CONCURRENCY CONTROL TECHNIQUES

CHAPTER 10

After reading this chapter, the reader will understand:

- *The need of concurrency control techniques*
- *The basic concept of locking, types of locks and their implementation*
- *Lock based techniques for concurrency control*
- *Two-phase locking and its different variants*
- *Specialized locking techniques, such as multiple-granularity locking, tree structured indexes, etc.*
- *Factors affecting the performance of locking*
- *Timestamp-based techniques for concurrency control*
- *Variants of timestamp ordering techniques*
- *Optimistic techniques for concurrency control*
- *Multiversion technique based on timestamp ordering*
- *Multiversion technique based on locking*
- *Handling deadlock*
- *Different deadlock prevention techniques*
- *Deadlock detection and recovery*

Transactions execute either serially or concurrently. If all the transactions are restricted to execute serially, isolation property of transaction is maintained and the database remains in a consistent state. However, executing transactions serially unnecessarily reduce the resource utilization. On the other hand, execution of several transactions concurrently may result in interleaved operations and isolation property of transaction may no longer be preserved. Thus, it may leave the database in an inconsistent state. Consider a situation in which two transactions concurrently access the same data item. One transaction modifies a tuple, and another transaction makes a decision on the basis

of that modification. Now, suppose that the first transaction rolls back. At this point, the decision of second transaction becomes invalid. Thus, **concurrency control techniques** are required to control the interaction among concurrent transactions. These techniques ensure that the concurrent transactions maintain the integrity of a database by avoiding the interference among them.

This chapter discusses the concurrency control techniques, which ensure serializability order in the schedule. These techniques are locking, timestamp-based, optimistic, and the multiversion.

10.1 LOCKING

Whenever a data item is being accessed by a transaction, it must not be modified by any other transaction. In order to ensure this, a transaction needs to acquire a *lock* on the required data items. A **lock** is a variable associated with each data item that indicates whether a read or write operation can be applied to the data item. In addition, it synchronizes the concurrent access of the data item. Acquiring the lock by modifying its value is called **locking**. It controls the concurrent access and manipulation of the locked data item by other transactions and hence, maintains the consistency and integrity of the database. Database systems mainly use two modes of locking, namely, *exclusive locks* and *shared locks*.

Exclusive lock (denoted by x) is the commonly used locking strategy that provides a transaction an exclusive control on the data item. A transaction that wants to read as well as write a data item must acquire exclusive lock on the data item. Hence, an exclusive lock is also known as an *update* lock or a *write* lock. If a transaction T_i has acquired an exclusive lock on a data item, say Q, then T_i can both read and write Q. Further, if another transaction, say T_j, requests to access Q, then T_j has to wait for T_i to release its lock on Q. It means that no transaction is allowed to access data item when it is exclusively locked by another transaction. By prohibiting other transactions to modify the locked data item,

exclusive lock prevents concurrent transactions from corrupting one another.

Shared lock (denoted by $\mathtt{s}$) can be acquired on a data item when a transaction wants to only read a data item and not modify it. Hence, it is also known as **read** lock. If a transaction T_i has acquired a shared lock on data item $\mathtt{Q}$, T_i can read but cannot write on $\mathtt{Q}$. Moreover, any number of transactions can acquire shared locks on the same data item simultaneously without any risk of interference between transactions. However, if a transaction has already acquired a shared lock on the data item, no other transaction can acquire an exclusive lock on that data item.

10.1.1 Lock Compatibility

Depending on the type of operations a transaction needs to perform, it should request a lock in an appropriate mode on that data item. If the requested data item is not locked by another transaction, the lock request is granted immediately by the concurrency control manager. However, if the requested data item is already locked, the lock request may or may not be granted depending on the *compatibility* of the locks. Lock compatibility determines whether locks can be acquired on a data item by multiple transactions at the same time.

Suppose a transaction T_i requests a lock of mode $\mathtt{m}_1$ on a data item $\mathtt{Q}$ on which another transaction T_j currently holds a lock of mode $\mathtt{m}_2$. If mode $\mathtt{m}_2$ is compatible with mode $\mathtt{m}_1$, the request is immediately granted, otherwise rejected. The lock compatibility can be represented by a matrix called the **compatibility matrix** (see Figure 10.1). The term "YES" indicates that the request can be granted and "NO" indicates that the request cannot be granted.

Requested mode	Shared	Exclusive
Shared	YES	NO
Exclusive	NO	NO

Fig. 10.1 *Compatibility matrix*

If the mode m_1 is shared, the lock request of transaction T_j is granted immediately, if and only if m_2 is also shared. Otherwise the lock request is not granted and the transaction T_j has to wait. On the other hand, if mode m_1 is exclusive, the lock request (either shared or exclusive) by transaction T_j is not granted and T_j has to wait.

Hence, it is clear that multiple shared-mode locks can be acquired on a data item by different transactions simultaneously. However, an exclusive-mode lock cannot be acquired on a data item until all other locks on that data item have been released.

Whenever a lock on a data item is released, the *lock status* of the data item changes. If the exclusive-mode lock on the data item is released by the transaction, the lock status of that data item becomes unlocked. Now, the lock request of the waiting transaction is considered. If more than one transactions are waiting, the lock request of one of them is granted.

If the shared-mode lock is released by the transaction holding the lock on the data item, the lock status of that data item may not always be unlocked. This is due to the reason that more than one shared-mode lock may be held on the data item. The lock status of the data item becomes unlock only when the transaction releasing the lock is the only transaction holding the shared-mode lock on that data item. In order to count the number of transactions holding a shared lock on a data item, the number of transaction is stored in an appropriate data structure along with the data item. The number is increased by one when another transaction is granted a shared lock and is decreased by one when a transaction releases the shared lock. Note that when this number becomes zero, the lock status of the data item becomes unlocked.

A data item `Q` can be locked in the shared mode by executing the statement `lock-S(Q)`. Similarly, `Q` can be locked in the exclusive mode by executing the statement `lock-X(Q)`. A lock on the data item `Q` can be released by executing the statement `unlock(Q)`. If a transaction already

holds a lock on any data item, we assume that it will not request a lock on that data item again. In addition, if a transaction does not hold a lock on a data item, we assume that it will not unlock a data item.

10.1.2 Implementation of Locking

A lock can be considered as a control block, which has the information about the nature of the locked data item and the identity of the transaction that acquires the lock. The locking or unlocking of the data items is implemented by a subsystem of the database system known as **lock manager**. It receives the lock requests from transactions and replies them with lock grant message or rollback message (in case of deadlock). In response to an unlock request, the lock manager only replies with an acknowledgement. In addition, it may also result in lock grant messages to other waiting transactions.

To select a transaction from the list of waiting transactions, the lock manager has to follow some priority technique. Otherwise, it may result in **starvation** of the transactions waiting for an exclusive lock. To understand the concept, consider a situation in which a transaction, say T_i, has a shared lock on a data item Q and another transaction, say T_j, requests an exclusive lock on the data item Q. Clearly, T_j has to wait until the transaction T_i releases the shared lock on Q. Further, suppose in the meanwhile, another transaction T_k requests a shared lock on Q. Since the mode of lock request is compatible with the lock held by T_i, the lock request is granted to T_k. Similarly, there may be a sequence of transactions requesting shared lock on Q. In that case, T_j never gets exclusive lock on Q, and is said to be **starved**, that is, waits indefinitely. T_j does not starve if the request of T_k is queued behind that of T_j, even if the mode of lock request is compatible with the lock held by T_i.

Learn More

Both the lock and unlock request must be implemented as atomic operations in order to prevent any type of inconsistency. For this, multiple instances of the lock manager code may run simultaneously to receive

the requests; however, the access to lock table requires some synchronization mechanism such as semaphore.

To guarantee freedom from starvation problem, the lock manager maintains a linked list of records for each locked data item in the order in which the requests arrive. Each record of the linked list is used to keep the transaction identifier that made the request and the mode in which the lock is requested. It also records whether the request has been granted. Lock manager uses a hash table known as **lock table**, indexed on the data item identifier, to find the linked list for a data item.

Consider a lock table, shown in Figure 10.2, which contains locks for three different data items, namely, Q_1, Q_2, and Q_3. In addition, two transactions, namely, T_i and T_j, are shown which have been granted locks or are waiting for locks.

Observe that T_i has been granted locks on Q_1 and Q_3 in shared mode. Similarly, T_j has been granted lock on Q_2 and Q_3 in shared mode, and is waiting to acquire a lock on Q_1 in exclusive mode, which has already been locked by T_i.

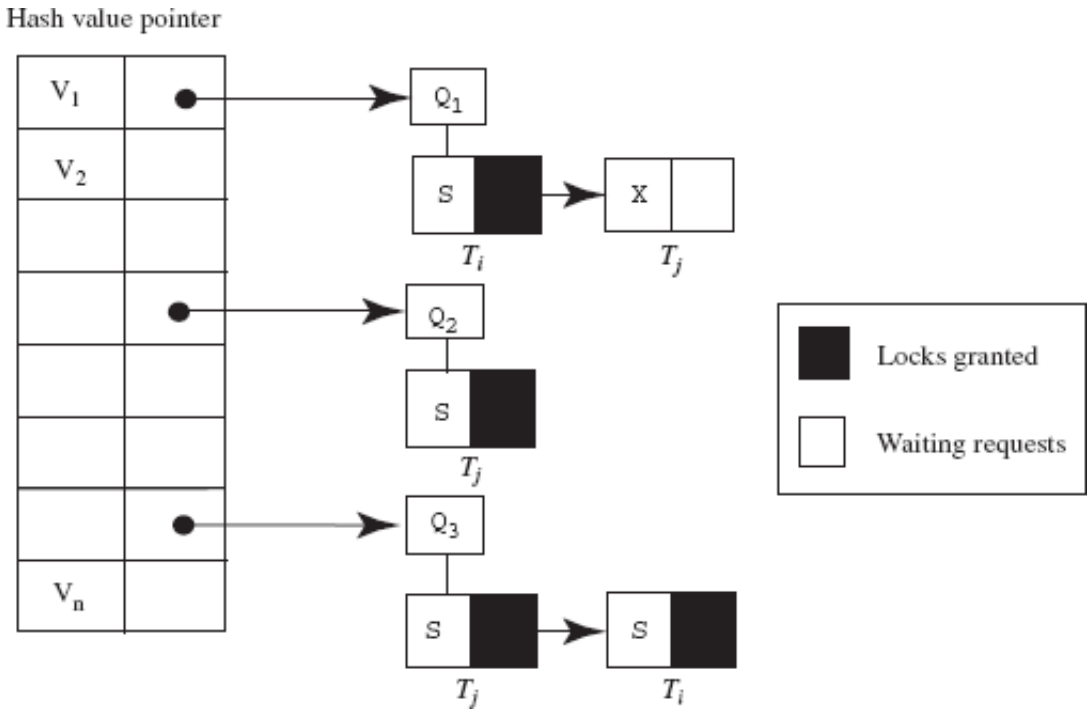


Fig. 10.2 *Lock table*

The lock manager processes the request by the transaction to lock and unlock a data item in the way given here.

- **Lock Request:** When first request to lock a data item arrives, the lock manager creates a new linked list to record the lock request for the data item. In addition, it immediately grants the lock request of the transaction.

  However, if the linked list for the data item already exists, it includes the request at the end of the linked list. Note that the lock request will be granted only if the lock request is compatible with all the existing locks and no other transaction is waiting for acquiring lock on this data item. Otherwise, the transaction has to wait.

- **Unlock Request:** When an unlock request for a data item arrives, the lock manager deletes the record corresponding to that transaction from the linked list for that data item. It then checks whether other waiting requests on that data item can be granted. If the request can be granted, it is granted by the lock manager, and the next record, if any, is processed.

