Unit-4 (Contd..)

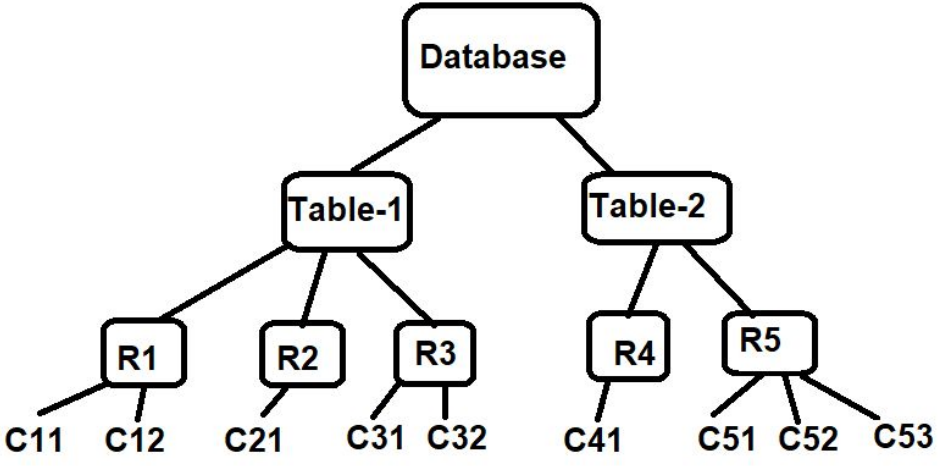
- Multiple Granularity of locks
  - Granularity hierarchy
  - Intention Lock modes
  - Compatibility matrix for lock modes
- Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Detection
  - Deadlock Recovery
- Multiversion schemes
  - Multiversion using Timestamp ordering
  - Multiversion using 2-phase locking

Situation when transaction holds multiple data item

- A Transaction Ti access more than one data item from database.
- Data item may be a column's value, row,table or entire database
- Database → Tables → Rows → Columns
- If a Transaction needs to access a database:
  - Should it needs to lock entire database? If so, it needs to apply multiple locks on its tables,rows and columns
- If a Transaction needs to access a specific data item (Q):
  - Should it needs to lock only that data item (Q) or entire database?
- Both in the above cases, the Intensity of locks differ

Granularity hierarchy

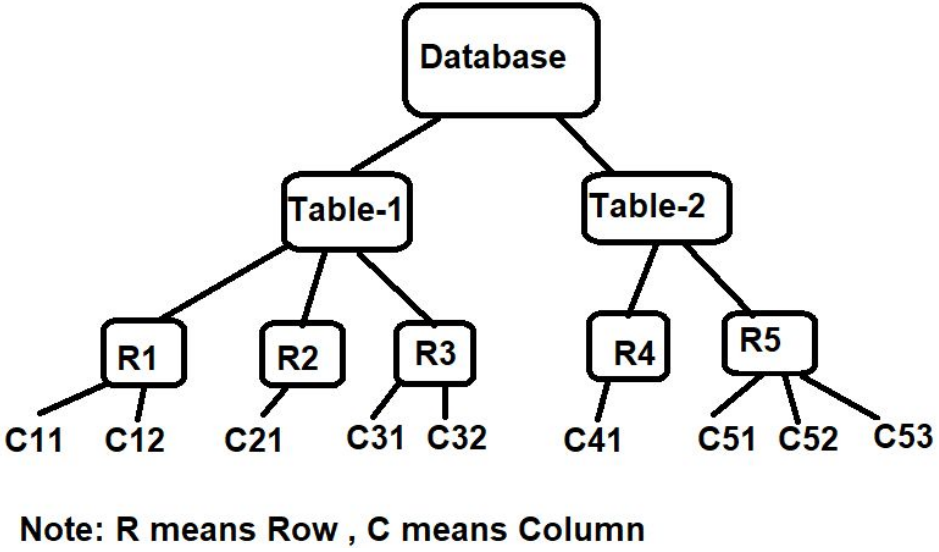
- Represents the level/effect of locks on data items requested by transaction.
- A database is a collection of tables,rows and columns grouped in a hierarchical manner.
- All Data items are grouped under sub-tree nodes and sub-tree nodes are grouped under root node.
- Two locks can be applied by default:
  - Shared lock – For read operations
  - Exclusive lock – for both read and write operations



