**. After that, the **wait times out** and the transaction is **rolled back.**
  - Ensures that **deadlocks** get **resolved** by **timeout** if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to **determine** good value of the **timeout interval**.
  - **Starvation** is also **possible**

Deadlock Detection in the systems that do not prevent deadlock

- **Wait-for graph**
  - *Vertices:* **transactions**
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the **wait-for graph has a cycle.**
- **Invoke 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 cycle**.
    - **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**: **Rollback** victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- **Starvation** can **happen** (why?)
  - One solution: **oldest transaction** in the **deadlock** set is **never chosen as victim**

Multiple Granularity

- it would be advantageous to group several data items, and to treat them as one individual synchronization unit.
- For example, if a **transaction** $\boldsymbol{T_i}$ needs to access an **entire relation**, and a locking protocol is used to **lock tuples**, then T_i must **lock each tuple in the relation**.
  - **acquiring many such locks is time-consuming**; even worse, the **lock table may become very large** and no longer fit in memory.
  - It would be better **if $\boldsymbol{T_i}$ could issue a single lock request** to lock the **entire relation**.
    - if transaction T_i needs to access only a few tuples, it should not be required to lock the entire relation,
- a mechanism to allow the system to define **multiple levels** of **granularity is needed then**

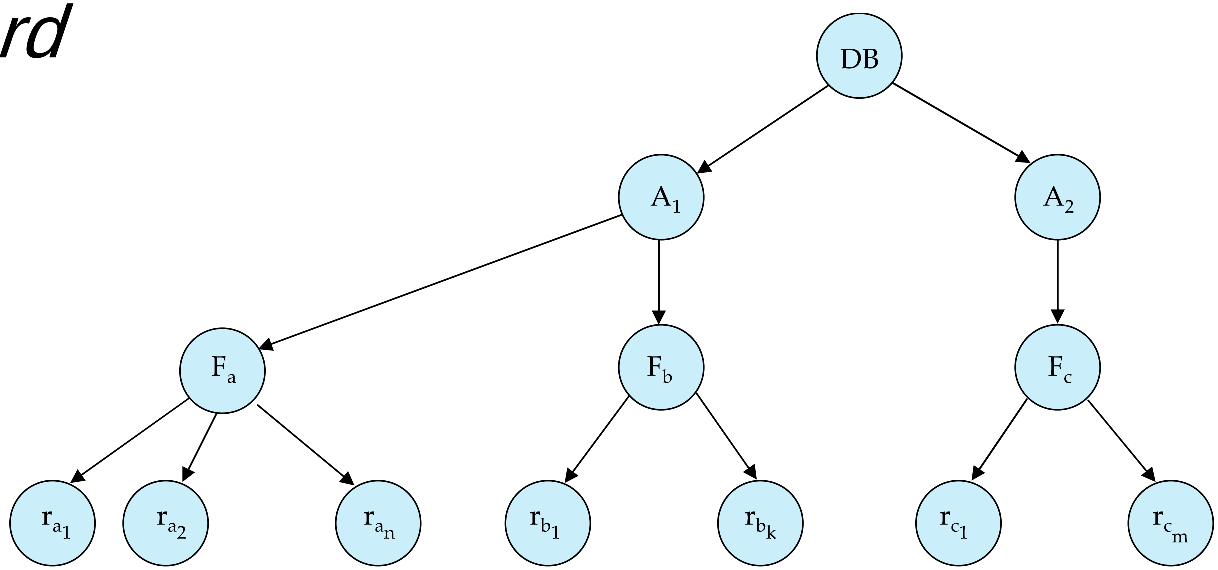
Multiple Granularity

- 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 <u>do not confuse with tree-locking protocol</u>)
- When a transaction **locks a node in the tree *explicitly***, it *implicitly* locks all the **node's descendants** in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