If a transaction aborts, the lock manager deletes all waiting lock requests by the transaction. In addition, the lock manager releases all locks acquired by the transaction and updates the records in the lock table.

10.2 LOCK-BASED TECHNIQUES

A transaction does not release a lock on the data item as long as it uses that data item. It may release the lock immediately after the final accessing of the data item is done. However, releasing the data item immediately is not always desirable. As an illustration, consider transactions T_1 and T_2, and data items Q and R with the initial value of 500 units and 1000 units, respectively. T_1 wants to deduct 200 units from the data item R and add 200 units to the data item Q. Whereas T_2 wants to add the values of Q and R. The transactions T_1 and T_2 with lock requests are given in Figure 10.3.

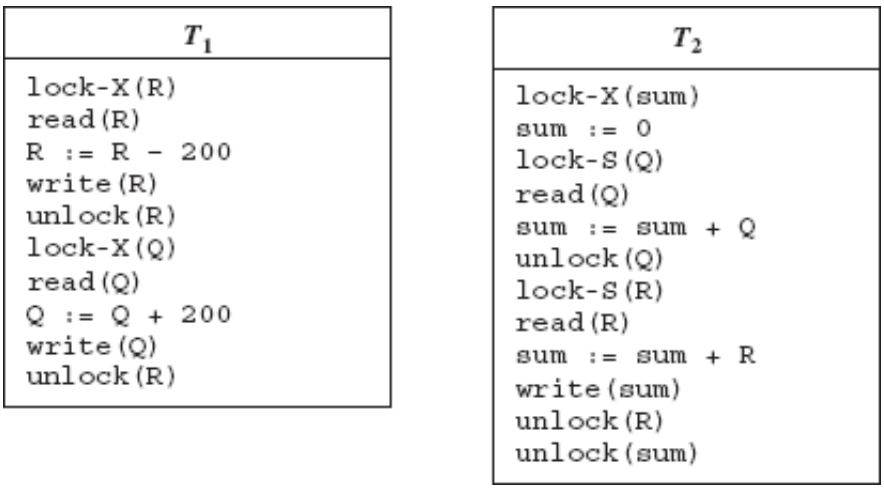


Fig. 10.3 *Transactions T_1 and T_2 with lock requests*

If these transactions are executed serially, either T_1 followed by T_2 or vice-versa, T_2 will display the sum 1500. Alternatively, T_1 and T_2 may be executed concurrently. A possible schedule of concurrent execution of T_1 and T_2 is shown in Figure 10.4. This schedule shows the statements issued by the transactions in the interleaved manner. Note that the lock must be granted after the transaction requests the lock but before the transaction executes its next statement. The lock can be granted anywhere in between this interval; however, we are not interested in the exact point of time where the lock is granted. For simplicity, we assume that the lock is granted immediately before the execution of next statement by the transaction.

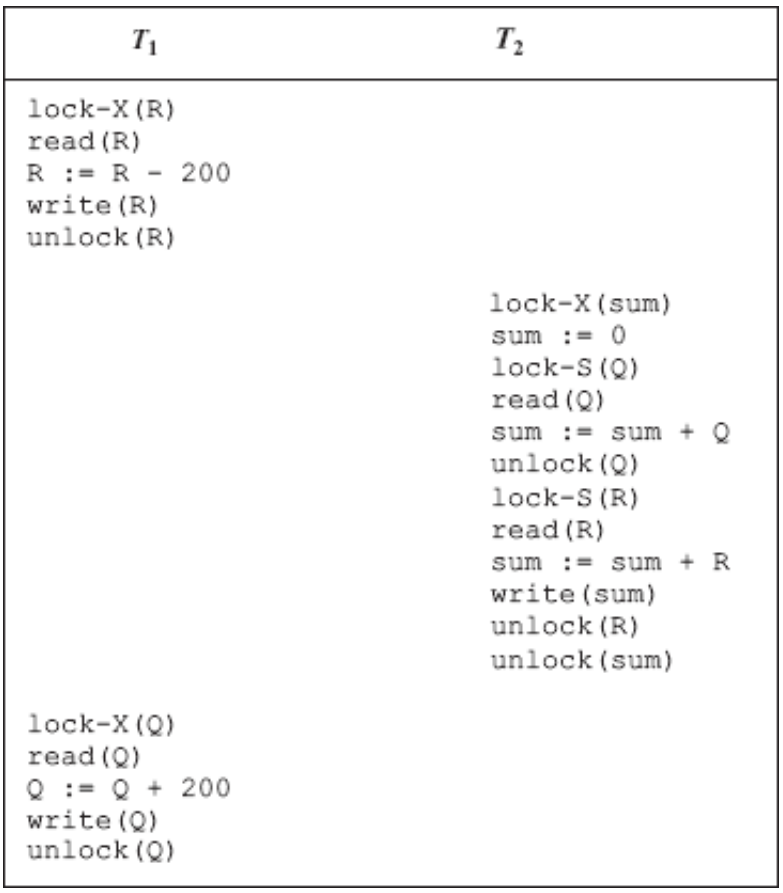


Fig. 10.4 *Schedule of T_1 and T_2*

In Figure 10.4, the concurrent execution of the transactions results in displaying the sum 1300 units instead of 1500. Observe that the database is in an inconsistent state since 200 units are deducted from data item R but not added to Q. In addition, T_2 is allowed to read the values of Q and R before transaction T_1 is complete. This is because the locks on the data items Q and R are released as soon as possible. Now, suppose that the unlock statements in Figure 10.4 are delayed to the end of the transactions. Then, T_1 unlocks Q and R only after the completion of its actions. So, the database remains in consistent state.

However, sometimes delaying the lock until the end of transaction may lead to an undesirable situation, called **deadlock**. Deadlock is a situation that occurs when all the transactions in a set of two or more transactions are in a simultaneous wait state and each of them is waiting for the release of a data item held by one of the other waiting transaction in the set. None of the transactions can proceed until at least one of the waiting transactions releases lock on the data item. For example, consider the

partial schedule of transactions T_3 and T_4 shown in Figure 10.5.

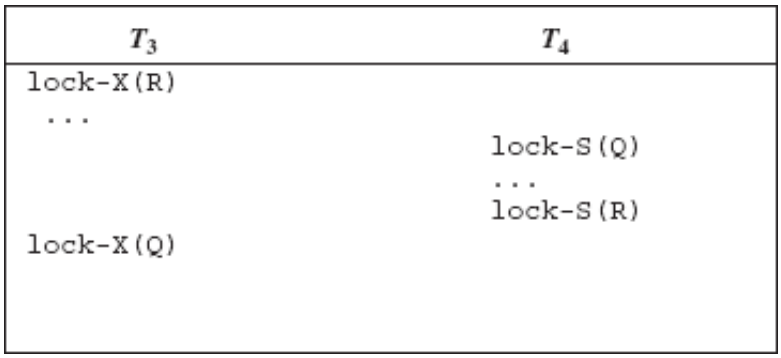


Fig. 10.5 *Partial schedule*

Observe that T_3 is waiting for T_4 to unlock Q, and T_4 is waiting for T_3 to unlock R. Thus, a situation is arrived where these two transactions can no longer continue with their normal execution. This situation is called deadlock. Now, one of these transactions must be rolled back by the system so that the data items locked by that transaction are released and become available to the other transaction.

From this discussion, it is clear that if locking is not used or if data items are unlocked too early, a database may become inconsistent. On the other hand, if the locks on data items are not released until the end of transaction, deadlock may occur. Out of these two problems, deadlocks are more desirable, since they can be handled by the database system but inconsistent state cannot be handled. Deadlock handling is discussed in detail in Section 10.8.

There is a need that all the transactions in a schedule must follow some set of rules called **locking technique**. These rules indicate when a transaction may lock or unlock any data item. We discuss several locking techniques in subsequent sections.

10.2.1 Two-Phase Locking

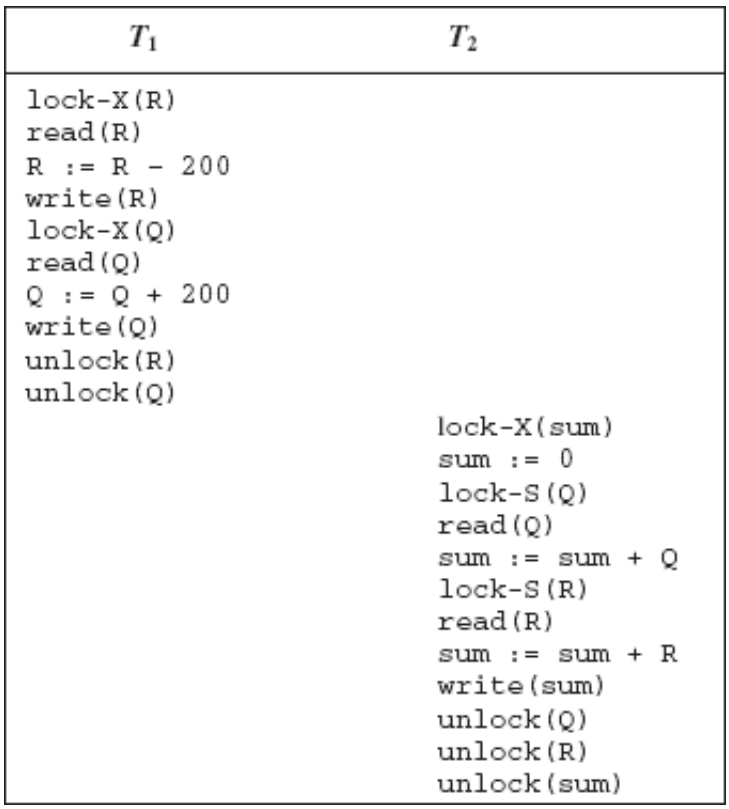
Two-phase locking requires that each transaction be divided into two phases. During the first phase, the transaction acquires all the locks; during the second phase, it releases all the locks. The phase during which locks are acquired is *growing* (or *expanding*) *phase*. In this phase,

the number of locks held by a transaction increases from zero to maximum. On the other hand, a phase during which locks are released is *shrinking* (or *contracting*) *phase*. In this phase, the number of locks held by a transaction decreases from maximum to zero.

Whenever, a transaction releases a lock on a data item, it enters into the shrinking phase, and from this point, it is not allowed to acquire any lock further. So, until all the required locks on the data items are acquired, the release of the locks must be delayed. Thus, the two-phase locking leads to lower degree of concurrency among transactions. This is because a transaction cannot release any data item (which is no longer required), if it needs to acquire locks on additional data items later on. Alternatively, it must acquire locks on additional data items before it actually performs certain action on those data items, in order to release other data items. Meanwhile, another transaction that needs to access any of these locked data items is forced to wait.

The point in the schedule at which the transaction successfully acquires its last lock is called the **lock point** of the transaction.

For example, the transactions T_1 and T_2 (see Figure 10.3) can be rewritten under two-phase locking as shown in Figure 10.6.



T_1	T_2
lock-X(R) read(R) R := R - 200 write(R) lock-X(Q) read(Q) Q := Q + 200 write(Q) unlock(R) unlock(Q)	
	lock-X(sum) sum := 0 lock-S(Q) read(Q) sum := sum + Q lock-S(R) read(R) sum := sum + R write(sum) unlock(Q) unlock(R) unlock(sum)

Fig. 10.6 *Transactions T_1 and T_2 in two-phase locking*

In Figure 10.6, the statements used for releasing the lock are written at the end of the transaction. However, such statements do not always need to appear at the end of the transaction to retain two-phase locking property. For example, the `unlock(R)` statement of T_1 may appear just after the `lock-X(Q)` statement and still maintains the two-phase locking property.