Note: R means Row , C means Column

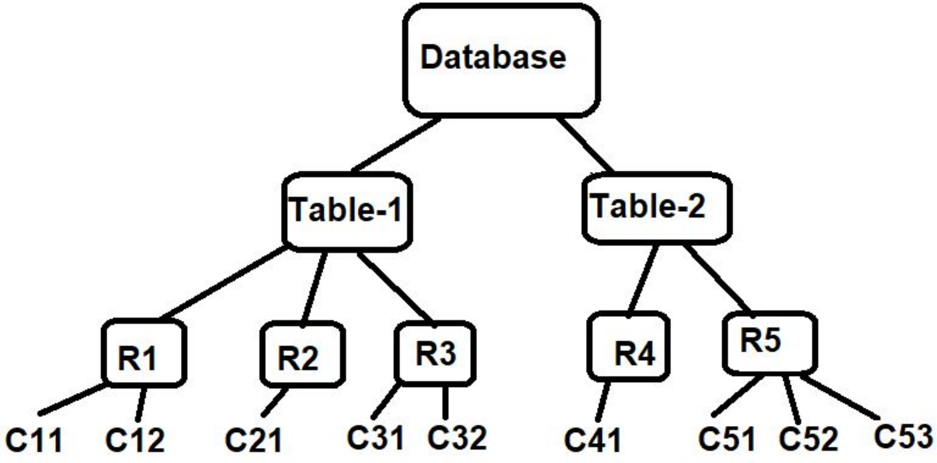
Granularity hierarchy

- A Lock can be explicit lock or implicit lock.
- Example: If Ti explicitly call shared lock on R5 then its descendants automatically will be applied with implicit locks.
- If Tj request X-lock on C51 explicitly then it will be rejected due to an implicit already applied before by Ti.
- If Tk request a lock on Database,it will be rejected as its child node R5 is locked by Ti.



Granularity hierarchy

- How Transactions Ti, Tj or Tk determines that other transactions are holding locks on data items?
  - They should traverse from root node to current node and check the lock is compatible or not with other transactions.
  - Obviously it's a time consuming process
- To overcome this, we use three more locks called **'Intention lock modes'**



Note: R means Row , C means Column

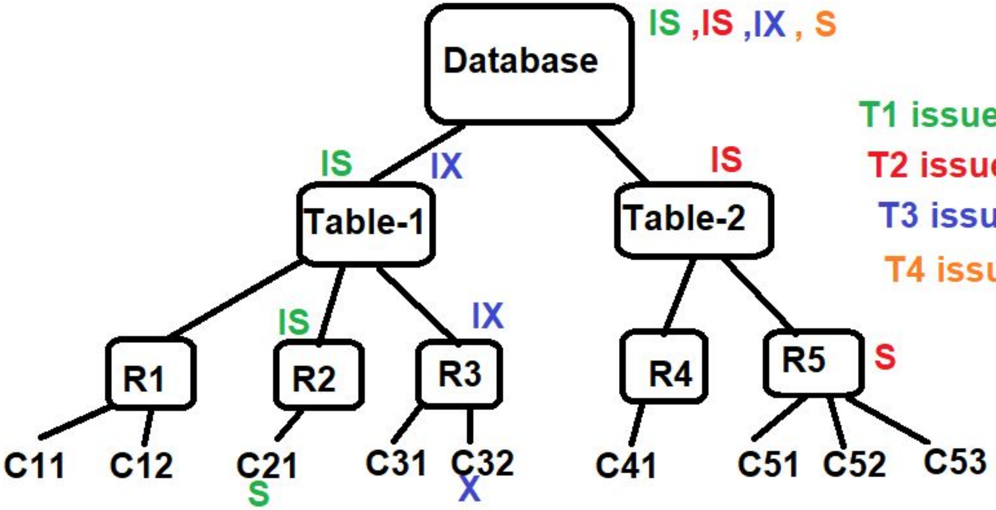
Granularity Locks

- **Shared lock (S)** – to read data item
- **Exclusive lock (X)** – to read/write data item
- **Intention Shared lock (IS)** – If Node N is Shared locked,then all its ancestors are locked in IS lock
- **Intention Exclusive lock (IS)** – If Node N is Exclusive locked,then all its ancestors are locked in IX lock
- **Shared Intention Exclusive lock (SIX)** – If Node N is Shared locked, its sub-Node is Exclusive locked then ancestors of N are locked in SIX mode lock

Lock Compatibility matrix for Multiple Granularity of locks

	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

Example



T1 issues read(C21)

T2 issues read(R5)

T3 issues write(C32)

T4 issues read(Database)

Note: R means Row , C means Column

which of them run concurrent?
T1, T2
T1,T3
T3,T4
T1,T2,T4

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.

Handling Deadlocks

- What is Deadlock – A situation in which two transactions Ti, Tj mutually depends on each other to access data item which will be a never ending transaction.
- Example: T1 → T2 → T3 → T4 → T1
- Some transactions facing Deadlocks can be recoverable but some cannot.
- Deadlock Prevention
- Deadlock Detection
- Deadlock Recovery

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 prevention

Deadlock prevention protocols ensure that the system will never enter into a deadlock state.

