LOCK MANAGEMENT

Lock manager — The part of the DBMS that keeps track of the locks issued to transactions

Lock table — A hash table with the data object identifier as the key

Transaction table — A descriptive entry for each transaction, and among other things, the entry contains a pointer to a list of locks held by the transaction.

A lock table entry for an object :

the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode)

the nature of the lock (shared or exclusive)

a pointer to a queue of lock requests

Implementing Lock and Unlock Requests

If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object

If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry

Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended

Notes:

If *TI* has a shared lock on O and T2 requests an exclusive lock, *T2's* request is queued

If *T3* requests a shared lock, its request enters the queue behind that of *T2,* even though the requested lock is compatible with the lock held by TI

This rule ensures that *T2* does not ***starve,*** that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent *T2* from getting the exclusive lock for which it is waiting

Deadlocks

Transaction *T*1 gets an exclusive lock on object *A*

*T*2 gets an exclusive lock on *B*

*T*1 requests an exclusive lock on *B* and is queued

*T*2 requests an exclusive lock on *A* and is queued

Now, *T*1 is waiting for *T*2 to release its lock and *T*2 is waiting for *T*1 to release its lock!

Deadlock

Deadlock Prevention

Giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (vice versa)

Give each transaction a **timestamp** when it starts up

If a transaction T_i requests a lock and transaction T_j holds a conflicting lock, the lock manager can use one of the following two policies:

Wait-die: If T_i has higher priority, it is allowed to wait; otherwise it is aborted

Wound-wait: If T_i has higher priority, abort T_j; otherwise T_i waits

Deadlock Detection

DBMS must periodically check for deadlocks

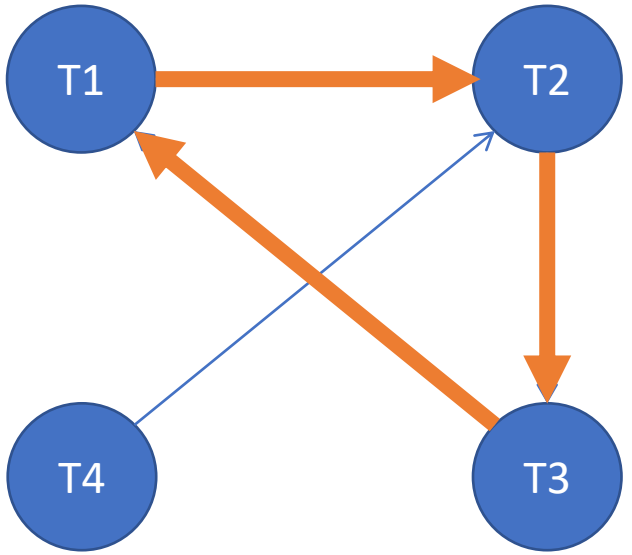
The lock manager maintains a **waits-for graph** to detect deadlock cycles.

The nodes correspond to active transactions,

The arc from T_i to T_j if (and only if) T_i is waiting for T_j to release a lock

Example :

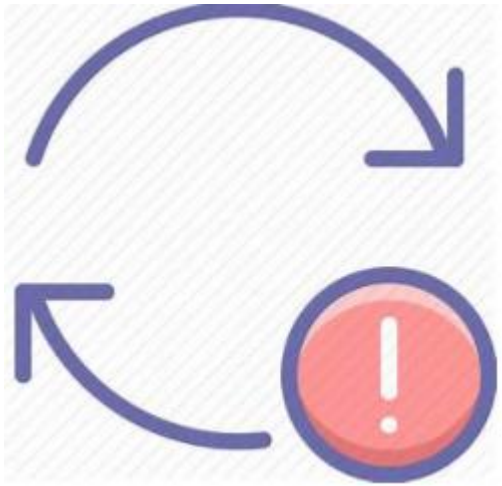
$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	



Conservative 2PL

Conservative 2PL *prevents* deadlocks.

The difference between 2PL and C2PL is that C2PL's transactions obtain all the locks they need before the transactions begin.
This is to ensure that a transaction that already holds some locks will not block waiting for other locks.



