UNIT - IV (Part-II)

Concurrency Control: Lock-based Protocols, Timestamp-based Protocols, Validation-based Protocols, Multiple Granularity, Multi-version Schemes, Deadlock Handling, Insert and Delete Operations, Weak Levels of Consistency, Concurrency of Index Structures.

CONCURRENCY CONTROL

LOCK-BASED PROTOCOLS:

A locking **protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

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.

<u>Lock-compatibility matrix:</u>

	S	X
S	true	false
X	false	false

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

Example of a transaction performing locking:

T2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A+B*)

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

Pitfalls of Lock-Based Protocols: 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 *T3* nor *T4* can make progress — executing lock**-S** *(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing lock**-X** *(A)* causes *T3* to wait for *T4* to release its lock on *A*. Such a situation is called a **deadlock**. To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released. The potential for deadlock exists in most locking protocols.

- **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: This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase: The transaction may obtain locks and transaction may not release locks.

Phase 2: Shrinking Phase: The transaction may release locks and transaction may not obtain locks.

- This protocol assures Serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **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.
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (Ex: ordering of access to data), two-phase locking is needed for conflict Serializability in the following sense:
  Given a transaction *Ti* that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

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.

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

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

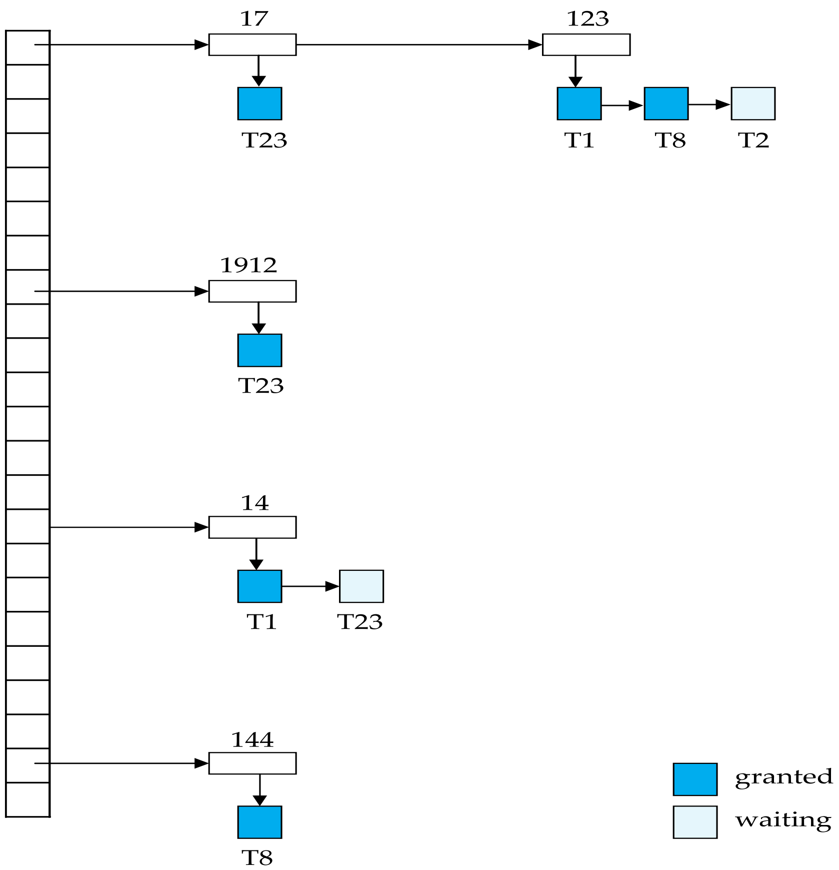
All locks are released after commit or abort

Implementation of Locking:

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

Lock Table: The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked. Lock table also records the type of lock granted or requested

- ➢ Blue rectangles indicate granted locks, Light Blue rectangles ones indicate waiting requests
- ➢ 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



Timestamp-Based Protocols:

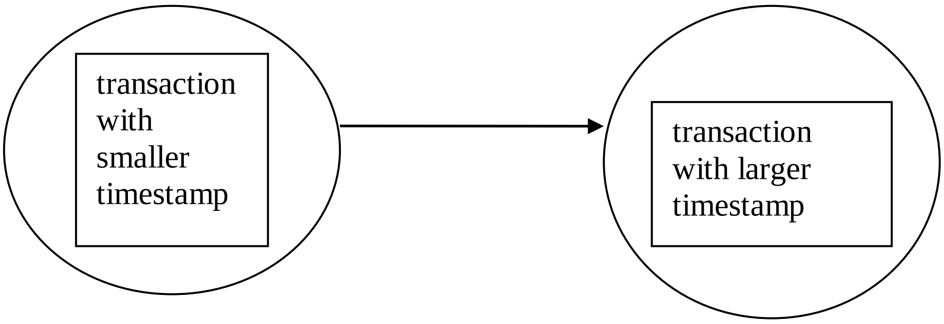
Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS (*Ti*), a new transaction *Tj* is assigned time-stamp TS (*Tj*) such that TS (*Ti*) < TS (*Tj*).

