SE2032

Database Management Systems

Concurrency Control Techniques

LEARNING OUTCOMES

- By the end of this lesson you should be able to

  - Describe the importance of locking mechanism
  - Identify the types of locks used in DBMS
  - Describe and identify the locking mechanism used in a schedule
  - Describe the techniques used to prevent dead lock
  - Describe and identify the Phantom problem

Concurrency Control

Concurrency Control ensures that simultaneous transactions on a DBMS do not interfere with each other. It also prevents common issues like dirty reads, lost updates, and non-repeatable reads. The key features include deadlock avoidance, ensuring serializability of transactions, and maintaining data integrity.

Concurrency Control Techniques

- Lock Based Protocols (Shared/ Exclusive)
- Tree Protocols
- Timestamp Based Protocols (Basic Timestamp Ordering/ Strict Timestamp Ordering/ Thomas's Write Rule)
- Multiple Granularity

Lock-Based Protocols

- Locks
  - It is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied.
  - Are used as a means of synchronizing the access concurrent transactions to the database items.

Types of Locks

1. Binary Locks
    - Simple but restrictive and so are not used in practice
    - Can have two states, locked and unlocked
    - A distinct lock is associated with each database item.
    - If a data item X is locked (i.e. LOCK(X)= 1) any other transaction that needs X is forced to wait.
    - If a data item X is unlocked (i.e. LOCK(X) := 0 (UNLOCK(X)) so that X may be accessed by other transactions.

- For binary lock every transaction must obey the following
  - A transaction T must issue the operation `lock_item(X)` before any `read_item(X)` or `write_item(X)` operations are performed in T.
  - A transaction T must issue the operation `unlock_item(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.
  - A transaction T will not issue a `lock_item(X)` operation if it already holds the lock on item X.
  - A transaction T will not issue a `unlock_item(X)` operation unless it already holds the lock on item X.

- For binary locks
  - At most one transaction can hold the lock on a particular item.
  - No two transaction can access the same item concurrently.

2. Shared(S)/Exclusive(X) (or Read/Write) Locks

- There are three locking operations

  read_lock(X)(or Lock_S), write_lock(X) (or Lock_X), unlock(X)

- A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item. No transaction can **write** the item while it's under a shared lock.

- A **write-locked item** is called an **exclusive lock**, because **only one transaction** can **read and write** the data item. No other transaction can even read it while it's locked.

- For shared/exclusive locking scheme every transaction must obey the following

1. A transaction T must issue the operation `read_lock(X)`/`Lock_S` or `write_lock(X)`/`Lock_X` before any `read_item(X)` operation is performed in T.

2. A transaction T must issue the operation `write_lock(X)`/`Lock_X` before any `write_item(X)` operation is performed in T.

3. A transaction T must issue the operation `unlock(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.

4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read lock or a write lock on item X.

5. A transaction T will not issue a `write_lock(X)` operation if it already holds the read lock or write lock on item X.

6. A transaction T will not issue a `unlock(X)` operation unless it already holds the read lock or write lock on item X.

- If a transaction has an exclusive lock on an item, it can both read and update it. To prevent interference from other transactions, only one transaction can hold an exclusive lock on an item at any given time.

- Conditions 4 and 5 could be relaxed in order to allow lock conversions.
- To upgrade a lock
- To downgrade a lock

4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read lock or a write lock on item X.

5. A transaction T will not issue a `write_lock(X)` operation if it already holds the read lock or write lock on item X.

- Shared lock (S) or Read_lock()
  - If a transaction T_i has obtained a shared-mode lock on item Q, then T_i can read but cannot write Q.
- Exclusive lock (X) or Write_lock()
  - If a transaction T_i has obtained an exclusive-mode lock on item Q, then T_i can both read and write Q.
- Every transaction should request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix

- Binary locks or shared/exclusive locks does not guarantee serializability.

- Consider the example

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X:=X+Y$;	$Y:=X+Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: $X=20$, $Y=30$

Result of serial schedule T_1 followed by T_2 :

$X=50$, $Y=80$

Result of serial schedule T_1 followed by T_2 :

$X=70$, $Y=50$

(c)

T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);	

	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	Y:=*X*+*Y*;
	write_item(*Y*);
	unlock(*Y*);

write_lock(*X*);
read_item(*X*);
X:=*X*+*Y*;
write_item(*X*);
unlock(*X*);	

Time

Initial values : X = 20, Y =30

Result of schedule S:
X=50, *Y*=50
(nonserializable)

- Schedule S follows the shared/exclusive locking rules.