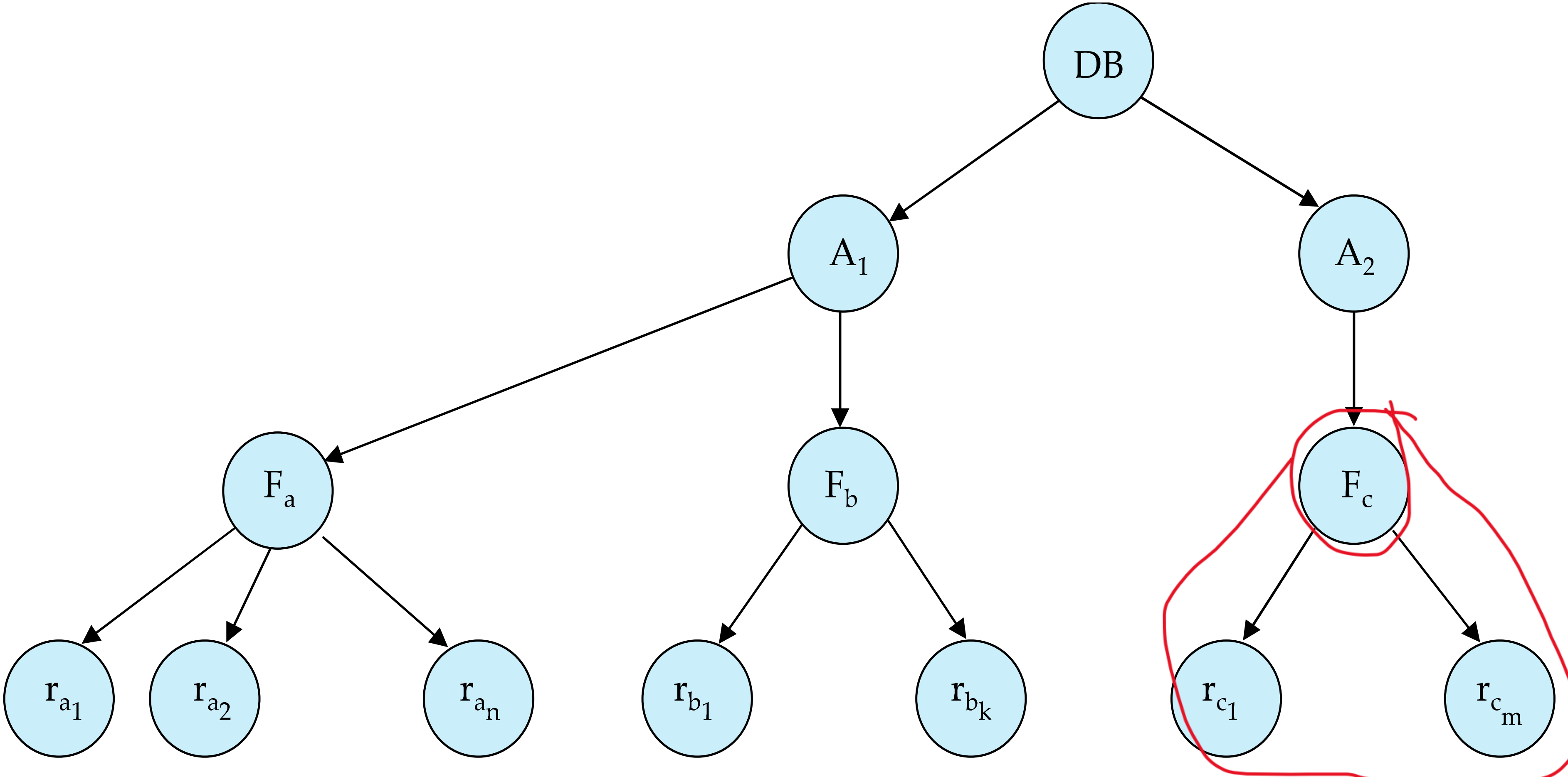
The levels, starting from the coarsest (top) level are

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



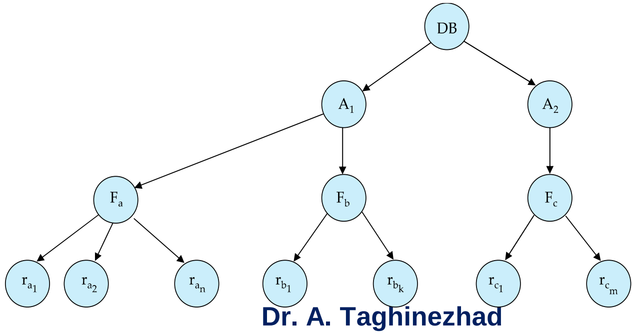
Example of Granularity Hierarchy

- if transaction T_i gets an explicit lock on file F_c of Figure, in exclusive mode, then it has an **implicit lock in exclusive mode** on all the records belonging to that file.



