19.4 THE PROBLEM OF CONCURRENCY CONTROL

Typically, in a database system, several transactions are under execution simultaneously. Since a transaction preserves database consistency, a database system can guarantee consistency by executing transactions serially, i.e., one at a time. However, such a serial execution of transactions is inefficient as: it results in poor response to user requests and poor utilization of system resources. Efficiency can be improved by executing transactions concurrently, that is, by executing read and write actions from several transactions in an interleaved manner. Because the actions of concurrently running transactions may access the same data objects, several anomalous situations may arise if the interleaving of actions is not controlled in some orderly way. Such situations are described next.

19.4.1 Inconsistent Retrieval

Inconsistent retrieval occurs when a transaction reads some data objects of a database before another transaction has completed with its modification of those data objects. In such situations, the former transaction faces the risk of retrieving incorrect values of the data objects.

Example 19.2. Suppose customer c_1 transfers \$500 from savings account S to checking account C, and teller t_1 concurrently reads both the accounts to compute the total balance. A possible trace of the execution of these transactions is as follows (suppose initially, S = 1000 and C = 500): c_1 reads S into its workspace, subtracts 500 from it, and writes it back to S; t_1 reads S (=500) and C (=500) into its workspace; c_1 reads C (=500) into its workspace, adds 500 to it, and writes it back to C (=1000); t_1 outputs 1000 as the balance. Here, t_1 reads S after c_1 has modified it and reads C before c_1 has modified it, resulting in the incorrect retrieval of the total balance.

19.4.2 Inconsistent Update

Inconsistent update occurs when many transactions read and write onto a common set of data objects of a database, leaving the database in an inconsistent state.

Example 19.3. Suppose two data objects A and B, which satisfy the consistency assertion "(A = 0) or (B = 0)", are concurrently modified by the following transactions [17]:

$$\text{“}T_1 : if\, A = 0 \text{ then } B := B + 1\text{”}$$

$$\text{“}T_2 : if\, B = 0 \text{ then } A := A + 1\text{”}.$$

A possible execution trace is as follows (initially A = 0 and B = 0): T_1 reads A (= 0) and B (= 0) in its workspace; T_2 reads A (= 0) and B (= 0) in its workspace; since A = 0 in the workspace of T_1, it increments B by 1 and writes it in the database (B = 1); since B = 0 in the workspace of T_2, it increments A by 1 and writes it in the database (A = 1); the final database state "(A = 1) and (B = 1)" is inconsistent.

Thus, if the interleaving of the actions of transactions is not controlled, some transactions may see an inconsistent state of the database and the database may be left in an inconsistent state. This fundamental problem is referred to as the *concurrency control* problem. In a database system, this problem is handled by a concurrency control mechanism that controls the relative order (or interleaving) of conflicting† actions, such that every transaction sees a consistent state of the database and, when all transactions are over, the database is in a consistent state. Nevertheless, the concurrency control mechanism exploits the underlying concurrency.

It is clear that the concurrent execution of transactions must be controlled to ensure database consistency. However, a question arises as to what degree the concurrency must be controlled to ensure that database consistency is maintained (obviously, it is too restrictive to execute transactions serially). We answer this question next and we state restrictions on the concurrency by characterizing the interleavings of transaction actions that produce correct results.

19.5 SERIALIZABILITY THEORY

In this section, we describe the theory of serializability, which gives precise rules and conditions under which a concurrent execution of a set of transactions is correct [4, 7, 15, 16]. A concurrency control algorithm is correct if all of its possible executions are correct. Since the execution of transactions is modeled by a log and the correctness condition is stated in terms of logs, we next introduce the concept of log.

19.5.1 Logs

The serializability theory models executions of a concurrency control algorithm by a history variable called the *log* [7] (also called the *schedule* in [8] and the *history* in [16]). A log captures the chronological order in which read and write actions of transactions are executed under a concurrency control algorithm. Let $T = \{T_0, T_1, ..., T_n\}$ be a transaction system. A log over T models an interleaved execution of $T_0, T_1, ..., T_n$ and is a partial order set $L = (S, <)$ where,

1. $S = \bigcup_{i=0}^{n} S_i$, and
2. $< \supseteq \bigcup_{i=0}^{n} <_i$

Condition (1) states that the database system executes all the actions submitted only by $T_0, T_1, ..., T_n$ and condition (2) states that the database system executes the actions in the order expected by each transaction.