- But it is not serializable.

- The items Y in T_1 and X in T_2 were unlocked too early.

3. Two-Phase Locking

- To follow the two-phase locking *all* locking operations (read_lock, write_lock) must precede the *first* unlock operation in the transaction.

Such a transaction can be divided into two- phases

- Expanding or growing phase
  - New locks on items can be acquired, but non can be released.
  - If lock conversion is allowed upgrading is done in this phase

- Shrinking phase
  - Existing locks can be released, but no new locks can be acquired.
  - If lock conversion is allowed downgrading is done in this phase

- The transactions T_1 and T_2 do not follow the two-phase locking protocol.

(c)

Time

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

If we enforce the 2PL then

- `write_lock(X)` operation follows the `unlock(Y)` operation in T_1
- `write_lock(Y)` operation follows the `unlock(X)` operation in T_2

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

- Now the schedule involving interleaved operations shown in the slide 20 is not permitted. This is because T1’ will issue its write_lock(X) before it unlocks Y; consequently, when T2’ issues its read_lock(X), it is forced to wait until T1’ issues its unlock(X) in the schedule.

- It can be proved that, if every transaction in a schedule follows the basic 2PL, the schedule is guaranteed to be serializable.

- Problem that may be introduced by 2PL protocol is deadlock.

- Two Phase locking can introduce some undesirable effects
- These are
  - Deadlocks
  - Starvation

- Dead Lock
  - Occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T’ in the set.
  - Each transaction in the set is on a waiting queue.
  - Consider the example

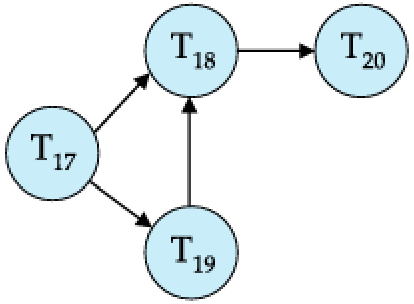
- T_1’ is on the waiting queue for X which is locked by T_2’
- T_2’ is on the waiting queue for Y which is locked by T_1’
- Neither T_1’ nor T_2’ nor any other transaction can access items X and Y

(a)

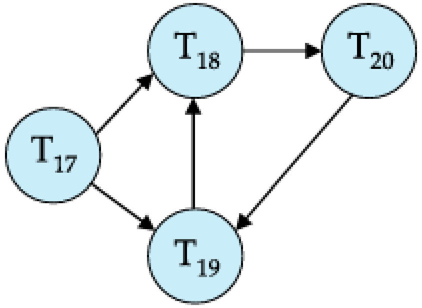
Time ↓	T_1'	T_2'
	read_lock(*Y*);	
	read_item(*Y*);	
		read_lock(*X*);
		read_item(*X*);
	write_lock(*X*);	
		write_lock(*Y*);

Deadlock Detection

- Wait-for graph
  - ✓Vertices: transactions
  - ✓Edge from Ti →Tj : if Ti is waiting for a lock held in conflicting mode byTj
- 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 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

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.

- Variations of two-phase locking
  - The one described is known as the basic 2PL
  - Conservative (or static) 2PL
    - Requires a transaction to lock *all* the items it accesses *before the transaction begins execution*, by predeclaring its read-set and write-set.
  - Strict 2PL
    - A transaction T does not release any of it exclusive locks until after it commits or abort, So no other transaction can read or write an item that T is written by unless T has committed.
  - rigorous 2PL
    - A transaction T does not release any of its locks (exclusive or shared) until after it commits or abort,

If we enforce the Strict – 2PL then

T_1'	T_2'
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
read_item (X);	read_item (Y);
$X:=X+Y$;	$Y:=X+Y$;
write_item (X);	write_item (Y);
unlock (Y);	unlock (X);
unlock (X);	unlock (Y);

Tree protocol

- if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database.
- There are various models that can give us additional information.
- The simplest model requires that we have prior knowledge about the order in which the database items will be accessed.
- impose a partial ordering $\rightarrow$ on the set $D = \{d_1, d_2, \ldots, d_h\}$ 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.