The two-phase locking produces the serializable schedules. To verify this, consider a schedule s for a set of transactions T_i, T_j, ..., T_k under two-phase locking. Now assume that s is non-serializable, thus, there must exist a cycle $T_i \rightarrow T_j \rightarrow T_m ... \rightarrow T_n \rightarrow T_i$ in the precedence graph for s. Here, $T_i \rightarrow T_j$ indicates that T_j acquires a lock on a data item released by T_i. Similarly, T_m acquires lock on a data item released by T_j, and finally T_i acquires a lock on a data item released by T_n. It means T_i acquires a lock on a data item after releasing a lock, and this is a contradiction that T_i is following two-phase locking. Thus, our assumption that s is non-serializable is wrong.

Although, serializability is ensured by two-phase locking, cascading

rollback and deadlock is possible under two-phase locking. To avoid cascading rollback, a modification of two-phase locking called **strict two-phase locking** can be used. In the strict two-phase locking, a transaction does not release any of its exclusive-mode locks until it commits or aborts. If all the transactions $\{T_1, T_2, ..., T_n\}$ participating in a schedule s follow strict two-phase locking, the schedule s is said to be **strict schedule**. Strict schedules ensure that no other transaction can read or write the data item exclusively locked by a transaction T until it commits. It makes strict schedules recoverable. However, strict two-phase locking can also lead to deadlock.

Another more restrictive variation of two-phase locking is known as **rigorous two-phase locking**. In rigorous two-phase locking, a transaction does not release any of its locks (both exclusive and shared) until it commits or aborts.

Lock Conversion

When **lock conversions** are allowed, the transactions can change the mode of locks from one mode to another on a data item on which it already holds a lock. A transaction can request a lock on a data item many times but it can hold only one lock on the data item at any time. Lock conversion helps in achieving more concurrency.

To understand the need of lock conversion, suppose that a transaction T_i has locked a data item Q_1 in shared mode. Further, consider a transaction T_j given in Figure 10.7.

Learn More

To achieve more concurrency, a new lock mode referred to as **update** mode can be introduced. This lock mode is compatible with shared mode but not with other lock modes. Initially, each transaction locks the data item in update mode. Later, it can upgrade the lock mode to exclusive if it needs to modify the data item, or downgrade to shared if it does not.

Suppose the transaction T_j follows two-phase locking, then it needs the data items Q_1, Q_2, and Q_3 to be locked in exclusive mode. Clearly, T_j has to wait, since T_i has locked the data item Q_1 in shared mode.

However, one can easily observe that T_j needs exclusive lock on Q_1 only when it updates Q_1, that is, at the end of its execution. Till then, it just needs a shared lock on Q_1. Thus, if T_j initially acquires shared lock on Q_1, it may start processing concurrently with T_i. Later, when T_i has done with Q_1 and released its shared lock, then, T_j can easily acquire exclusive lock on Q_1 to update it. This changing of lock mode from shared (less restrictive) to exclusive (more restrictive) is known as **upgrading**. Similarly, changing the lock mode from exclusive to shared is known as **downgrading**. Therefore, to achieve more concurrency, two-phase locking is extended to allow conversion of locks.

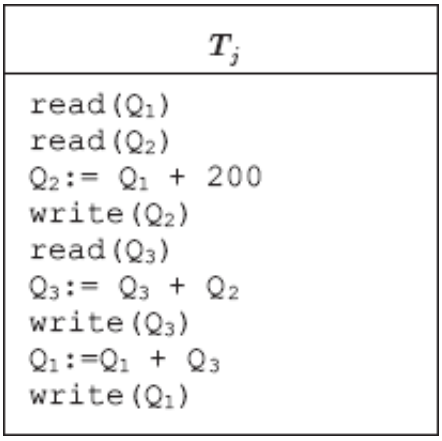


Fig. 10.7 *Transaction T_j*

NOTE The lock on a data item can be upgraded in only growing phase. On the other hand, the lock on a data item can be downgraded in only shrinking phase.

A transaction that already holds a lock of mode m_1 on Q can upgrade it to a more restrictive mode m_2, if no other transaction holds a lock conflicting with mode m_2, otherwise it has to wait. On the other hand, a transaction that already holds a lock of mode m_1 on Q can downgrade it to a less restrictive mode m_2 on Q at any time.

10.2.2 Graph-Based Locking

We have discussed two-phase locking, which ensures serializability even when the information about the order in which the data items are accessed is unknown. However, if we know in advance the order in which the data items are accessed, it is possible to develop other techniques to ensure conflict serializability. One such technique is **graph-based locking**.

In this technique, a directed acyclic graph called a **database graph** is formed, which consists of a set `S = {A,B,…,L}` of all data items as its nodes, and a set of directed edges. A directed edge from `A` to `B (A → B)` in the database graph denotes that any transaction T_i must access `A` before `B` when it needs to access both `A` and `B`. For simplicity, we discuss a simple kind of graph-based locking called the **tree-locking**. In this all database graphs are tree-structured, and any transaction T_i can acquire only exclusive locks on data items. A tree-structured database graph for the set `S` is shown in Figure 10.8.

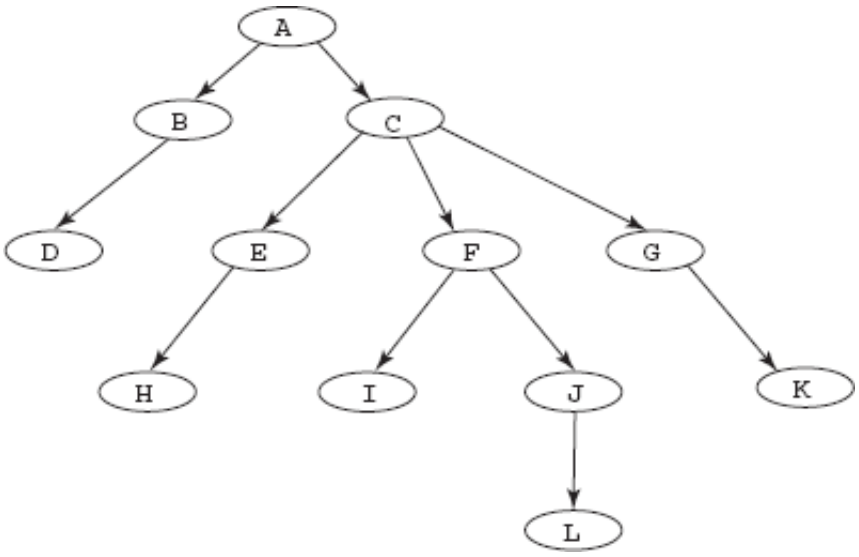


Fig. 10.8 *Tree-structured database graph*

A transaction T_i can acquire its first lock on any data item. Suppose T_i requests a lock on data item `F`, it is granted. After that, T_i can lock a data item only if it has already locked the parent of that data item. For example, if transaction T_i needs to access `L`, it has to lock `J` before

requesting lock on L. Note that locking the parent of any data item does not automatically lock that data item.

Tree locking allows a transaction to unlock a data item at any time. However, once a transaction releases lock on a data item, it cannot relock that data item. Tree-locking not only ensures conflict serializability but also ensures freedom from deadlock. However, recoverability and cascadelessness are not ensured.

To ensure recoverability and cascadelessness, the tree-locking technique can be modified. The modified technique does not allow any transaction to release its exclusive locks until it commits or aborts. Clearly, it reduces the concurrency. Alternatively, we have a way that helps in improving concurrency but ensures only recoverability. In this, for each data item, we record the uncommitted transaction T_i that issued the last write instruction to that data item. Further, whenever T_j reads any such data item, we ensure that T_j cannot commit before T_i. To ensure this, we record the commit dependency of T_j on all the transactions that issued the last write instruction to the data items that T_j reads. If any one of them rolls back, T_j must also be rolled back.

The advantages of the tree-locking are given here.

- It ensures freedom from deadlock.
- Unlike two-phase locking, transactions can unlock data items earlier.
- It ensures shorter waiting times and greater amount of concurrency.

The disadvantage of the tree-locking is that in order to access a data item, a transaction has to lock other data items even if it does not need to access them. For example, in order to lock data items A and L in Figure 10.8, a transaction must lock not only A and L, but also data items C, F, and J. Thus, the number of locks and associated locking overhead including possibility of additional waiting time is increased.

10.3 SPECIALIZED LOCKING TECHNIQUES

A static view of a database has been considered for locking as discussed so far. In reality, a database is dynamic since the size of database changes over time. To deal with the dynamic nature of database, we need some specialized locking techniques, which are discussed in this section.

10.3.1 Handling the Phantom Problem

Due to the dynamic nature of database, the **phantom problem** may arise. Consider the BOOK relation of *Online Book* database that stores information about books including their price. Now, suppose that the PUBLISHER relation is modified to store information about average price of books that are published by corresponding publishers in the attribute Avg_price. Consider a transaction T_1 that verifies whether the average price of books in PUBLISHER relation for the publisher *P001* is consistent with the information about the individual books recorded in BOOK relation that are published by *P001*. T_1 first locks all the tuples of books that are published by *P001* in BOOK relation and thereafter locks the tuple in PUBLISHER relation referring to *P001*. Meanwhile, another transaction T_2 inserts a new tuple for a book published by *P001* into BOOK relation, and then, before T_1 locks the tuple in PUBLISHER relation referring to *P001,* T_2 locks this tuple and updates it with the new value. In this case, average information of T_1 will be inconsistent even though both transactions follow two-phase locking, since new book tuple is not taken into account. The new book tuple inserted into BOOK relation by T_2 is called a **phantom tuple**. This is because T_1 assumes that the relation it has locked includes all information of books published by *P001*, and this assumption is violated when T_2 inserted the new book tuple into BOOK relation.

Here, the problem is that T_1 did not lock what it logically required to lock. Instead of locking existing tuples of publisher *P001*, it also needed to restrict the insertion of new tuples having P_ID = "P001". This can be done by using a general technique known as **predicate locking**. In this technique, all the tuples (whether existing or new tuples that are to be inserted) that satisfy an *arbitrary predicate* are locked. In our example, since the attribute that refers to the publisher who has published the

book is `P_ID`, the predicate which locks all the books of *P001* publisher is `P_ID = "P001"`. However, the predicate locking is an expensive technique, so most of the systems do not support predicate locking. These systems yet manage to prevent the phantoms problem by using another technique, called **index locking**. In this technique, any transaction that tries to insert a tuple with predicate `P_ID = "P001"` must insert a data entry pointing to the new tuple into the index and is blocked until T_i releases the lock. These indexes must be locked in order to eliminate the phantom problem. Here, we take only B^+-tree index to maintain simplicity.

In order to access a relation, one or more indexes are used. In our example, assume that we have an index on `BOOK` relation for `P_ID`. Then, the entry in `P_ID` will be locked for *P001*. In this way, the creation of phantoms will be prevented since the creation requires the index to be updated and it also requires an exclusive lock to be acquired on that index.

10.3.2 Concurrency Control in Tree-Structured Indexes

Like other database objects, indexes also need to be accessed concurrently by the transactions. Further, the changes in index structure between two accesses by a transaction are acceptable as long as the index structure directs the transaction to the correct set of tuples.

We can apply two-phase locking to tree-structured indexes like B^+-trees; however, it results in low degree of concurrency. This is because searching an index always starts at the root, and if any transaction locks the root node, then all other transactions with conflicting lock requests have to wait. Thus, we require other techniques for managing concurrent access to indexes. In this section, we present two techniques for concurrency control in tree-structured indexes.