- The protocol manages concurrent execution such that the time-stamps determine the Serializability order.
- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  - **W-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **write** (*Q*) successfully.
  - **R-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **read** (*Q*) successfully.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction Ti issues a **read**(*Q*)
  - If TS(*Ti*) ≤ **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten.
    - Hence, the **read** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*)≥ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(*Ti*)).
- Suppose that transaction *Ti* issues **write**(*Q*).
  - If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.
    - Hence, this **write** operation is rejected, and *Ti* is rolled back.
  - Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

Example Use of the Protocol: A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T1	T2	T3	T4	T5
				read(*X*)
	read(Y)			
read(Y)				
		write(*Y*)		
		write(Z)		
				read(Z)
	read(*X*)			
	abort			
read(*X*)				
		write(Z)		
		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.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose *Ti* aborts, but *Tj* has read a data item written by *Ti*
  - Then *Tj* must abort; if *Tj* had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by *Tj* 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 *Ti* attempts to write data item *Q*, if TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of {*Q*}.
  - ***Rather than rolling back Ti 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

Execution of transaction *Ti* is done in three phases.

1) Read and execution phase: Transaction *Ti* writes only to temporary local variables
2) Validation phase: Transaction *Ti* performs a "validation test'' to determine if local variables can be written without violating Serializability.
3) Write phase: If *Ti* is validated, the updates are applied to database; otherwise, Ti 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

Each transaction Ti has 3 timestamps

1. Start(Ti) : the time when Ti started its execution
2. Validation(Ti): the time when Ti entered its validation phase
3. Finish(Ti) : the time when Ti finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).
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 *Tj*

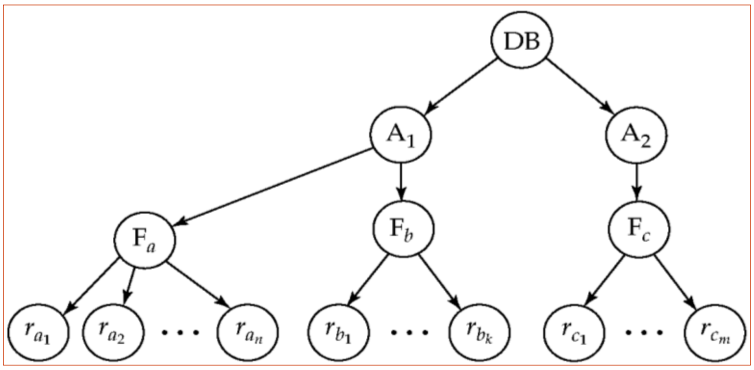
- If for all *Ti* with TS (*Ti*) < TS (*Tj*) either one of the following condition holds:
  - **finish**(*Ti*) < **start**(*Tj*)
  - **start**(*Tj*) < **finish**(*Ti*) < **validation**(*Tj*) **and** the set of data items written by *Ti* does not intersect with the set of data items read by *Tj*.
- Then validation succeeds and *Tj* can be committed. Otherwise, validation fails and *Tj* is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - Writes of *Tj* do not affect reads of *Ti* since they occur after *Ti* has finished its reads.
  - Writes of *Ti* do not affect reads of *Tj* since *Tj* does not read any item written by *Ti.*

Schedule Produced by Validation: Example of schedule produced using validation

T14	T15
read(*B*)	
	read(*B*)
	B:= *B*-50
	read(*A*)
	A:= *A*+50
read(*A*)	
(*validate*)	
display (*A*+*B*)	
	(*validate*)
	write (*B*)
	write (*A*)

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 nodes descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency



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

1. *database*
2. *area*
3. *file*
4. *record*

Intention Lock Modes: In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

1. ***intention-shared*** (IS): Indicates explicit locking at a lower level of the tree but only with shared locks.
2. ***intention-exclusive*** (IX): Indicates explicit locking at a lower level with exclusive or shared locks
3. ***shared and intention-exclusive*** (SIX): The sub tree 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 *Ti* 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 *Ti* in S or IS mode only if the parent of *Q* is currently locked by *Ti* in either IX or IS mode.
4. A node *Q* can be locked by *Ti* in X, SIX, or IX mode only if the parent of *Q* is currently locked by *Ti* in either IX or SIX modes.
5. *Ti* can lock a node only if it has not previously unlocked any node (that is, *Ti* is two-phase).
6. *Ti* can unlock a node *Q* only if none of the children of *Q* are currently locked by *Ti.*