- partial ordering - $d_i \rightarrow d_j$ may be
  - the result of either the logical or the physical organization of the data, or
  - it may be imposed solely for the purpose of concurrency control.

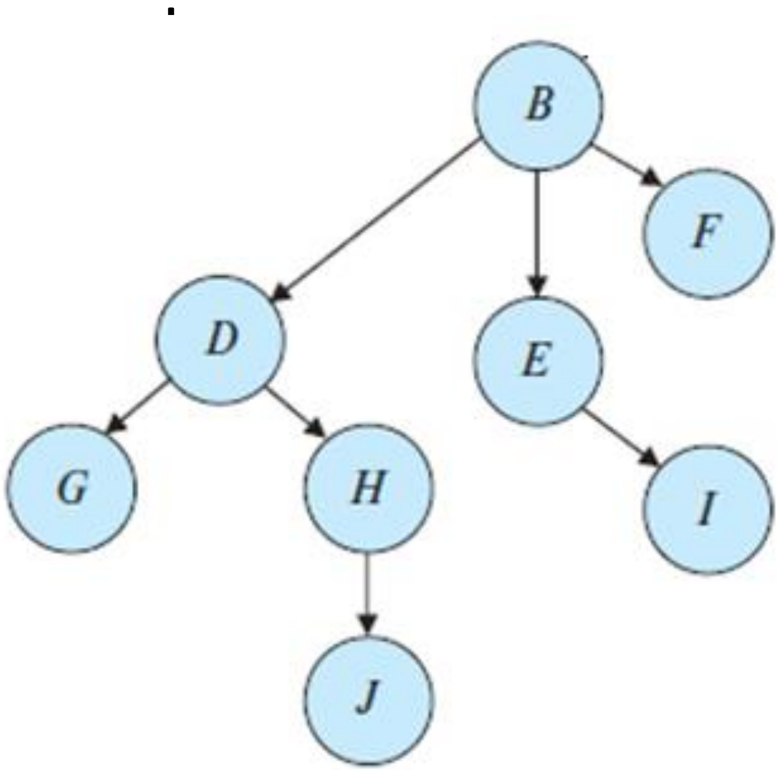
- The partial ordering implies that the set D is viewed as a directed acyclic graph, called a database graph.

- a simple protocol, which is restricted to employ only exclusive locks is called the **tree protocol**

- In the tree protocol, the only lock instruction allowed is write_lock or lock-X. Each transaction 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.
  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.

All schedules that are legal under the tree protocol are conflict serializable.

Consider the following tree-structured databas



*T*10: W_L(*B*); W_L(*E*); W_L(*D*); unlock(*B*); unlock(*E*); lock(*G*); unlock(*D*); unlock(*G*).

*T*11: W_L(*D*); W_L(*H*); unlock(*D*); unlock(*H*).

*T*12: W_L(*B*); W_L(*E*); unlock(*E*); unlock(*B*).

*T*13: W_L(*D*); W_L(*H*); unlock(*D*); unlock(*H*).

Follows the tree protocol.

- This is a serializable schedule under the tree protocol

T10	T11	T12	T13
W_L(B)			
	W_L(D)		
	W_L(H)		
	Unlock(D)		
W_L(E)			
W_L(D)			
Unlock(B)			
Unlock(E)			
		W_L(B)	
		W_L(E)	
	Unlock(H)		
W_L(G)			
Unlock(D)			
			W_L(D)
			W_L(H)
			Unlock(D)
			Unlock(H)
		Unlock(E)	
		Unlock(B)	
Unlock(G)			

- This protocol does not ensure recoverability. Can be modified to not permit the release of exclusive locks until the end of the transaction

Timestamps based Protocol

- A timestamp is a unique identifier created by the DBMS to identify a transaction
- Timestamp values are assigned in the order that they are submitted
- It is the transaction start time.
- Timestamp of transaction T is TS(T)

Timestamp Ordering Algorithm

- Read_TS(X) the read timestamp of item X; the **largest** timestamps of transactions that have successfully read item X. i.e. read_TS(X) = T where T is the **youngest** transaction that has read X successfully.