The first technique is **crabbing**, which proceeds in the similar manner as a crab walks, that is, releasing lock on a parent node and acquiring lock on a child node and so on alternately. To understand this technique, consider a transaction T_i which wants to insert a tuple in a relation on

which a B$^+$-tree index exists. Clearly, T_i also needs to insert a key-value (or corresponding entry) in the appropriate node. For this, the transaction T_i proceeds as follows:

- T_i first locks the root node of tree in shared mode. Then, it acquires a shared lock on the child node, which is to be traversed next, and releases the lock on the parent node. This process is repeated till we traverse down to the appropriate leaf node.
- T_i then upgrades its shared lock on leaf node to exclusive lock, and inserts the key-value there, assuming that the leaf node has space for the new entry. Otherwise, the node needs to be split.
- In case of split, T_i acquires exclusive-mode lock on its parent, performs the splitting operation (as explained in Chapter 07), and releases its lock on the parent node.

Note that in case of split, T_i needs to acquire exclusive locks on the nodes while moving up the tree. Thus, there is a possibility of deadlock of T_i with any other transaction T_j that is trying to traverse down the tree. However, such deadlocks can be handled by restarting T_j.

An alternative technique uses a variant of the B$^+$-tree called **B-link tree**. In B-link tree, every node of the tree (not just leaf nodes) maintains a pointer that points to its right sibling. This technique differs from crabbing as it allows the transactions to release locks on nodes even before acquiring locks on the child nodes. Thus, allows more transactions to operate concurrently.

Now, consider a transaction T_i that inserts a key-value in a leaf node and a split occurs. The split causes the redistribution of key-values in the sibling nodes as well as in the parent node. Suppose at that point of time, another transaction T_j follows the same path for searching a key-value. Since, the key-values of the node are already redistributed with the right siblings by T_i, T_j may not find the required key-value in the desired leaf node. However, T_j can still complete its searching process by following the pointer to the right sibling. This is the reason for the nodes at every level of the tree to have pointers to right siblings. Note that the search

and insertion operation cannot result in deadlock.

Performing the deletion operation is very simple one. The transaction traverses down the tree to search the required leaf node and removes the key-value from the leaf node. If there are few key-values left in the node after the deletion operation, it needs to be coalesced (or combined). In case of coalescing, the sibling nodes must be locked in exclusive mode. After coalescing on the same level, an exclusive lock is requested on the parent node to remove the deleted node. At this point, the locks on the nodes that are combined are released. Note that if there are few key-values left in the parent node, it can also be coalesced with its siblings. Once the transaction has combined the two parent nodes, its lock is released.

In case of coalescing of nodes during deletion operation, a concurrent search operation may find a pointer to a deleted node, if the parent node is not updated. In such situations, search operation has to be restarted. To avoid such inconsistencies, nodes can be left uncoalesced. If this is done, the B^+-tree will contain the nodes with few values, which violates its properties.

10.3.3 Multiple-Granularity Locking

Before discussing multiple-granularity locking, one must first understand the concept of locking granularity. **Locking granularity** is the size of the data item that the lock protects. It is important for the performance of a database system. **Coarse granularity** refers to large data item sizes such as an entire relation or a database, whereas **fine granularity** refers to a small data item sizes such as either a tuple or an attribute. If the larger data item is locked, it is easier for the lock manager to manage the lock. The overhead associated with locking the large data item is low since there are fewer locks to manage. In this case, the degree of concurrency is low because the transaction is holding a lock on a large data item, even if it is accessing only a small portion of the data item. Observe that the transaction completes its operations successfully, but at the expense of forcing many transactions to wait. On the other hand, the overhead

associated with locking the small data item is considerably high since there are many locks to be managed. However, the degree of concurrency in this case is high, since any number of transactions can acquire any number of locks, which will be managed by the lock manager.

A transaction requires lock granularity on the basis of the operation being performed. An attempt to modify a particular tuple requires only that tuple to be locked. At the same time, an attempt to modify or delete multiple tuples requires an entire relation to be locked. Since different transactions have dissimilar requests, it is desirable that the database system provides a range of locking granules called **multiple-granularity locking**. The efficiency of the locking mechanism can be improved by considering the lock granularity to be applied. However, the overheads associated with multiple-granularity locking can outweigh the performance gains of the transactions.

Multiple-granularity locking permits each transaction to use levels of locking that are most suitable for its operations. This implies that long transactions use coarse granularity and short transactions use fine granularity. In this way, long transactions can save time by using few locks on a large data item and short transactions do not block the large data item, when its requirement can be satisfied by the small data item.

Note that the size of the data items to be locked influences the level of concurrency. While choosing locking granularity, trade-off between concurrency and overheads should be considered. In other words, the choice is between the loss of concurrency due to coarse granularity and increased overheads of maintaining fine granularity. This choice depends upon the type of the applications being executed and how those applications utilize the database. Since the choice depends upon the type of the applications, it is appropriate for the database system to support multiple-granularity locking.

As shown in Figure 10.9, the hierarchy of data items of various sizes can be represented in the form of a tree in which small data items are nested within larger data items. The hierarchy in this figure consists of database

DB with three files, namely, F_1, F_2, and F_3. Each file consists of several records as its child nodes. Here, notice the difference between the multiple-granularity tree and tree-locking. In the multiple-granularity tree, each non-leaf node represents data associated with its descendents whereas each node in tree-locking is an independent data item.

Learn More

To predict an appropriate locking granularity for a given transaction, a technique called **lock escalation** may be used. In this technique, initially, a transaction starts locking items of fine granularity. But when the transaction has exceeded a certain number of locks at that granularity, it starts obtaining locks at the next higher level of granularity.

Whenever a transaction acquires shared-mode or exclusive-mode lock on a node in multiple-granularity tree, the descendants of that node are implicitly locked in the same mode. Consider the scenario given here to clear this point: Suppose T_i acquires an exclusive lock on file F_1 in Figure 10.9. In that case, it has an exclusive lock on all the records, that is, R_{11}, …, R_{1n} of that file. Observe that, in this way, T_i has to acquire only a single lock on F_1 instead of acquiring locks on individual records of F_1. Now, assume that T_j requests a shared lock on R_{11} of F_1. Now, T_j must traverse from the root of the tree to record R_{11} to determine whether this request can be granted. If any node in that path is locked in an incompatible mode, the lock request for R_{11} is denied and T_j must wait. Further, assume that another transaction T_k wants to lock the entire database. This can be done by locking the root of the tree. However, this request should not be granted since T_i is holding a lock on file F_1. In order to determine whether the root node can be locked, the tree has to be traversed to examine every node to see whether any of them is currently locked by any other transaction. This would be undesirable and would defeat the purpose of multiple-granularity locking technique.

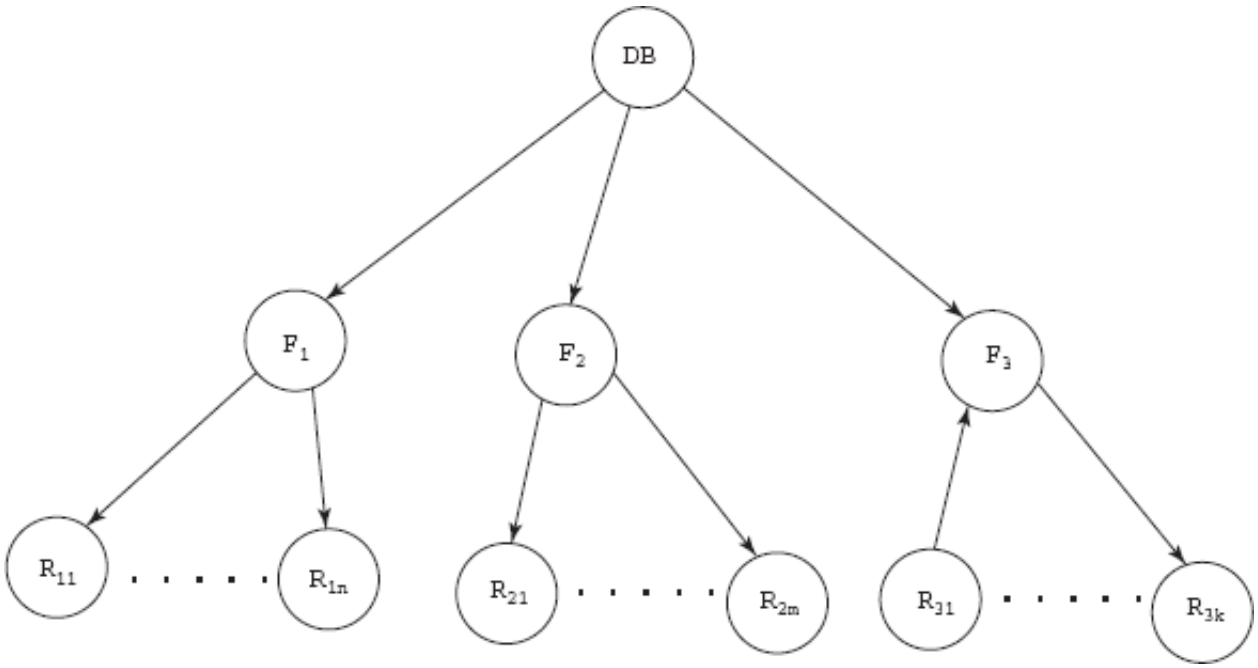


Fig. 10.9 *Multiple-granularity tree*

Instead, we introduce another type of lock mode, called **intention lock**, in which a transaction intends to explicitly lock a lower level of the tree. While traversing from the root node to the desired node, all the nodes along the path are locked in intention-mode. In simpler terms, no transaction is allowed to acquire a lock on a node before acquiring an intention-mode lock on all its ancestor nodes. For example, to lock a node R_{11} in Figure 10.9, a transaction needs to lock all the nodes along the path from root node to node R_{11} in an intention-mode before acquiring lock on node R_{11}.

To provide higher degree of concurrency, the intention-mode is associated with shared-mode and exclusive-mode. When intention-mode is associated with shared-mode, it is called **intention-shared (IS) mode**, and when it is associated with exclusive-mode, it is called **intention-exclusive (IX) mode**. To lock a node in shared mode, all its ancestor nodes must be locked in intention-shared (IS) mode. Similarly, to lock a node in exclusive mode, all its ancestor nodes must be locked in intention-exclusive (IS) mode.

NOTE Intention-shared (IS) lock is incompatible only with exclusive lock whereas intention-exclusive (IX) lock is incompatible with both shared

and exclusive locks.

It is clear from the discussion that the lower level of the tree has to be explicitly locked in the mode requested by the transaction. However, it is undesirable if a transaction needs to access only a small portion of the tree. Thus, another type of lock mode, called **shared and intention-exclusive (SIX) mode** is introduced. The shared and intention-exclusive mode explicitly locks the sub tree rooted by a node in the shared mode, and the lower level of that sub tree is explicitly locked in exclusive-mode. This mode provides the higher degree of concurrency than exclusive mode because it allows other transactions to access the part of the sub tree that is not locked in the exclusive mode.

Relative Privilege of Locking Modes

Now, consider Figure 10.10, which shows the relative privilege of the locking modes. Among all the locking modes, the highest privilege is given to the exclusive mode. This is because it does not allow other transactions to acquire any lock on the portion of the tree, which is rooted at the node locked in exclusive mode. In addition, the lower level of the sub tree, rooted by that node, is implicitly locked in the exclusive mode. On the other hand, the lowest privilege is given to the intention-shared mode. The shared mode and the intention-exclusive mode are given same privilege.