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

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 <*Q1, Q2,...., Qm*>. Each version *Qk* contains three data fields:
  - **Content** -- the value of version *Qk.*
  - **W-timestamp**(*Qk*) -- timestamp of the transaction that created (wrote) version Qk
  - **R-timestamp**(*Qk*) -- largest timestamp of a transaction that successfully read version Qk
- When a transaction *Ti* creates a new version *Qk* of *Q*, Qk's W-timestamp and R-timestamp are initialized to TS(*Ti*).
- R-timestamp of *Qk* is updated whenever a transaction *Tj* reads *Qk*, and TS(*Tj*) > R-timestamp(*Qk*).
- Suppose that transaction *Ti* issues a **read**(*Q*) or **write**(*Q*) operation. Let *Qk* denote the version of *Q* whose write timestamp is the largest write timestamp less than or equal to TS(*T*i).
  - If transaction *Ti* issues a **read**(*Q*), then the value returned is the content of version Qk.
  - If transaction *Ti* issues a **write**(*Q*)
    - if TS(*Ti*) < R-timestamp(*Qk*), then transaction *Ti* is rolled back.
    - if TS(*T*i) = W-timestamp(*Qk*), the contents of *Qk* are overwritten
    - else a new version of *Q* is created.

- Observe that
    - Reads always succeed
    - A write by *Ti* is rejected if some other transaction *Tj* that (in the serialization order defined by the timestamp values) should read *Ti*'s write, has already read a version created by a transaction older than *Ti*.
- ✓ *This 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 current value of **ts-counter** before they start execution; they follow Multiversion timestamp-ordering protocol to perform reads.
- 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 *Ti* completes, commit processing occurs:
    - *Ti* sets timestamp on the versions it has created to **ts-counter** + 1
    - *Ti* increments **ts-counter** by 1
- Read-only transactions that start after *Ti* increments **ts-counter** will see values updated by *Ti*.
- Read-only transactions that start before *Ti* increments the **ts-counter** will see the value before the updates by *Ti.*
- *Only serializable schedules are produced.*

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
    - Ex: if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > Q9, then Q5 will never be required again

Deadlock Handling

Consider the following two transactions:

*T*1:	write (*X*)	*T*2:	write(*Y*)
	write(*Y*)		write(*X*)

Schedule with deadlock is given as:

T1	T2
lock-X on *X*	
write (*X*)	
	lock-X on *Y*
	write (*X*)
	wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

- 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 to unlock the data item.

Deadlock Prevention: Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (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).

Deadlock Prevention Strategies: Following schemes use transaction timestamps for the sake of deadlock prevention alone.

1. **Wait-die scheme** (non-preemptive): Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item
2. **Wound-wait scheme** (preemptive) : Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than *wait-die* scheme.

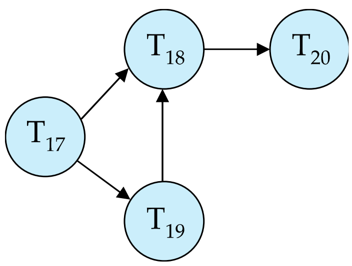
Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

3. **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. It is simple to implement; but starvation is possible. It is also difficult to determine good value of the timeout interval.

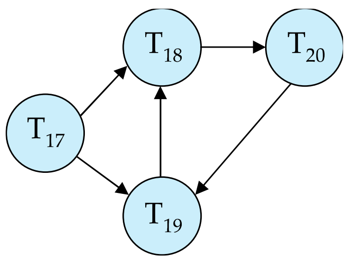
Deadlock Detection: Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$.

- V is a set of vertices (all the transactions in the system)
- E is a set of edges; each element is an ordered pair $Ti \rightarrow Tj$.

- If $Ti \rightarrow Tj$ is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- When Ti requests a data item currently being held by Tj, then the edge Ti Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- 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.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery: When a deadlock is detected:

- Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
- Rollback -- determine how far to roll back transaction
  - Total rollback: Abort the transaction and then restart it.
  - More effective to roll back transaction only as far as necessary to break deadlock.
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

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 database is given an X-mode lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation (Ex: find sum of balances of all accounts in Perryridge) and a transaction that inserts a tuple in the relation (Ex: insert a new account at Perryridge) conflict (conceptually) in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - Ex: the scan transaction does not see the new account, but reads some other tuple written by the update transaction
- 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 information should be locked.
- 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.
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

Index Locking Protocol: According to this protocol 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 *Ti* that performs a lookup must lock all index leaf nodes that it accesses, in S-mode
  - Even if the leaf node does not contain any tuple satisfying the index lookup
- A transaction *Ti* that inserts, updates or deletes a tuple *ti* in a relation *r*
  - must update all indices to *r*
  - must obtain exclusive 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

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, Ex: 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.
    - 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.
- ✓ 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 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; such as 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

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 and has to be explicitly changed to serializable when required as:

- **set isolation level serializable**