The Phantom Problem

The collection of database objects that is not fixed, but can grow and shrink through the insertion and deletion of objects

```
T1: SELECT MAX(IPK)
    FROM Nilai_Mhs
    WHERE Level = 1    → hasil = 3.4
    SELECT MAX(IPK)
    FROM Nilai_Mhs
    WHERE Level = 2    → hasil = 3.7
T2: INSERT INTO Nilai_Mhs (Level, IPK)
    VALUES (1, 3.9)
    DELETE FROM Nilai_Mhs
    WHERE Level = 2 AND IPK = Max(IPK)
    COMMIT
```

```
T1: SELECT MAX(IPK)
    FROM Nilai_Mhs
    WHERE Level = 1 (3,4)
T2: INSERT INTO Nilai_Mhs (Level, IPK)
    VALUES (1, 3.9)
    DELETE FROM Nilai_Mhs
    WHERE Level = 2 AND IPK = Max(IPK)
    COMMIT                          (3,7)
T1: SELECT MAX(IPK)
    FROM Nilai_Mhs
    WHERE Level = 2 (3,3)
```

Index Locking

If there is no index, and all pages in the file must be scanned, T1 must somehow ensure that no new pages are added to the file, in addition to locking all existing pages.

If there is a dense index1 on the level field, T1 can obtain a lock on the index page, that contains a data entry with level = 1.

If there are no such data entries, that is, no records with this level value, the page that would contain a data entry for level=1 is locked, in order to prevent such a record from being inserted.

Any transaction that tries to insert a record with level=1 into the Students relation must insert a data entry pointing to the new record into this index page and is blocked until T1 releases its locks.

Concurrency Control in B+ Trees

The higher levels of the tree only serve to direct searches, and all the `real' data is in the leaf levels

Searches: obtain shared locks on nodes, starting at the root and proceeding along a path to the desired leaf

A lock on a node can be released as soon as a lock on a child node is obtained, because searches never go back up

Search

Concurrency Control in B+ Trees

A node must be locked (in exclusive mode) only if a split can propagate up to it from the modified leaf

Obtain exclusive locks on all nodes from the root to the leaf node to be modified, because splits can propagate all the way from a leaf to the root.