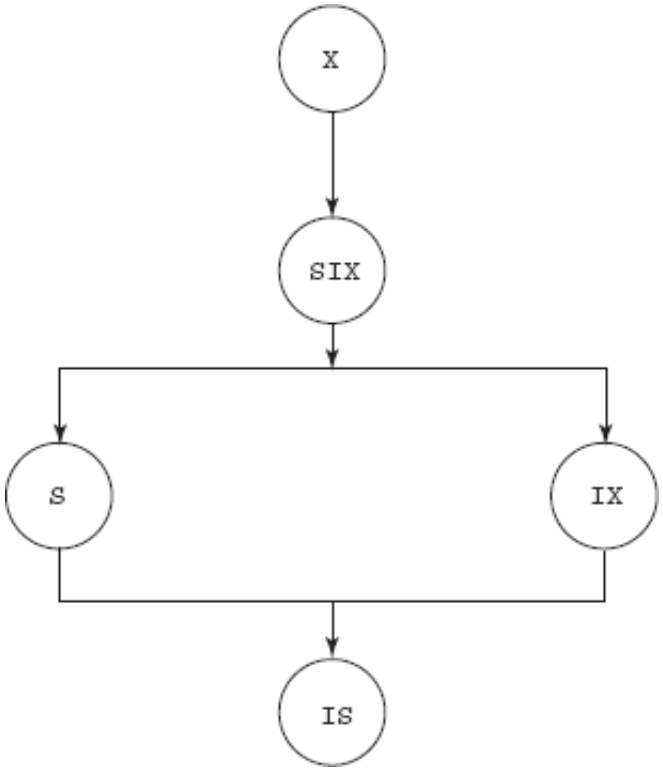


Fig. 10.10 *Relative privilege of the locking modes*

Consider two lock modes, namely, m_1 and m_2, such that m_1 is given higher privilege than m_2. Now, if a lock request of mode m_2 is denied on a data item, then a lock request of mode m_1 on the same data item will also be denied definitely. It implies that if there is "NO" in column of m_2 in the compatibility matrix (see Figure 10.11) for a particular lock request, there will be "NO" in the column of m_1.

	Current lock on the node				
Requested mode	X	SIX	IX	S	IS
X	NO	NO	NO	NO	NO
SIX	NO	NO	NO	NO	YES
IX	NO	NO	YES	NO	YES
S	NO	NO	NO	YES	YES
IS	NO	YES	YES	YES	YES

Fig. 10.11 *Compatibility matrix for different access modes*

Locking Rules

Each transaction T_i can acquire a lock on node N in any locking mode (see Figure 10.11) by following certain rules, which ensure serializability. These rules are given here.

- T_i must adhere to the compatibility matrix given in Figure 10.11.
- T_i must acquire any lock on the root of the tree before acquiring any lock on N.
- T_i can acquire S or IS lock on N only if it has successfully acquired either IX or IS lock on the parent of N.
- T_i can acquire X, SIX, or IX lock on N only if it has successfully acquired either IX or SIX lock on the parent of N.
- T_i can acquire additional locks if it has not released any lock. Observe that this is requirement of two-phase locking.
- T_i must release the locks in bottom-up order (that is, leaf-to-root). Therefore, T_i can release its lock on N only if it has released all its locks on the lower level of the sub tree rooted by N.

10.4 PERFORMANCE OF LOCKING

Normally, two factors govern the performance of locking, namely, *resource contention* and *data contention*. **Resource contention** refers to the contention over memory space, computing time and other resources. It determines the rate at which a transaction executes between its lock requests. On the other hand, **data contention** refers to the contention over data. It determines the number of currently executing transactions.

Now, assume that the concurrency control is turned off; in that case the transactions suffer from resource contention. For high loads, the system may thrash, that is, the throughput of the system first increases and then decreases. Initially, the throughput increases since only few transactions request the resources. Later, with the increase in the number of transactions, the throughput decreases. If the system has enough resources (memory space, computing power, etc.) that make the contention over resources negligible, the transactions only suffer from data contention. For high loads, the system may thrash due to *aborting* (or rollback) and *blocking*. Both the mechanisms degrade the performance.

Clearly, aborting the transaction degrades the performance as the work

done so far by the transaction is wasted. However, performance degradation due to blocking is more subtle as it prevents other transactions to access the data item which is held by blocked transaction. An instance of blocking is deadlock in which two more transactions are in a simultaneous wait state, and are waiting for some data item, which is locked by one of the other transactions. Hence, the transactions are blocked till the time one of the deadlocked transactions is aborted. Practically, it is seen that there are less than 1% of transactions, which are involved in a deadlock and there are comparatively lesser aborts. Thus, the system thrashes mainly due to blocking.

Consider how system thrashes due to blocking. Initially, the first few transactions are unlikely to block each other; as a result, the throughput increases. Transactions begin to block each other when more and more transactions execute concurrently. Thus, chances of occurrence of blocking increase with the increase in the number of active transaction. However, the throughput does not increase with the increase in the number of transactions. In fact, there comes a point when including additional transaction decreases the throughput. This is because additional transaction is blocked and competes with other transactions to acquire locks on the data item, which decreases the throughput. At this point, the system thrashes, which is shown in Figure 10.12.

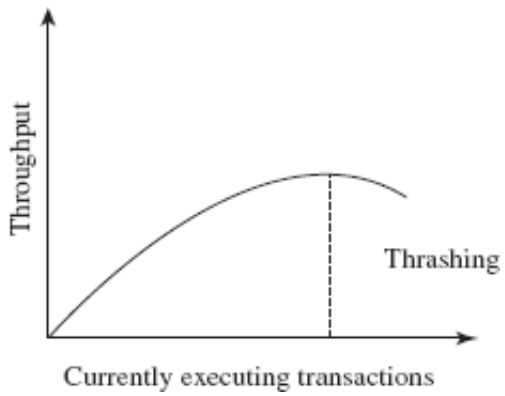


Fig. 10.12 *Thrashing*

In order to handle this situation, the database administrator should reduce the number of concurrently executing transactions. It is seen that

thrashing occurs when 30% of the currently executing transactions are blocked. So, the database administrator should use this fraction of blocked transaction to check whether the system thrashes or not. In addition, the database administrator can increase the throughput in following ways:

- By locking the data item with smallest granularity.
- By reducing the time for which the transaction can hold lock on a data item.
- By reducing **hot spots** which are frequently accessed data items that remain in the buffer for long period of time.

10.5 TIMESTAMP-BASED TECHNIQUE

So far, we have discussed that the locks with the two-phase locking ensures the serializability of schedules. Two-phase locking generates the serializable schedules based on the order in which the transactions acquire the locks on the data items. A transaction requesting a lock on a locked data item may be forced to wait till the data item is unlocked. Serializability of the schedules can also be ensured by another method, which involves ordering the execution of the transactions in advance using timestamps.

Timestamp-based concurrency control is a non-lock concurrency control technique, hence, deadlocks cannot occur.

10.5.1 Timestamps

Timestamp is a unique identifier assigned to transactions in the order they begin. If transaction T_i begins before transaction T_j, T_i is assigned a lower timestamp. The priority of a transaction will be higher if it is assigned lower timestamp. More precisely, the older transaction has the higher priority since it is assigned a lower timestamp. The timestamp of a transaction T_i is denoted by $\mathrm{TS}(\mathrm{T_i})$. Timestamp can be considered as *starting time* of the transaction and it is assigned to each transaction T_i in the order in which the transactions are entered in the database system.

Suppose a new transaction T_j enters in the system after T_i. In that case, the timestamp of T_i is less than that of T_j, which can be denoted as $TS(T_i) < TS(T_j)$.

The timestamp of each transaction can be generated in two simple ways:

- The value of *system clock* can be used as the timestamp. When a transaction enters the system, it must be assigned a timestamp that is equal to the value (that is, current date/time) of the system clock. It must be ensured that no two transactions are assigned the timestamp values at the same tick of the system clock.
- A *logical counter* can be used which is incremented each time a transaction is assigned a timestamp. When a transaction enters the system, it must be assigned a value that is equal to the value of the counter. The value of the counter may be numbered as 1, 2, 3, …. A counter has finite number of values, so the value of the counter must be reset periodically to zero when no transaction is currently executing for some short period of time.

10.5.2 The Timestamp Ordering

In the timestamp ordering, the transactions are ordered on the basis of their timestamps. It means when two concurrently executing transactions, namely, T_i and T_j are assigned the timestamp values $TS(T_i)$ and $TS(T_j)$, respectively, such that $TS(T_i)<TS(T_j)$, the system generates a schedule equivalent to a serial schedule in which the older transaction T_i appears before the younger transaction T_j. This is termed as **timestamp ordering (TO)**.

Things to Remember

Although the timestamp ordering is a non-locking technique in the sense that it does not lock a data item from concurrent access for the duration of a transaction. But in actual, at the time of recording timestamps against the data item, it requires a lock on the data item or on its instance for a small duration.

When the timestamp ordering is enforced, the order in which the data item is accessed by conflicting operations in the schedule does not violate the serializability order. In order to determine this, the following two timestamps values must be associated with each data item `Q`.

- **`read_TS(Q)`:** The **read timestamp** of `Q`; this is the largest timestamp among all the timestamps of transactions that have successfully executed `read(Q)`.
- **`write_TS(Q)`:** The **write timestamp** of `Q`; this is the largest timestamp among all the timestamps of transactions that have successfully executed `write(Q)`.

NOTE Whenever new `read(Q)` or `write(Q)` operations are executed, `read_TS(Q)` and `write_TS(Q)` are updated.

A number of implementation techniques based on the timestamp ordering have been proposed for concurrency control techniques, which ensure that all the conflicting operations of the transactions are executed in timestamp order. Three of them, namely, *basic timestamp ordering, strict timestamp ordering*, and *Thomas' write rule* are discussed here.

Basic Timestamp Ordering

The **basic timestamp ordering** ensures that the transaction T_i is executed in the timestamp order whenever T_i requests read or write operation on `Q`. This is done by comparing the timestamp of T_i with `read_TS(Q)` and `write_TS(Q)`. If the timestamp order is violated, the system rolls back the transaction T_i and restarts it with a new timestamp. In this situation, a transaction T_j, which may have used a value of the data item written by T_i, must be rolled back. Similarly, any other transaction T_k, which may have used a value of the data item written by T_j, must also be rolled back, and so on. This situation is known as **cascading rollback**, which is one of the problems with basic timestamp ordering.

Basic timestamp ordering operates as follows:

1. Transaction T_i requests a read operation on Q:
    1. If $TS(T_i) < write\_TS(Q)$, the read operation is rejected and T_i is rolled back. This is because another transaction with higher timestamp (that is, younger transaction) has already written the value of Q, which T_i needs to read. The transaction T_i is too late in performing the required read operation and any other values it has acquired are expected to be inconsistent with the modified value of Q. Thus, it is better to rollback T_i and restart it with a new timestamp.
    2. If $TS(T_i) \geq write\_TS(Q)$, the read operation is executed, and $read\_TS(Q)$ is set to the maximum of $read\_TS(Q)$ and $TS(T_i)$. This is because T_i is younger than the transaction that has written the value of Q, which T_i needs to read. If the timestamp of T_i is more than the current value of read timestamp (that is, $read\_TS(Q)$), the timestamp of T_i becomes the new value of $read\_TS(Q)$.
2. Transaction T_i requests a write operation on Q:
    1. If $TS(T_i) < read\_TS(Q)$, the write operation is rejected and T_i is rolled back. This is because another transaction with higher timestamp (that is, younger transaction) has already read the value of Q, which T_i needs to write. So, writing the value of Q by T_i violates the timestamp ordering. Thus, it is better to rollback T_i and restart it with a new timestamp.
    2. If $TS(T_i) < write\_TS(Q)$, the write operation is rejected and T_i is rolled back. This is because another transaction with higher timestamp (that is, younger transaction) has already written the value of Q, which T_i needs to write. So, the value that T_i is attempting to write becomes obsolete. Thus, it is better to rollback T_i and restart it with a new timestamp.
    3. If $TS(T_i) \geq read\_TS(Q)$ and $TS(T_i) \geq write\_TS(Q)$, the write operation is executed, and the value of $TS\ (T_i)$ becomes the new value of $write\_TS(Q)$. This is because T_i is younger than both the transactions that have last written the value of Q and

read the value of Q.