Using Timestamp ordering mechanism of transactions in order to predetermine a deadlock situation

a) Wait-Die Scheme -- allows older transaction to wait and younger one to die.

If a transaction Ti requests to lock a data item, which is already held with a conflicting lock by another transaction Tj, then one of the two possibilities may occur:

- If TS(Ti) < TS(Tj) then Ti is allowed to wait till data item is available.
- If TS(Ti) > TS(Tj) then Ti dies and would be restarted later

b) Wound-Wait Scheme -- allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

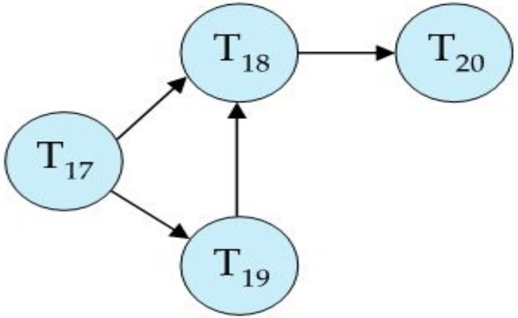
- If TS(Ti) < TS(Tj) then Ti forces Tj to rollback and ti waits till data item available. i.e Ti wounds Tj.
- If TS(Ti) > TS(Tj) then Ti is forced to wait till data item is available.

Deadlock prevention

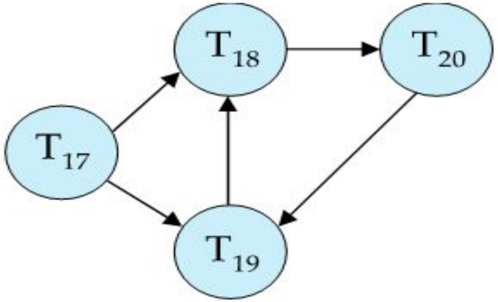
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily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

Deadlock Detection

- **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.
    - Using **Victim Selection**, Select a transaction as victim that will have minimum risk
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for

Multi version Schemes

- It is a mechanism of creating a new copy of data item when a write operation is performed by a transaction on Data Item-Q.
- The new copy is called as Version and each version is identified as {Q1,Q2,Q3…Qn} where Q1,Qn are the older,newer versions of data item Q respectively.
- Idea behind this scheme:
  - To allow a transaction Ti to perform read operation on Q while other Transaction Tj performs conflicting write operation on same data item Q concurrently.

Multi version Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
- Key ideas:
  - 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 issuing the read request, and return the value of the selected version.
  - **read**s never have to wait as an appropriate version is returned immediately.

Multi version schemes

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

Data Item - Q

Content	W-Timestamp	R-Timestamp

Multi version using Timestamp Ordering

- 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
     - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) = TS(T_i),
  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. Otherwise, a new version Q_i of Q is created
        - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

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.
  - Read of a data item returns the latest version of the item
  - The first **write** of Q by T_i results in the creation of a new version Q_i of the data item Q written
    - W-timestamp(Q_i) set to ∞ initially
  - When update transaction T_i completes, commit processing occurs:
    - Value **ts-counter** stored in the database is used to assign timestamps
      - **ts-counter** is locked in two-phase manner
    - Set TS(T_i) = **ts-counter** + 1
    - Set W-timestamp(Q_i) = TS(T_i) for all versions Q_i that it creates
    - **ts-counter** = **ts-counter +** 1

Multiversion Two-phase locking

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- 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.

Multiversion Timestamp ordering - Example

Multiversion Timestamp ordering

	TST	TST
	5	10
Step	T1	T2
1	Read X	
2	Write X	
3		Read X
4		Write X
5	Read X	
6	Write X	
7		Write X

step 0

Content	WTS	RTS
X0	0	0

Step 1 Read by T1

If RTS<TST Update RTS=TST

If RTS>TST read older version

Content	WTS	RTS
X0	0	5

Step 2 Write by T1

If TST<RTS then Roll back

If TST=WTS Overwrite content

If TST>WTS Create new version

Change both WTS and RTS=TST

Content	WTS	RTS
X1	5	5

Step 3 Read by T2

If RTS<TST Update RTS=TST

Content	WTS	RTS
X1	5	10

Step 4 Write by T2

TST>WTS Create new version

Change both WTS and RTS=TST

Content	WTS	RTS
X2	10	10

Step 5 Read by T1

If RTS>TST read older version

where TST=RTS

Do not update anything

Content	WTS	RTS
X1	5	5

Step 6 Write by T1

Check latest version of X

Content	WTS	RTS
X2	10	10

TST<RTS so Roll Back

Since T1 had aborted T2 has to abort as it has read contents written by T1

Remove version X0 and X1 as their WST<least TST

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait

Undesirable properties

Reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one.

Second, the conflicts between transactions are resolved through rollbacks, rather than through waits.