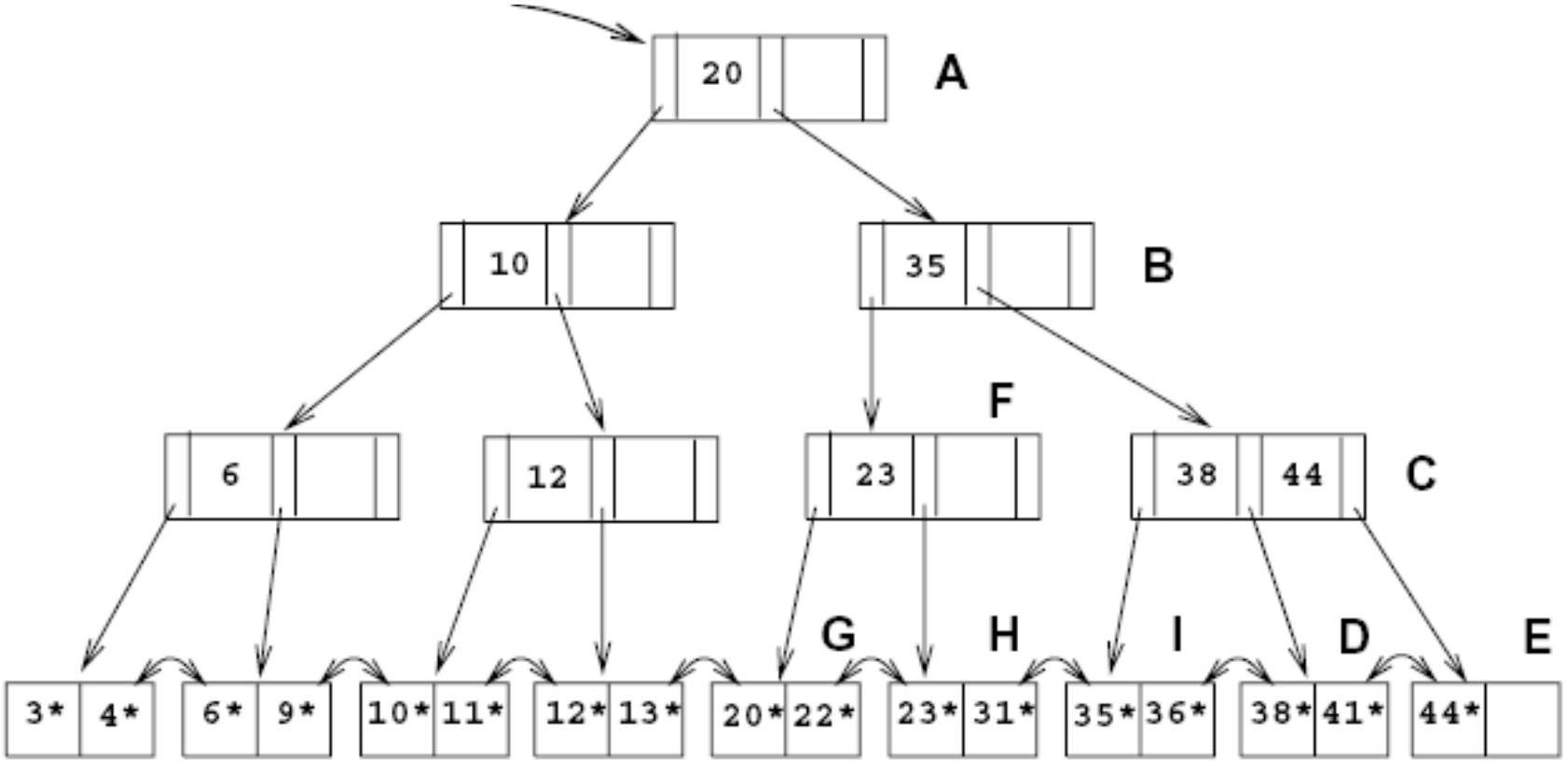
Once the child of a node is locked, the lock on the node is required only in the event that a split propagates back up to it.

If the child of this node (on the path to the modified leaf) is not full when it is locked, any split that propagates up to the child can be resolved at the child, and will not propagate further to the current node. → the lock on the parent can be released if the child is not full.

Insert

The locks held by an insert force any other transaction following the same path to wait at the earliest point (i.e., the node nearest the root) that might be affected by the insert

B+ Tree Locking Example:
search 38*,delete 38* insert 45*, insert 25*



Search 38*

- obtain an *S* lock on node *A,* read the contents and determine that it needs to examine node *B,*
- obtain an *S* lock on node *B* and release the lock on *A,*
- obtain an *S* lock on node C and release the lock on *B,*
- obtain an *S* lock on node D and release the lock on C.

Delete 38*

- If transaction Tj wants to delete 38*, also traverse the path from the root to node *D* and is forced to wait until Ti is done.
- If some transaction *Tk* holds a lock on node C before *Ti* reaches this node, *Ti* is similarly forced to wait for *Tk* to complete.

Insert 45*

- To insert data entry 45*, a transaction must obtain an *S* lock on node *A,*
- Obtain an S lock on node *B* and release the lock on *A,*
- obtain an *S* lock on node C (observe that the lock on *B* is *not* released, because C is full),
- Obtain an X lock on node *E* and release the locks on C and then *B.*
- Because node *E* has space for the new entry, the insert is accomplished by modifying this node.

Insert 25*

- Proceeding as for the insert of 45*, obtain an X lock on node *H.*
- Unfortunately, node H is full and must be split. Splitting *H* requires that we also rnodify the parent, node *F,*
- the transaction has only an *S* lock on *F.* Thus, it must request an upgrade of this lock to an X lock.
- If no other transaction holds an *S* lock on *F,* the upgrade is granted, and since *F* has space, the split does not propagate further and the insertion of 25* can proceed

End of File