Consider two transactions, namely, T_5 and T_6 such that $TS(T_5) < TS(T_6)$. One possible schedule for T_5 and T_6 is shown in Figure 10.13. For maintaining simplicity, only the read and write operations of the transactions are shown in this schedule.

When T_5 issues read(R) operation, it is observed that $TS(T_5) < write\_TS(R)$. This is due to the reason that $write\_TS(R) = TS(T_6)$, hence, the read operation of T_5 is rejected and T_5 is rolled back. This rollback is required in basic timestamp ordering but it is not always necessary.

Fig. 10.13 *Schedule for T_5 and T_6*

Basic timestamp ordering executes the conflict operations in timestamp order; hence, it ensures conflict serializability. In addition, it is deadlock-free since no transaction ever waits. However, cyclic restart of a transaction may occur in basic timestamp ordering, if it is repeatedly rolled back and restarted. This cyclic restart of the transaction leads to the starvation of the transaction. If the restarting of a transaction occurs several times, conflicting transactions need to be temporarily blocked which allows the transaction to finish.

Strict Timestamp Ordering

The **strict timestamp ordering** is a variation of basic timestamp ordering. In addition to the basic timestamp ordering constraints, it follows another constraint when the transaction T_i requests read or write operations on some data item. This constraint ensures a strict schedule, which guarantees recoverability in addition to serializability. Suppose a transaction T_i requests a read or write operation on Q and $TS(T_i) > write\_TS(Q)$, T_i is delayed until the transaction, say T_j, that wrote the value of Q has committed or aborted.

The strict timestamp ordering is implemented by locking Q that has been written by transaction T_j until it is either committed or aborted.

Furthermore, the strict timestamp ordering ensures freedom from deadlock, since T_i waits for T_j only if $TS(T_i) > TS(T_j)$.

Thomas' Write Rule

Thomas' write rule is the modification to the basic timestamp ordering, in which the rules for write operations are slightly different from those of basic timestamp ordering. The rules for a transaction T_i that request a write operation on data item Q are given here.

1. If $TS(T_i) < read\_TS(Q)$, the write operation is rejected and T_i is rolled back. This is because another transaction with higher timestamp (that is, younger transaction) has already read the value of Q, which T_i needs to write. So, the value of Q that T_i is producing was previously needed, and it had been assumed that the value would never be produced.
2. If $TS(T_i) < write\_TS(Q)$, the write operation can be ignored. This is because another transaction with higher timestamp (that is, younger transaction) has immediately overwritten the value of Q, which T_i wrote. So, no transaction can read the value written by T_i and hence, T_i is trying to write an obsolete value of Q.
3. If both the conditions given in points 1 and 2 do not occur, the write operation of T_i is executed and $write\_TS(Q)$ is set to $TS(T_i)$.

The main difference between these rules and those of basic timestamp ordering is the second rule. It states that if T_i requests a write operation on Q and $TS(T_i) < write\_TS(Q)$, the write operation of T_i is ignored. Whereas, according to basic timestamp ordering, if T_i requests a write operation on Q and $TS(T_i) < write\_TS(Q)$, T_i has to be rolled back.

Fig. 10.14 *A non-conflict serializable schedule*

Unlike other techniques discussed so far, Thomas' write rule does not enforce conflict serializability; however, it makes use of view serializability. It is possible to generate serializable schedules using Thomas' write rule that are not possible under other techniques like two-

phase locking, tree-locking, etc. To illustrate this, consider the schedule shown in Figure 10.14. In this figure, the write operation of T_{10} succeeds the read operation and precedes the write operation of T_9. Hence, the schedule is not conflict serializable.

Under Thomas' write rule, the `write(Q)` operation of T_9 is ignored. Therefore, the schedule shown in Figure 10.15 is view equivalent to the serial schedule of T_9 followed by T_{10}.

Fig. 10.15 *A schedule under Thomas' write rule*

10.6 OPTIMISTIC (OR VALIDATION) TECHNIQUE

All the concurrency control techniques, discussed so far (locking and timestamp ordering) result either in transaction delay or transaction rollback, thereby named as pessimistic techniques. These techniques require performing a check before executing any read or write operation. For instance, in locking, a check is done to determine whether the data item being accessed is locked. On the other hand, in timestamp ordering, a check is done on the timestamp of the transaction against the read and write timestamps of the data item to determine whether the transaction can access the data item. These checks can be expensive and represent overhead during transaction execution as they slow down the transactions. In addition, these checks are unnecessary overhead when a majority of transactions are read-only transactions. This is because the rate of conflicts among these transactions may be low. Therefore, these transactions can be executed without applying checks and still maintaining the consistency of the system by using an alternative technique, known as **optimistic (or validation) technique**.

NOTE *Optimistic technique is named so because the transactions execute optimistically and thereby assumes that few conflicts will occur among the transactions.*

In optimistic concurrency control techniques, it is assumed that the transactions do not directly update the data items in the database until

they finish their execution. Instead, each transaction maintains local copies of the data items it requires and updates them during execution. All the data items in the database are updated at the end of the transaction execution. In this technique, each transaction T_i proceeds through three phases, namely, *read phase, validation phase*, and *write phase*, depending on whether it is a read-only or an update transaction. The execution of transaction T_i begins with read phase, and the three timestamp values are associated with it during its lifetime. The three phases are explained here.

1. **Read phase:** At the start of this phase, transaction T_i is associated with a timestamp $\texttt{Start}(T_i)$. T_i reads the values of data items from the database and these values are then stored in the temporary local copies of the data items kept in the workspace of T_i. All modifications are performed on these temporary local copies of the data items without updating the actual data items of the database.
2. **Validation phase:** At the start of this phase, transaction T_i is associated with a timestamp $\texttt{Validation}(T_i)$. The system performs a validation test when T_i decides to commit. This validation test is performed to determine whether the modifications made to the temporary local copies can be copied to the database. In addition, it determines whether there is a possibility of T_i to conflict with any other concurrently executing transaction. In case any conflict exists, T_i is rolled back, its workspace is cleared and T_i is restarted.
3. **Write phase:** In this phase, the system copies the modifications made by T_i in its workspace to the database only if it succeeds in the validation phase. At the end of this phase, T_i is associated with a timestamp $\texttt{Finish}(T_i)$.

Using the value of the timestamp $\texttt{Validation}(T_i)$, the serializability order of the transactions can be determined by the timestamp ordering technique. Therefore, the value of timestamp $\texttt{Validation}(T_i)$ is chosen as the timestamp of T_i instead of $\texttt{Start}(T_i)$. This is because we expect that it provides faster response time, if conflict rates among transactions are indeed low.

In addition to transaction timestamp, the optimistic technique requires that the `read_set` and `write_set` of the transaction be maintained by the system. The `read_set` of the transaction is the set of data items it reads and the `write_set` of the transaction is the set of data items it writes.

For every pair of transactions T_i and T_j such that $TS(T_i)<TS(T_j)$, the generated schedule must be equivalent to a serial schedule in which T_i appears before T_j. In addition, this pair of transactions must hold one of the following validation conditions.

1. **$Finish(T_i)<Start(T_j)$:** This condition ensures that the older transaction T_i must complete its all three phases before transaction T_j begins its start phase. Thus, this condition maintains serializability order.
2. **$Finish(T_i)<Validation(T_j)$:** This condition ensures that T_i completes its write phase before T_j starts its validation phase. It also ensures that the `write_set` of T_i does not overlap with the `read_set` of T_j. In addition, the older transaction T_i must complete its write phase before younger transaction T_j finishes its read phase and starts its validation phase. This is due to the reason that writes of T_j are not overwritten by writes of T_i. Hence, it maintains the serializability order.
3. **$Validation(T_i) < Validation(T_j)$:** This condition ensures that T_i completes its read phase before T_j completes its read phase. More precisely, this condition ensures that the `write_set` of T_i does not intersect with the `read_set` or `write_set` of T_j. Since T_i completes its read phase before T_j completes its read phase, T_i cannot affect the read or write phase of T_j, which also maintains serializability.

NOTE The validation conditions ensure that modifications made by the younger transaction, say T_j, are not visible to the older transaction say T_i.

To validate T_j, the first condition is checked for each committed transaction T_i such that $TS(T_i) < TS(T_j)$. Only if the first condition does not hold, the second condition is checked. Further, if the second

condition is false, the third condition is checked. If any one of the mentioned validation conditions holds, there is no conflict and T_j is validated successfully. If none of these conditions holds, there is conflict and the validation of T_j fails and it is rolled back and restarted. Observe that the first condition permits T_j to see the changes made to data items by T_i. The second condition permits T_j to *n* read data items when T_i is writing data items. However, since T_j does not read any data item modified by T_i, there is no conflict. The third condition permits both T_i and T_j to write data items at the same time, but the sets of data items written by the two transactions cannot overlap.

Things to Remember

To choose an optimistic or pessimistic concurrency control technique for a transaction depends on the type of transaction. The transactions for which recovery would be risky in case of a failure can be better managed with pessimistic techniques. While the transactions for which it is better to take the risk of failure rather than compromising the efficiency can be better managed with optimistic techniques.

No transaction can be allowed to commit if any transaction is being validated. This is because the transaction, which is being validated, might overlook conflicts with regard to the newly committed transaction. Before validating other transactions, the write phase of a validated transaction must also be completed. So that the changes, if any, made by the validated transaction must be applied to the database.

It can be ensured that at most one transaction is in its validation/write phase at any time by using a synchronization mechanism such as a **critical section**. This mechanism may lead to lower degree of concurrency. So, it is necessary to keep these phases as short as possible. However, if the values of temporary local copies of data items have to be copied to the actual data items of the database, this can make the write phase long. Thus, an alternative approach that uses a level of indirection to shorten the write phase may be followed. This approach uses a logical pointer to access any data item and we simply change the

logical pointer to point to the temporary local copies of data items in the write phase instead of copying the data items.

T_{11}	T_{12}
read(Q)	
	read(Q)
	Q:=Q-200
	read(R)
	R:=R+200
read(R)	
<Validate>	
display(Q+R)	
	<Validate>
	write(Q)
	write(R)
	display(Q+R)

Fig. 10.16 *A schedule under optimistic technique*

For example, consider the schedule given in Figure 10.16. Suppose that the transaction T_{11} starts before T_{12}, such that $TS(T_{11}) < TS(T_{12})$. Since the pair of transactions satisfies the validation conditions given earlier, the validation phase will be successful. Note that the values of temporary local copies of data items are copied to the actual data items of the database only after the validation phase of T_{12}. Furthermore, this schedule maintains serializability order because T_{11} reads the old values of Q and R.

Since the values of temporary local copies of data items are copied to the actual data items of the database only after the validation phase of T_{12}, cascading rollbacks cannot occur. However, starvation of long transactions can be possible due to the conflicts that occur because of short transactions. This conflict results in restarting the long transaction repeatedly. The starvation must be avoided by temporarily blocking the conflicting transactions so that the long transaction can complete its execution.

It is clear that optimistic concurrency control technique has certain overheads like locking technique. Following are the overheads in optimistic technique.