- Write_TS(X) the write timestamp of item X; the **largest** timestamps of transactions that have successfully written item X. i.e. write_TS(X) = T where T is the **youngest** transaction that has written X successfully.

Basic TO algorithm

- Transaction T issues a write_item(X) :
  - If TS(T) < read_TS(X) or if TS(T) < write_TS(X), then abort and roll back T and reject the operation.
    - Some younger transaction with a timestamp grater than TS(T) and hence after T in the timestamp ordering has already read or written the value X, before T had a chance to write X, thus violating the timestamp ordering
  - If the condition above does not violate then execute the write_item(X) of T and set write_TS(X) = TS(T).

Basic TO algorithm

- Transaction T issues a read_item(X) :
  - If TS(T) < write_TS(X), then abort and roll back T and reject the operation.
    - Some younger transaction with a timestamp grater than TS(T) and hence after T in the timestamp ordering has already written the value of X, before T had a chance to read X
  - If the condition above does not violate then execute the read_item(X) of T and set read_TS(X) = max(read_TS(X), TS(T)).

- Problem – cascading rollback

Strict TO

- Transaction T issues read_ietm(X) or write_item(X) s.t. TS(T) > write_TS(X)
  - Delay the read or write until the transaction T' that wrote the value of X has committed or aborted

Thomas’s write Rule

- Modifies the checks for write_item(X) operation as follows:

- If TS(T) < read_TS(X), then abort and rollback T and reject the operation
- If TS(T) < write_TS(X), then do not execute the write operation but ignore and continue processing.
- If the above conditions do not occur execute the write_item(X) and set write_TS(X) to TS(T).

Example

Consider three transactions **T1**, **T2**, and **T3** with timestamps TS(T1) = 3, TS(T2) = 7, and TS(T3) = 12.

- The operations on item **A** happen in this order:
  - T1: write(A)
  - T2: read(A)
  - T3: write(A)
  - T2: write(A)

Assume initially:

- read_TS(A) = 0
- write_TS(A) = 0.

(a) Under **Basic Timestamp Ordering Protocol**, state for each operation whether it is allowed or causes abort.
(b) Under **Strict Timestamp Ordering Protocol**, explain how execution would differ.
(c) Using **Thomas' Write Rule**, determine whether the operations would succeed, abort, or be ignored.
(d) What is the final value of **read_TS(A)** and **write_TS(A)** after all successful operations?

(a)

T1 (TS1 = 3)	T2 (TS2 = 7)	T3 (TS3 = 12)
write(A) : 3<0 or 3<0 false, Execute write_TS(A) = 3		
	read(A) : 7<3 false, Execute, read_TS(A) = max(0,7) = 7	
		write(A) : 12<7 or 12<3 false, Execute, write_TS(A) = 12
	write(A) : 7<7 or 7<12 true, abort	

(b) Under Strict TO, transaction T2 will be delayed without abort it.

(C) Under Thomas's Write Rule, transaction T2 will be ignored and continue without abort.

(d) read_TS(A) = 7, write_TS(A) = 12

Granularity of items

- Until now we used the term ‘data item’ without specifying its exact meaning
- When speaking about concurrency control, data item can be:
  - A field of a database record
  - A database record,
  - A disk block
  - A whole file
  - A whole database
- As coarser data item granularity, as more contention between transitions, and less productivity (more waits and aborts)

- As finer data granularity, as higher locking overhead of the DBMS lock manager (due to many items in the db)
- The best size depends on the type of the transaction
  - If the transaction accesses a small number of records, then
    - Data item = record
  - If the transaction accesses a many records in the same file, then
    - Data item = file or block
- Some DBMS automatically change granularity level with regard to the number of records a transaction is accessing

Multiple Granularity Level Locking

- Data items are of different sizes (depends on the transaction)
- Can represent in a form of a granularity hierarchy (a tree form)