Example of Granularity Hierarchy

- How does the system determine if the root node can be locked?
  - One possibility is for it to search the entire tree.
    - This solution, is not suitable
  - A more efficient way: a new class of lock modes, called intention lock modes.
    - If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity)
    - **Intention** locks are put on all the **ancestors** of a node before that node is locked explicitly. So a transaction does not need to search the entire tree to determine whether it can lock a node successfully



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**): if a node is locked in (SIX) mode, 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:

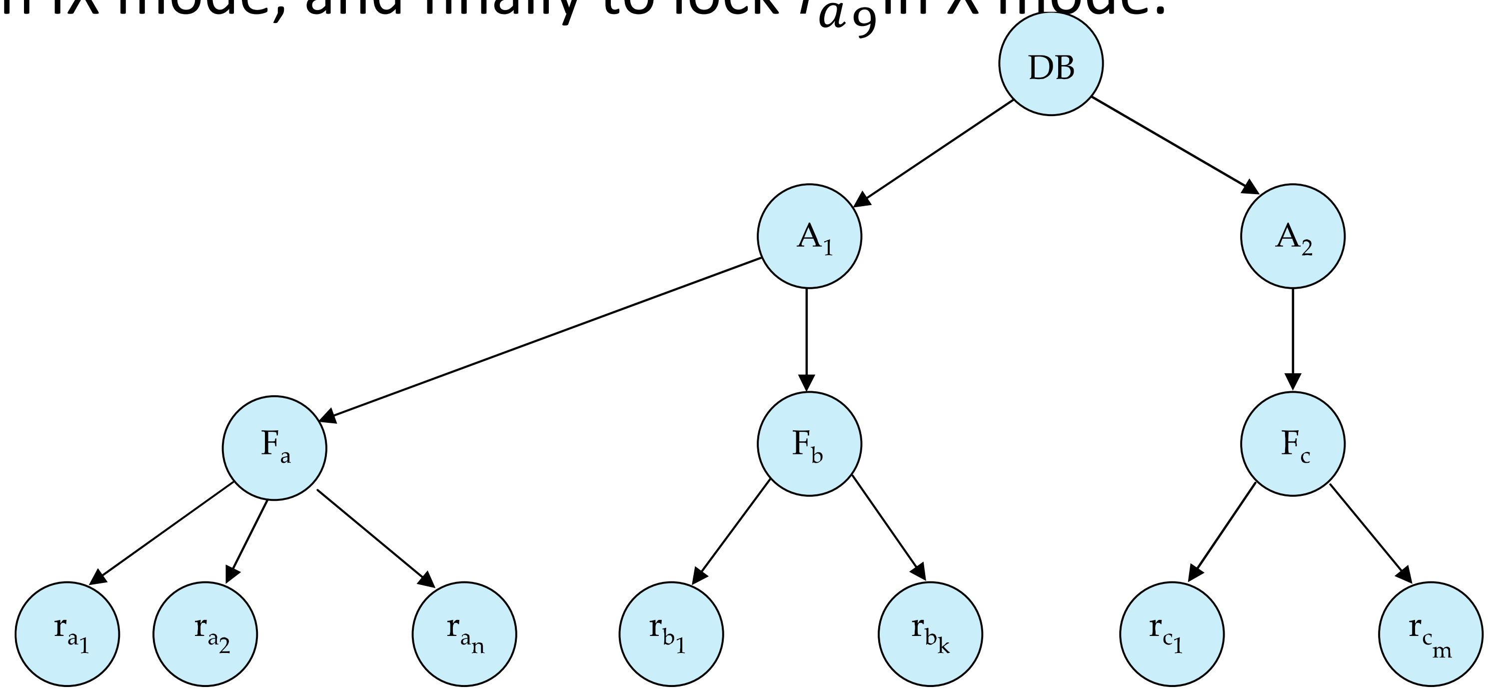
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