- It maintains `read_set` and `write_set` for each transaction.
- It checks for conflicts among the transactions.
- It copies the values from the workspace of the transaction to the actual database.
- It repeatedly restarts the transactions, which leads to wastage of work they have done so far.

10.7 MULTIVERSION TECHNIQUE

So far, we have discussed that serializability must be maintained when one or more of the transactions need to modify the data item. The serializability can be maintained either by delaying an operation or by aborting the requesting transaction. For example, if the appropriate value of the data item has not been written yet, a read operation may be delayed; or the transaction issuing the read operation must be rolled back in case the value that it was supposed to read has already been overwritten. However, these problems can be avoided by another concurrency control technique, which keeps the old version (or value) of each data item in the system when the data item is updated. This concurrency control technique is known as **multiversion technique**.

In this technique, several versions (or values) of a data item are maintained. When a transaction issues a write operation on the data item Q, it creates a new version of Q, the old version of Q is retained. When a transaction issues a read operation on the data item Q, an appropriate version of Q is chosen by concurrency control techniques in such a way that the serializability of the currently executing transactions be maintained.

The main drawback of the multiversion technique is that more memory space is required to keep multiple versions of the data items. However, old versions of the data items have to be maintained for recovery purposes. Out of several proposed multiversion techniques, this section discusses two multiversion techniques, namely, *multiversion technique based on timestamp ordering* and *multiversion two-phase locking*.

10.7.1 Multiversion Technique Based on Timestamp Ordering

For each version of a data item, say Q_i, system maintains the value of the version and associates the following two timestamps.

- **$read\_TS(Q_i)$:** The **read timestamp** of Q_i; this is the largest timestamp among all the timestamps of transactions that have successfully read Q_i.
- **$write\_TS(Q_i)$:** The **write timestamp** of Q_i; this is the timestamp of the transaction that has written the value of Q_i.

Consider a data item Q with the most recent version Q_i, that is $write\_TS(Q_i)$ is the largest among all the versions. Further, assume a transaction T_j with $write\_TS(Q_i) <= TS(T_i)$. Now when T_j issues $read(Q)$ request, the system returns the value of Q_i and updates the value of $read\_TS(Q_i)$ with $TS(T_i)$ if $read\_TS(Q_i)<TS(T_i)$. On the other hand, when T_j issues a $write(Q)$ request, the following situations may occur.

- $read\_TS(Q_i)>TS(T_i)$, which means any younger transaction has already read the value of Q_i. In this situation, T_j is rolled back.
- $TS(Q_i)=write\_TS(Q_i)$. In this situation, the contents of Q_i are overwritten.
- If none of the above situation holds, a new version, say Q_j, of Q is created with $read\_TS(T_i)=write\_TS(Q_i)=TS(T_i)$.

The main advantage of the multiversion timestamp ordering technique is that read requests by the transaction are never blocked. Hence, it is important for typical database systems in which read requests are more frequent than write requests. On the other hand, there are two main disadvantages of this technique, which are given here.

- Whenever a transaction reads a data item, $read\_TS(Q_i)$ is updated. It results in accessing the disk twice, that is, one for data item and another for updating $read\_TS(Q_i)$.
- Whenever two transactions conflict, one of them is rolled back

(rather than wait) in order to resolve the conflict, which could result in cascading and hence, can be expensive.

The multiversion based on timestamp ordering technique does not ensure recoverability and cascadelessness.

10.7.2 Multiversion Two-Phase Locking

In addition to read and write lock modes, **multiversion two-phase locking** provides another lock mode, that is, **certify**. In order to determine whether these lock modes are compatible with each other or not, consider Figure 10.17.

Fig. 10.17 *Compatibility matrix for multiversion two-phase locking*

The term "YES" indicates that, if a transaction T_i holds the lock on data item Q with the mode specified in column header and another transaction T_j requests to acquire the lock on Q with the mode specified in row header, then the lock can be granted. This is because the requested mode is compatible with the mode of lock held. On the other hand, the term "NO" indicates that the requested mode is not compatible with the mode of lock held, so, the transaction that has requested the lock must wait until the lock is released.

Unlike locking technique, in multiversion two-phase locking, other transactions are allowed to read a data item while a transaction still holds an exclusive lock on the data item. This is done by maintaining two versions for each data item. One version, known as **certified version**, must be written by any committed transaction and second version, known as **uncertified version**, is created when an active transaction acquires an exclusive lock on a data item. The basic idea behind this technique is that transactions can read only the most recently certified version.

Suppose a transaction, T_j, requests a shared lock on a data item Q, on which another transaction T_i holds an exclusive lock. In this situation, T_j is

allowed to read the certified version of Q while T_i is writing the value of uncertified version of Q. However, once T_i is ready to commit, it must acquire a certify lock on Q. Since certify lock is not compatible with other locks (see Figure 10.17), T_i must delay its commit until there is no transaction accessing certified version of the data item in order to obtain the certify lock. Once T_i acquires the certify lock on Q, the value of uncertified version becomes the certified version of Q and the uncertified version is deleted.

This technique has an advantage over two-phase locking that many read operations can execute concurrently with a single write operation on a data item, which is not possible under two-phase locking. This technique; however, has an overhead that the transaction may have to delay its commit until the certify locks are acquired on all the data items it has updated. This technique avoids cascading rollbacks because transactions read certified version instead of uncertified version. Whereas, deadlock may occur if a read lock is allowed to upgrade to a write lock, and they must be handled using the some deadlock handling technique.

10.8 DEALING WITH DEADLOCK

As discussed earlier, **deadlock** is a situation that occurs when all the transactions in a set of two or more transactions are in a simultaneous wait state and each of them is waiting for the release of a data item held by one of the other waiting transaction in the set. None of the transactions can proceed until at least one of the waiting transactions releases lock on the data item.

To get rid of this undesirable situation, system has to rollback some of the transactions involved in the deadlock. Various steps the system can take to deal with deadlock are discussed in this section.

Deadlock Prevention

Deadlock prevention ensures that deadlock never happens. Generally,

this technique is used when the chances of occurring deadlock are high. Following are the approaches to prevent the deadlocks.

- **Conservative 2PL:** In this approach, each transaction locks in advance all the data items that it needs during its life-time.
- **Assigning an order to all the data items:** In this approach, an ordering is imposed on all the data items in the database. Each transaction acquires locks on the data items in a sequence consistent with that order.
- **Using timestamps along with locking:** In this approach, each transaction is assigned a priority using a unique timestamp, which determines whether a transaction is allowed to wait or is rolled back. A lower timestamp denotes higher priority and vice-versa.

The first two approaches can be easily implemented if all the data items that are to be accessed by the transaction are known at the beginning. However, it is not a practical assumption because it is often difficult to predict what data items are needed by a transaction before it begins execution. In addition, both approaches limit concurrency since a transaction locks many data items that remain unused for a long duration. Thus, the third approach is mainly used for deadlock prevention.

To understand the third approach, consider a situation in which a data item Q is locked by a transaction T_i and another transaction T_j issues an incompatible lock request on Q. In this situation, two deadlock prevention techniques using timestamps have been proposed which are discussed here.

- **Wait-die:** If T_i has a lower timestamp (that is, higher priority), T_i is allowed to wait. Otherwise, T_i is rolled back (*dies*).
- **Wound-wait:** If T_i has a lower timestamp (that is, higher priority), T_j is rolled back (T_j is *wounded* by T_i); otherwise, T_i waits.

When a transaction is rolled back and restarted, it retains the same timestamp that it was originally assigned.

Both the wait-die and wound-wait techniques have some similarities, which are discussed here.

- Starvation is avoided by both the wait-die and wound-wait techniques. In both the techniques, a transaction with the smallest timestamp is not rolled back. In addition, the new transactions are given higher timestamp than the older transactions. Thus, a transaction that is rolled back repeatedly will eventually become the oldest transaction and has the highest priority. Observe that the oldest transaction has the smallest timestamp. Therefore, it will not be rolled back again and will be granted all the locks that it had requested.
- The request to acquire a lock on a data item held by another transaction does not necessarily involve a deadlock. Therefore, unnecessary rollbacks may occur in both wait-die and wound-wait techniques.

In spite of these similarities, there are certain important differences between the wait-die and wound-wait techniques, which are given in Table 10.1.

Table 10.1 *Wait-die and wound-wait technique*

Wait-Die Technique	Wound-Wait Technique
In this technique, the waiting that may be required by a transaction with higher priority (that is, older transaction) could be significantly higher.	In this technique, an older transaction gets the greater probability of acquiring a lock on the data item. It never waits for a younger transaction to release the lock on its data item.
In this technique, a transaction may be rolled back many times before acquiring the lock on the requested data item. For example, when a younger transaction T_i requests a data item Q held by an older transaction T_j, then it is rolled back. When it is restarted, it again requests a lock on Q. Now, if Q is still locked by T_j, T_i will be rolled back again. In this	Rollbacks in wound-wait technique are less as compared to wait-die technique. For example, when an older transaction T_i requests a data item Q held by a younger transaction T_j then T_j is rolled back. When it is restarted, it requests Q, which is now locked by T_i. In this situation, T_j waits.

way, T_i may be rolled back several times till it acquires lock on Q.	

Deadlock Detection and Recovery

If there is a little chance of interference among the transactions and the transactions are short and require few locks, we use deadlock detection and recovery techniques instead of deadlock prevention. To detect the deadlock, the system maintains a wait-for graph, which consists of nodes and directed arcs. The nodes of this graph represent the currently executing transactions, and there exists a directed arc from one node to another, if the transaction is waiting for another transaction to release a lock. If there exists a cycle in the wait-for graph, it indicates the deadlock in the system. To understand the creation of a wait-for graph, consider four transactions T_{13}, T_{14}, T_{15}, and T_{16}, whose partial schedule is given in Figure 10.18.

Things to Remember

Invoking the deadlock detection algorithm at small intervals will add considerable overhead and if it is invoked at large intervals, the deadlock will not be detected for a long time. A number of factors, such as the frequency of deadlocks, the number of transactions affected by deadlock, waiting time of transactions, etc., may affect the choice of interval.

T_{13}	T_{14}	T_{15}	T_{16}
lock-S(Q)			
	lock-X(Q_1)		
	lock-S(Q_2)		
		lock-S(Q_4)	
			lock-X(Q_3)
		lock-X(Q_1)	
	lock-X(Q_3)		
lock-X(Q_2)			

Fig. 10.18 *A partial schedule for T_{13}, T_{14}, T_{15}, and T_{16}*

In this schedule, the transaction T_{13} is waiting for transactions T_{14} to release lock on the data item Q_2. Similarly, T_{14} is waiting for T_{16}, and T_{15} is

waiting for T_{14}. For this situation, the wait-for graph is represented in Figure 10.19.

Observe that there exists no cycle in this graph, thus, there is no deadlock. Now assume that the transaction T_{16} requests a lock on the data item Q_4 held by T_{15}, a directed arc is added in the wait-for graph from T_{16} to T_{15} (see Figure 10.20). The creation of this directed arc results in a cycle in the wait-for graph, thus, a deadlock occurs in the system in which the transactions T_{14}, T_{15}, and T_{16} are involved.

Fig. 10.19 *Wait-for graph*

The wait-for graph is periodically examined by the system to check the existence of deadlock. Once the deadlock is detected, there is a need to recover from the deadlock. The simplest solution is to abort some of the transactions involved in the deadlock, in order to allow other transactions to proceed. The transactions chosen to be aborted are called **victim transactions**. Some criteria should be followed to choose victim transaction, like the transaction with fewest locks, the transaction that has done the minimum work, and so on.