Example 19.4. Figure 19.3 shows three transactions T_1, T_2, and T_3 and two logs L1 and L2 over these transactions. Notations used are as follows: ri[x] and wi[x]", respec-

†Recall that two actions conflict if they operate on the same data object, and at least one of them is a write action.

T_1 = r1[x] r1[z] w1[x]

T_2 = r2[y] r2[z] w2[y]

T_3 = w3[x] r3[y] w3[z]

L1 = w3[x] r1[x] r3[y] r2[y] w3[z] r2[z] r1[z] w2[y] w1[x]

L2 = w3[x] r3[y] w3[z] r2[y] r2[z] w2[y] r1[x] r1[z] w1[x]

FIGURE 19.3
Examples of logs.

tively, denote the read and the write operation of transaction T_i on data object x.

19.5.2 Serial Logs

In a database system, if transactions are executed strictly serially, that is, all the actions of each transaction must complete before any action of the next transaction can start, then the resulting log is termed a *serial log* [7]. A serial log represents an execution of transactions where actions from different transactions are not interleaved. For example, for a set of transactions T_1, T_2,..., T_n, a serial log is of the form T_{i1} T_{i2} T_{in}, where $i1$, $i2$,..., in, is a permutation of 1, 2,..., n.

Example 19.5. Log L2 of Fig. 19.3 is an example of a serial log because actions from different transactions have not been interleaved.

Since each transaction individually maintains the database consistency, it follows by induction that a serial log maintains the database consistency.

19.5.3 Log Equivalence

Two logs are equivalent if all the transactions in both the logs see the same state of the database and leave the database in the same state after all the transactions are finished. Let L be a log over a transaction system T = $\{T_0, T_1, ..., T_n\}$ and on a database system D = (x, y, z,...). If w_i[x] and r_j[x] are two operations in L, then we say r_j[x] reads from w_i[x] iff,

1. w_i[x] < r_j[x] and
2. There is no w_k[x] such that w_i[x]<w_k[x]<r_j[x].

Example 19.6. In log L1 of Fig. 19.3, action r1[x] reads x from action w3[x] and action r2[z] reads z from action w3[z].

We call w_i[x] a final write, if there is no w_k[x] such that w_i[x]<w_k[x].

Example 19.7. In log L1 of Fig. 19.3, w3[z], w2[y] and w1[x] are the final writes.

Two logs over a transaction system are equivalent iff

1. Every read operation reads from the same write operation in both the logs, and
2. Both the logs have the same final writes.

Condition (1) ensures that every transaction reads the same value from the database in both the logs and condition (2) ensures that the final state of the database is same in both the logs.

Example 19.8. In Fig. 19.3, log L2 is equivalent to log L1.

19.5.4 Serializable Logs

Note that serial logs are correct because each transaction sees a consistent state of the database and when all the transactions terminate, the database is in a consistent state. However, serial logs result in poor performance. Therefore, there has been a motivation to find out if a log obtained by interleaving actions from several transactions produces the same effect as a serial log. Such logs are called *serializable logs*. Formally, a log obtained by interleaving actions of transactions T_1,T_2,...,T_n is serializable if it produces the same output and has the same effect on the database as the serial execution of a permutation of T_1, T_2,..., T_n. Thus, a serializable log is equivalent to a serial log and represents a correct execution.

Example 19.9. In Fig. 19.3, log L1 is equivalent to serial log L2, hence, it represents a correct execution.

19.5.5 The Serializability Theorem

It is natural to ask what conditions an interleaved execution (log) should satisfy in order to be serializable. Several researchers (e.g., [15, 16, 18]) have investigated this problem and have stated the condition in terms of a graph, called a *serialization graph*, which is constructed from a log. In this section, we present the results as a theorem (called *the serializability theorem*), which states the required conditions for serializability.

Suppose L is a log over a set of transactions $\{T_0, T_1,..., T_n\}$. The serialization graph for L, SG(L), is a directed graph whose nodes are T_0, T_1,..., T_n and which has all the possible edges satisfying the following condition: There is an edge from T_i to T_j provided for some x, either $r_i[x] < w_j[x]$, or $w_i[x] < r_j[x]$, or $w_i[x] < w_j[x]$. Note that an edge $T_1 \rightarrow T_2$ in a serialization graph, SG(L), denotes that an action of T_1 precedes a conflicting action of T_2 in log L.