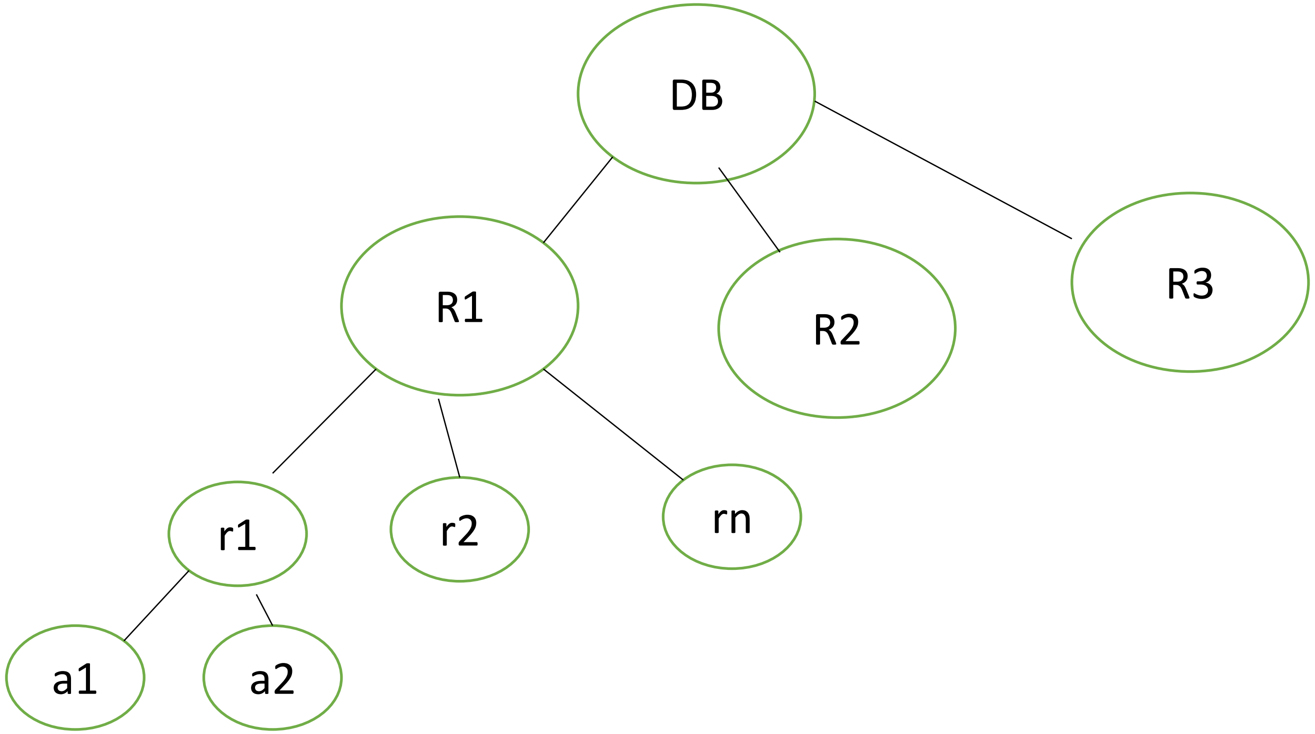
If T1 Locks R1

the lower levels are also locked

When a node is locked all children

are automatically locked.