Fig. 10.20 *Wait-for graph with cycle*

Another approach for deadlock handling is to specify the time-interval for which a transaction is allowed to wait for acquiring a lock on a data item. If the time-out occurs, the transaction is rolled back and restarted. Thus, in case of deadlock one or more transactions will time-out and roll back automatically, thereby, allowing other transactions to proceed. This approach lies somewhere in between deadlock prevention and deadlock detection and recovery. However, this approach has limited use in practical situations, since there is a chance of occurrence of starvation.

SUMMARY

1. Concurrency control techniques are required to control the interaction among concurrent transactions. These techniques ensure that the concurrent transactions maintain the integrity of a

database by avoiding the interference among them.

2. Whenever a data item is to be accessed by a transaction, it must not be modified by any other transaction. In order to ensure this, a transaction needs to acquire a lock on the required data items.
3. A lock is a variable associated with each data item that indicates whether read or write operations can be applied to it. Acquiring the lock by modifying its value is called locking.
4. Database systems mainly use two modes of locking. They are exclusive locks and shared locks. Exclusive lock provides a transaction an exclusive control on the data item. Shared lock can be acquired on a data item when a transaction wants to only read a data item and not modify it.
5. The locking or unlocking of the data items is implemented by a subsystem of the database system known as lock manager.
6. The lock manager maintains a linked list of records for each locked data item in the order in which the requests arrive. It uses a hash table known as lock table, which is indexed on the data item identifier.
7. Two-phase locking requires that each transaction be divided into two phases. During the first phase, the transaction acquires all the locks; during the second phase, it releases all the locks. The phase during which locks are acquired is a growing (or expanding) phase. On the other hand, a phase during which locks are released is a shrinking (or contracting) phase.
8. In strict two-phase, a transaction does not release any of its exclusive-mode locks until that transaction commits or aborts.
9. In rigorous two-phase locking, a transaction does not release any of its locks (both exclusive and shared) until that transaction commits or aborts.
10. A transaction is also permitted to change the lock from one mode to another on the data item on which it already holds a lock. This is known as lock conversion.
11. In tree locking, a transaction can acquire only exclusive locks on data items. Transactions are allowed to unlock a data item at any time;

however, a data item released once cannot be relocked.

12. In predicate locking technique, all the tuples (whether existing or new tuples that are to be inserted) that satisfy an arbitrary predicate are locked to avoid phantom problem.
13. There are two techniques for concurrency control in tree-structured indexes like B^+-trees. The first technique is crabbing. The second technique uses a variant of the B^+-tree called B-link tree, in which every node of the tree maintains a pointer that refers to its right sibling.
14. Locking granularity is the size of the data item that the lock protects. Locking a large data item such as either an entire relation or a database is termed as coarse granularity, whereas, locking a small data item such as either a tuple or an attribute is termed as fine granularity.
15. In intention lock, a transaction intends to explicitly lock a lower level of the tree. When intention-mode is associated with shared-mode, it is called intention-shared (IS) mode, and when it is associated with exclusive-mode, it is called intention-exclusive (IX) mode.
16. Timestamp-based concurrency control is a non-lock concurrency control technique; hence, deadlocks cannot occur. Timestamp is a unique identifier assigned to each transaction in the order it begins.
17. A number of implementation techniques based on the timestamp ordering have been proposed for concurrency control techniques, which ensure that all the conflicting operations of the transactions are executed in timestamp order. Three of them are basic timestamp ordering, strict timestamp ordering, and Thomas' write rule.
18. In optimistic concurrency control techniques, it is assumed that the transactions do not directly update the data items until they finish their execution.
19. In multiversion technique, several versions (or values) of a data item are maintained. When a transaction issues a write operation on the data item Q, it creates a new version of Q, the old version of Q is retained.
20. In multiversion two-phase locking, other transactions are allowed to

read a data item while a transaction still holds an exclusive lock on the data item.

21. Deadlock is a situation that occurs when all the transactions in a set of two or more transactions are in a simultaneous wait state and each of them is waiting for the release of a data item held by one of the other waiting transaction in the set.
22. Different approaches to prevent deadlocks are conservative 2PL, assigning an order to data items and using timestamps along with locking.
23. To detect a deadlock, the system maintains a directed graph called wait-for graph. If there exists a cycle in the wait-for graph, it indicates the deadlock in the system.
24. One of the deadlocked transactions chosen to be aborted is termed as the victim transaction.

KEY TERMS

- Concurrency control techniques
- Lock
- Locking
- Exclusive lock
- Shared lock
- Lock compatibility
- Lock status
- Lock manager
- Starvation
- Lock table
- Lock request
- Unlock request
- Deadlock
- Locking technique
- Two-phase locking
- Growing phase
- Shrinking phase
- Lock point

- Strict two-phase locking
- Rigorous two-phase locking
- Lock conversion
- Upgrading
- Downgrading
- Graph-based locking
- Database graph
- Tree-locking
- Phantom problem
- Predicate locking
- Index locking
- Tree-structured indexes
- Crabbing
- B-link tree
- Multiple-granularity locking
- Locking granularity
- Coarse granularity
- Fine granularity
- Intention lock mode
- Intention-shared (IS) mode
- Intention-exclusive (IX) mode
- Shared and intention-exclusive (SIX) mode
- Resource contention
- Data contention
- Blocking
- Thrashing
- Hot spots
- Timestamp-based technique
- Timestamp
- System clock
- Logical counter
- Timestamp ordering
- Read timestamp
- Write timestamp

- Basic timestamp ordering
- Cascading rollback
- Strict timestamp ordering
- Recoverability
- Thomas' write rule
- Optimistic or validation technique
- Read phase
- Validation Phase
- Validation test
- Write phase
- Critical section
- Multiversion technique
- Certified version
- Uncertified version
- Multiversion timestamp ordering
- Multiversion two-phase locking
- Certify lock mode
- Deadlock prevention
- Conservative 2PL
- Ordering of data items
- Timestamp with locking
- Wait-die technique
- Wound-wait technique
- Deadlock detection and recovery
- Wait-for graph

EXERCISES

A. Multiple Choice Questions

1. Strict two-phase locking does not ensure
    1. Cascadelessness
    2. Freedom from deadlock
    3. Serializability
    4. None of these

2. If a transaction T_i holds a lock on some data item Q in shared and intention-exclusive (SIX) mode, then which of the following is true for another transaction T_j?
    1. T_j can lock Q in exclusive (X) mode
    2. T_j can lock Q in intention-exclusive (IX) mode
    3. T_j can lock Q in intention-shared (IS) mode
    4. T_j can lock Q in shared (S) mode
3. Strict timestamp ordering technique ensures
    1. Serializability
    2. Freedom from deadlock
    3. Recoverability
    4. All of these
4. Which of the following is not true in case of tree-locking?
    1. It does not ensure freedom from deadlock
    2. Unlike two-phase locking, transaction can unlock data items earlier
    3. It ensures shorter waiting times and greater amount of concurrency
    4. All of these
5. The number of modes in which a transaction may request a lock in multiple-granularity locking is
    1. Two
    2. Three
    3. Five
    4. Six
6. Which of the following is true for timestamp ordering?
    1. Basic timestamp ordering does not ensure conflict serializability
    2. Strict timestamp ordering ensures freedom from deadlock
    3. Thomas' write rule enforces conflict serializability
    4. None of these
7. Which phase is not a part of the optimistic technique?
    1. Execution phase
    2. Read phase
    3. Write phase

    4. Validation phase
8. The techniques used to handle the phantom problem are
    1. Predicate locking
    2. Index locking
    3. Timestamping
    4. Both (a) and (b)
9. Consider the following wait-for graph.

    Which of the following statement does not hold true for this wait-for graph?

    1. Transaction T_1 is waiting for the data item locked by transaction T_4
    2. Transaction T_2 is waiting for the data item locked by transaction T_1
    3. If transaction T_2 requests for the data item locked by transaction T_3, deadlock occurs
    4. If transaction T_4 requests for the data item locked by transaction T_2, deadlock occurs
10. Which of the following timestamp is not associated with a transaction $\mathtt{T}$?
    1. Start (T)
    2. Read (T)
    3. Validation (T)
    4. Finish (T)

B. Fill in the Blanks

1. A set of rules that all the transactions in a schedule must follow is called _____.
2. A database system mainly uses two modes of locking, namely, _____ and _____.
3. The lock on a data item can be downgraded only in _____ phase.
4. In _____ locking, a transaction does not release any of its locks until it commits or aborts.
5. _____ and _____ techniques are used for concurrency control in

tree-structured indexes.

6. _____ permits each transaction to use levels of locking that are most suitable for its operations.
7. In the compatibility matrix for different access modes, the highest privilege is given to _____ and the lowest privilege is given to _____.
8. The frequently accessed data items that remain in buffer for long period of time are known as _____.
9. _____ ensure that modifications made by the younger transaction are not visible to the older transaction.
10. Two deadlock prevention techniques that use timestamps are _____ and _____.

C. Answer the Questions

1. What is concurrency control? How is it implemented in DBMS? Explain.
2. What is deadlock? Explain how do you prevent and detect it. Give suitable examples.
3. Define the following.
    1. Lock
    2. Coarse granularity and Fine granularity
    3. Critical section
    4. Certified version
    5. Victim transaction
    6. Predicate locking
    7. Database graph
    8. Tree-locking
4. What do you understand by lock compatibility? Discuss in detail with examples.
5. How is locking implemented? What is the role of lock table in implementation? How are requests to lock and unlock a data item handled?
6. Explain the two-phase locking protocol with the help of an example. What are its disadvantages? How can these disadvantages be

overcome? What is the benefit of rigorous two-phase locking?

7. How one can change the mode of lock over a data item? What do you understand by lock upgrade and lock downgrade?
8. Discuss the graph-based locking technique. What is the role of database graph?
9. What are the two factors that govern the performance of locking? Discuss how blocking of transactions can degrade the performance.
10. All schedules for a set of transactions under the tree locking are conflict serializable. Justify this statement by taking suitable example.
11. What is phantom problem for concurrency control and how index locking resolves this problem?
12. Discuss two techniques for concurrency control in tree-structured indexes like B^+-trees.
13. What is multiple-granularity locking? Under what situations it is used?
14. What are intention locks? How does it provide higher degree of concurrency?
15. What is timestamp? How system generates timestamps? What are the values associated with timestamps?
16. What is timestamp ordering protocol for concurrency control? How is strict timestamp ordering protocol different from basic timestamp ordering?
17. How optimistic concurrency control techniques are different from other concurrency control techniques? Why they are also called validation techniques? Discuss the typical phases of an optimistic technique.
18. What is deadlock? Explain how do you prevent and detect it with the help of an example.
19. Discuss “wait-die” and “wound-wait” approaches of deadlock avoidance. Compare these approaches of deadlock prevention with a deadlock prevention approach in which data items are locked in a particular order.
20. How deadlock is detected by wait-for graph? What are the measures

for recovery from deadlock? Explain.

21. Discuss multiversion technique, for what purpose is it used? Discuss multiversion timestamp ordering and multiversion two-phase locking. Compare multiversion two-phase locking with basic two-phase locking.
22. Write short note on the following.
    1. Deadlock prevention protocols
    2. Locks and its types
    3. Intention lock modes
    4. Lock conversion
    5. Lock point
    6. Thomas' write rule
    7. Locking rules for multiple-granularity locking
23. Draw suitable graph for the following requests and find whether the transactions are deadlocked or not.

T_1	T_2	T_3
S_lock A	–	–
–	X_lock B	–
–	S_lock C	–
–	–	X_lock C
–	S_lock A	–
S_lock B	–	–
S_lock A	–	–
–	–	S_lock A
All the locking requests start from here		