Example 19.10. The serialization graph for log L1 of Fig. 19.3 is shown in Fig. 19.4.

THE SERIALIZABILITY THEOREM.

Theorem 19.1. A log L is serializable iff SG(L) is acyclic.

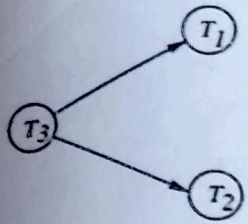

FIGURE 19.4
The serialization graph of L1.

Proof of this theorem is beyond the scope of this book (interested readers should refer to [5] for details). Given an acyclic SG(L), we can determine a serial log corresponding to log L by topologically sorting the SG(L).

Example 19.11. The serialization graph of L1, SG(L1), in Fig. 19.4 is acyclic; therefore, L1 is serializable (which we have already confirmed by showing that it is equivalent to a serial log L2).

CHAPTER 20

CONCURRENCY CONTROL ALGORITHMS

20.1 INTRODUCTION

A concurrency control algorithm controls the interleaving of conflicting actions of transactions so that the integrity of a database is maintained, i.e., their net effect is a serial execution. In this chapter, we discuss several popular concurrency control algorithms. We begin by describing the basic synchronization primitives used by these algorithms.

20.2 BASIC SYNCHRONIZATION PRIMITIVES

20.2.1 Locks

In lock based techniques, each data object has a lock associated with it [8]. A transaction can request, hold, or release the lock on a data object. When a transaction holds a lock, the transaction is said to have locked the corresponding data object. A transaction can lock a data object in two modes: *exclusive* and *shared*. If a transaction has locked a data object in exclusive mode, no other transaction can concurrently lock it in any mode. If a transaction has locked a data object in shared mode, other transactions can concurrently lock it but *only* in shared mode. Basically, by locking data objects, a transaction ensures that the locked data objects are inaccessible to other transactions, while temporarily in inconsistent states.

20.2.2 Timestamps

A timestamp is a unique number that is assigned to a transaction or a data object and is chosen from a monotonically increasing sequence. Timestamps are commonly generated according to Lamport's scheme [17]. Every site S_i has a logical clock C_i, which takes monotonically nondecreasing integer values. When a transaction T is submitted at a site S_i, S_i increments C_i by one and then assigns a 2-tuple (C_i, i) to T. The 2-tuple is referred to as the timestamp of T and is denoted by TS(T). Every message contains the current clock value of its sender site, and when a site S_j receives a message with clock value t, it sets C_j to max($t + 1$, C_j). For any two timestamps $ts_1 = (t_1, i_1)$ and $ts_2 = (t_2, i_2)$, $ts_1 < ts_2$, if either ($t_1 < t_2$), or ($t_1 = t_2$ and $i_1 < i_2$).

Timestamps have two properties: (1) *uniqueness* (i.e., they are unique systemwide) because timestamps generated by different sites differ in their site id part and timestamps generated by the same site differ in their clock value part and (2) *monotonicity* (i.e., the value of timestamps increases with time) because a site generates timestamps in increasing order.

Timestamps allow us to place a total ordering on the transactions of a distributed database system by simply ordering the transactions by their timestamps. In concurrency control algorithms for distributed database systems, whenever two concurrent transactions conflict, all sites must agree on a common order of serialization. This can be achieved by assigning timestamps to transactions in the manner described above and then having every site serialize conflicting transactions by their timestamps.

20.3 LOCK BASED ALGORITHMS

In *lock* based concurrency control algorithms, a transaction must lock a data object before accessing it [8]. In a locking environment, a transaction T is a sequence $\{a_1(d_1), a_2(d_2), \ldots, a_n(d_n)\}$ of n actions, where a_i is the operation performed in the ith action and the d_i is the data object acted upon in ith action. In addition to read and write, lock and unlock are also permissible actions in locking algorithms. A transaction can lock a data object d_i with a "lock(d_i)" action and can relinquish the lock on d_i by an "unlock(d_i)" action. A log that results from an execution where a transaction attempting to lock an already locked data object waits, is referred to as a *legal* log [8].

A transaction is *well-formed* [8] if it

- Locks a data object before accessing it,
- Does not lock a data object more than once, and
- Unlocks all the locked data objects before it completes.