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

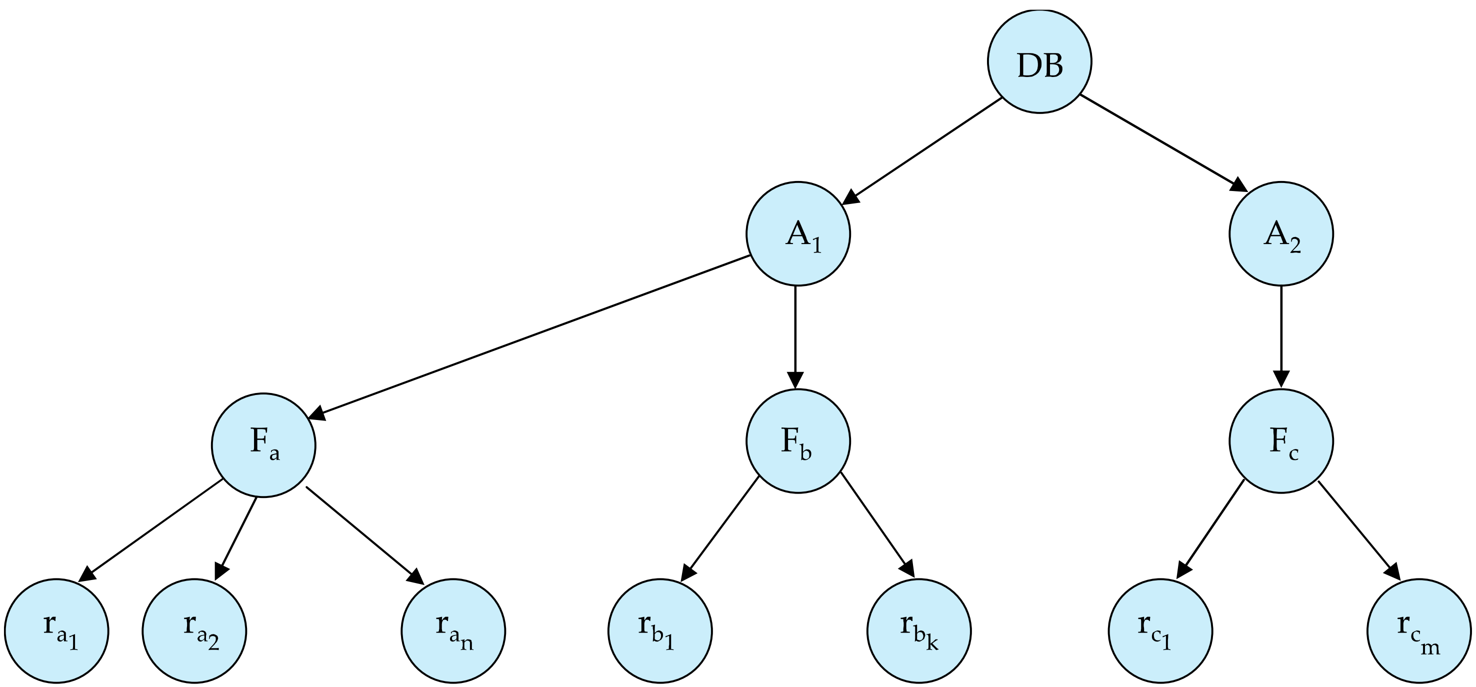
Multiple Granularity Locking Scheme

- Suppose that transaction **T21** reads record r_{a_2} in file F_a . Then, T21 needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T22 modifies record r_{a_9} in file F_a. Then, T22 needs to lock the database, area A1, and file F_a (and in that order) in IX mode, and finally to lock r_{a_9}in X mode.



Multiple Granularity Locking Scheme

- Suppose that transaction **T23** reads all the records in file F_a. Then, T23 needs to lock the database and area A1 (and in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T24 reads the entire database. It can do so after locking the database in S mode.



Timestamp Based Concurrency Control

Timestamp Based Concurrency Control

- The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes.
- a timestamp-ordering scheme
  - For **determining the serializability order** is to select an **ordering among transactions in advance**.
  - With each transaction T_i in the system**, we associate a unique fixed timestamp**, denoted by **TS(T_i).** This timestamp is assigned by the database system before the transaction T_i starts execution

Timestamp-Based Protocols

- Each **transaction** T_i is issued a **timestamp** TS(T_i) when it enters the system.
  - **Each** transaction has a ***unique*** timestamp
  - **Newer** transactions have timestamps strictly **greater** than earlier ones TS(T_i) < TS(T_j)
    - Timestamp could be based on a **logical counter**
    - system clock as the timestamp;
  - Timestamp-based protocols manage concurrent execution such that

    time-stamp order = serializability order.
      - **The system must ensure it is equivalent to a serial Schedule**

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) 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.
- Imposes rules on read and write operations to ensure that
  - Any **conflicting operations** are executed in **timestamp order**
  - **Out of order** operations cause transaction rollback

Timestamp-Based Protocols (Cont.)