Implicit lock on children



- Problems

1. Say T1 has locked r1

If another transaction T2 needs to lock say a2

Need to check whether the parent/ancestor are

Locked. Must start from root

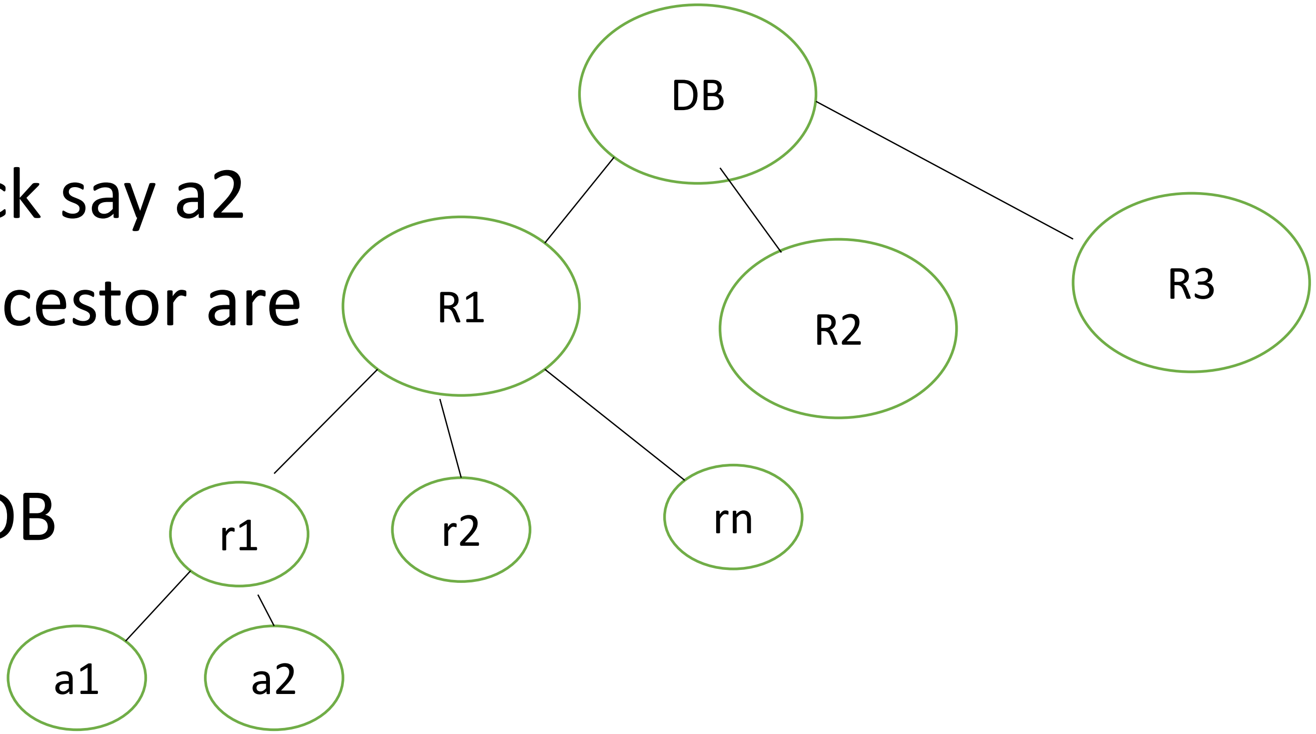
2. Say a transaction T3 needs to lock DB

    need to check whether any of the

    descendants are locked.

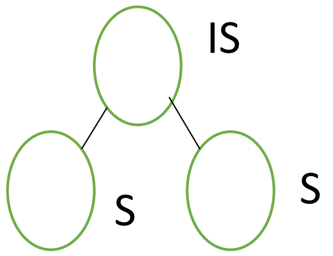
    Search the entire tree

So need additional types of locks

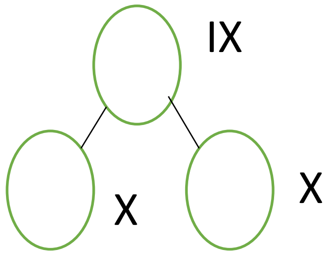


- Intention Lock modes
  - The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node’s descendants
  - There are three types of Intention Locks:

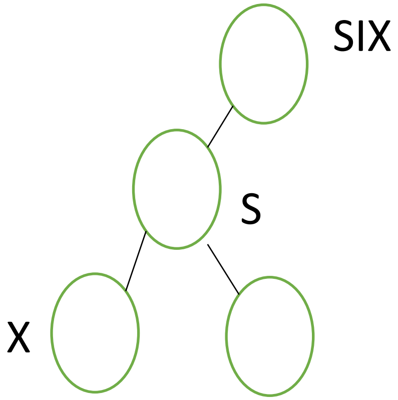
1. Intention-shared(IS) – indicates that a shared lock(s) will be requested on a descendant node(s)



2. Intention-exclusive(IX) – indicates that an exclusive lock(s) will be requested on some descendant node(s)



3. Shared-intention-exclusive(SIX) – indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant nodes



Five Locks

S, X, IS, IX, SIX

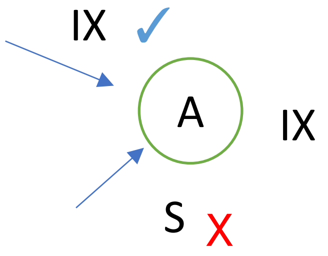
Compatibility Matrix (Table)