- 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) $\geq$ **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).

Timestamp-Based Protocols (Cont.)

- If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Example of Schedule Under TSO

- Is this schedule valid under TSO?
  - We consider transactions T25 and T26. Transaction T25 displays the contents of accounts A + B
  - Transaction T26 transfers $50 from account B to account A, and then displays the contents of both

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Assume that initially:

R-TS(A) = W-TS(A) = 0

R-TS(B) = W-TS(B) = 0

Assume $TS(T_{25}) = 25$ and

$TS(T_{26}) = 26$

Example of Schedule Under TSO

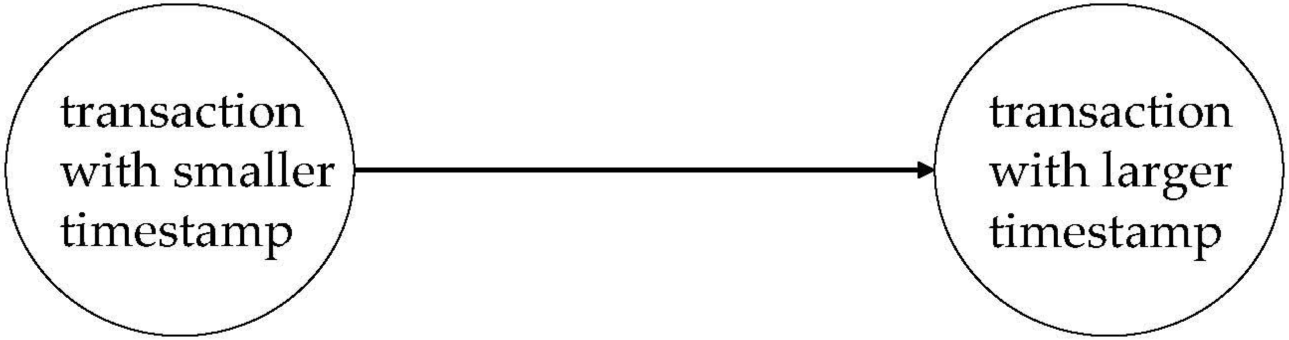
- Is this schedule valid under TSO?
  - How about this one, where initially
    - R-TS(Q)=W-TS(Q)=0
    - TS(T27) < TS(T28)

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

- The read(Q) operation of T27 succeeds, as does the write(Q) operation of T28.
- When T27 attempts its write(Q) operation:
  - Because TS(T27) < W-timestamp(Q), since W-timestamp(Q) = TS(T28). Thus, the write(Q) by T27 is rejected and transaction T27 must be rolled back.
- Any transaction T_j with TS(T_j) > TS(T28) must read the value of Q written by T28, rather than the value that T27 is attempting to write

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.

Recoverability and Cascade Freedom

- 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.**

Validation-Based Protocol

- Idea: can we use commit time as **serialization** order?
- To do so:
  - **Postpone writes** to **end** of transaction
  - Keep track of **data items read/written** by transaction
  - **Validation** performed at **commit time**, **detect** any **out-of-serialization order reads/writes**
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

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.
  - We 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.

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **StartTS**(T_i) : the **time** when T_i started its **execution**
  - **ValidationTS**(T_i): the time when T_i entered its **validation** phase
  - **FinishTS**(T_i) : the time when T_i **finished** its **write phase**
- Validation tests use above timestamps and read/write sets to ensure that serializability order is determined by validation time
  - Thus, **TS(T_i) = ValidationTS(T_i)**
- **Validation-based** protocol has been found to give **greater degree of concurrency** than locking/TSO if probability of conflicts is low.

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**:
  - **finishTS**(T_i) < **startTS**(T_j)
  - **startTS**(T_j) < **finishTS**(T_i) < **validationTS**(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**:
  - First condition applies when execution is not concurrent
    - The writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - If the second condition holds, execution is concurrent, T_j does not **read any item written by T_i.**

Schedule Produced by Validation

- Example of schedule produced using validation Suppose that TS(T25) < TS(T26).

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)

- The validation phase succeeds in the schedule.
- Note that the writes to the actual variables are performed only after the validation phase of T26. Thus, T25 reads the old values of B and A, and this schedule is serializable.