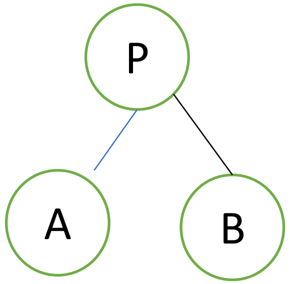
	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

The multiple granularity locking (MGL) protocol consists of the following rules:

So a transaction that attempts to lock a node must follow these rules:

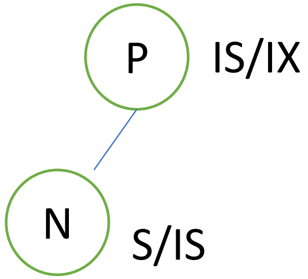
1. The lock compatibility must be adhered to.



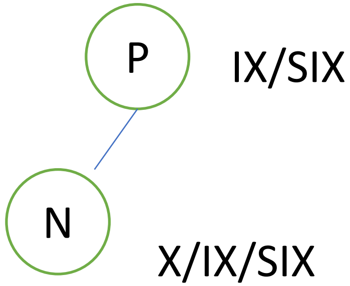


2. The root of the tree must be locked first, in any mode

3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.

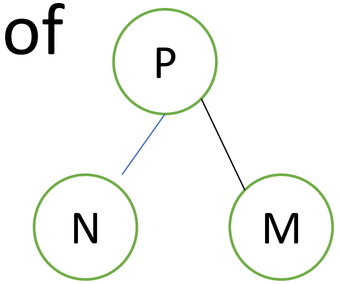


4. A node N can be locked by a transaction T in X, IX or SIX mode only if the parent node N is already locked by transaction T in either IX or SIX mode.



5. A transaction T can lock a node only if it has not unlocked any node (to enforce 2PL protocol)

6. A transaction T can unlock a node, only if none of the children of Node N are currently locked by T



Other Concurrency control issues

- When a new record is inserted
  - Cannot be accessed until after the item is created and the insertion operation is completed.
  - In a locking environment
    - a lock for the item can be created and set to exclusive mode
  - For a time stamp based protocol
    - The read and write timestamps of the new item are set to the time stamp of the creating transaction

- Delete operation
  - In a locking environment
    - an exclusive lock must be obtained before the transition can delete the item
  - For a time stamp based protocol
    - The protocol must ensure that no later transactions has read or written the item before allowing the item to be deleted.

Dynamic Databases and the Phantom Problem

- If the collection of database objects is not fixed but can grow and shrink through the insertion and deletion of objects, we must deal with a problem known as the **phantom problem**.

Consider the following sailor table

sid	sname	age	rating
S123	Gamunu	80	2
S124	Piyal	71	1
S125	Kamal	63	2
S126	Gayan	54	1

Consider two transactions T1 and T2

T1 wants to find the oldest sailor of rating = 1 and rating = 2

T2 wants to insert/delete records of the sailor table

Step 1: T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (age = 71)

Step 2: Next, T2 inserts a new sailor onto a new page (rating = 1, age = 96)

Step 3: T2 locks pages with rating = 2, deletes oldest sailor with rating = 2 (age = 80), commits releases all locks

Step 4: T1 now locks all pages with rating = 2, and finds oldest sailor (age = 63)

- No consistent DB state where T1 is "correct"

- If T1 followed by T2 gets age as 71 and 80
- If T2 followed by T1 gets age as 96 and 63
- The interleaved execution is not identical to any serial execution of T1 and T2 (even though both transactions follow strict 2PL)

- T1 found oldest sailor with rating = 1 before modification by T2
- T1 found oldest sailor with rating = 2 after modification by T2

- Conflict serializability guarantees serializability only if the set of objects is fixed

- Problem:
- T1 implicitly assumed that it had locked the set of all sailor records with rating = 1
- Assumption only holds if no sailor records are added while T1 is executing

Adding records to unlock area and conflict occurs with ongoing transactions called as Phantom problem.

- How the problem can be handled
  - If there is an index on the *rating* field, *T1* can obtain a lock on the index page
  - If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added to ensure that no new records with rating = 1 are added.