It is important to note that just being well-formed is not sufficient for correctness (that is, to guarantee serializability). Additional constraints, as to when a lock can be acquired and released, are needed. These constraints are expressed as locking algorithms. Next, locking algorithms are described.

20.3.1 Static Locking

In static locking, a transaction acquires locks on all the data objects it needs before executing any action on the data objects. Static locking requires a transaction to predeclare all the data objects it needs for execution. A transaction unlocks all the locked data objects only after it has executed all of its actions.

Static locking is conceptually very simple. However, it seriously limits concurrency because any two transactions that have a conflict must execute serially. This may significantly limit the performance of the underlying database system. Another drawback of static locking is that it requires a priori knowledge of the data objects to be accessed by transactions. This may be impractical in applications where the next data object to be locked depends upon the value of another data object.

20.3.2 Two-Phase Locking (2PL)

Two-phase locking is a dynamic locking scheme in which a transaction requests a lock on a data object when it needs the data object. However, database consistency is not guaranteed if a transaction unlocks a locked data object immediately after it is done with it.

Two-phase locking imposes a constraint on lock acquisition and the lock release actions of a transaction to guarantee consistency [8]. In two-phase locking, a transaction cannot request a lock on any data object after it has unlocked a data object. Thus, a transaction must have acquired locks on all the needed data objects before unlocking a data object.

Thus, as the name suggests, two-phase locking has two phases: a *growing phase* during which a transaction requests locks (without releasing any lock); and, a *shrinking phase*, which starts with the first unlock action, during which a transaction releases locks (without requesting any more locks). The stage of a transaction when the transaction holds locks on all the needed data objects is referred to as its *lock point*. A schematic diagram of the execution of a two-phase transaction is shown in Fig. 20.1.

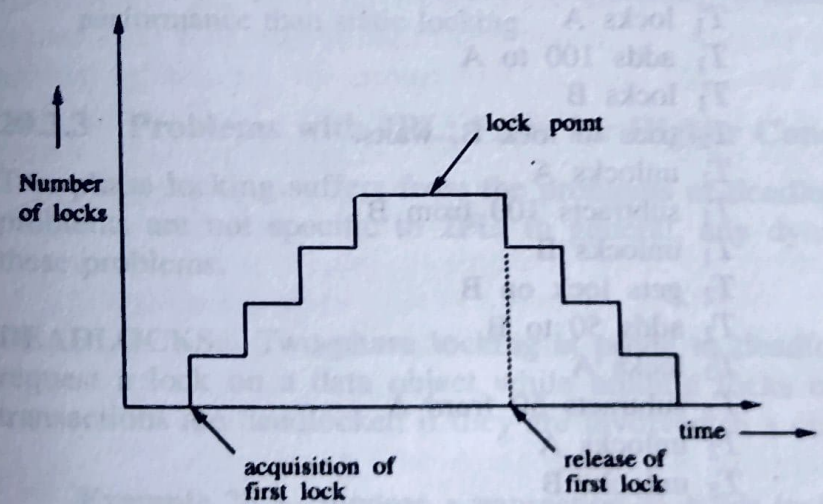

FIGURE 20.1
A schematic diagram of a two-phased transaction.

T_1	T_2
lock A	lock B
A + 100 → A	B + 50 → B
lock B	lock A
unlock A	A − 50 → A
B − 100 → B	unlock B
unlock B	unlock A

FIGURE 20.2
Well-formed, two-phased transactions.

Example 20.1. Figure 20.2 shows two well-formed, two-phased transactions T_1 and T_2. Transaction T_1 transfers \$100 from account B to account A, and transaction T_2 transfers \$50 from account A to account B. In Fig. 20.3, we show a legal schedule of T_1 and T_2, which is serializable.

Eswaran et al. [8] shows that if a set of transactions are well-formed and follow the two-phase structure for requesting and releasing data objects, then all legal logs (legal schedules) are serializable. Minoura [20] shows that if no semantic information on transactions and the database system are available, then two-phase locking is a necessary condition for database consistency.

Two-phase locking increases concurrency over static locking because locks are held for a shorter period. With the help of an example, we next show how 2PL results in higher concurrency.

Example 20.2. Suppose two transactions T_1 and T_2 have the following readsets and writesets:

$$RS(T_1) = \{d_2, d_3\}, WS(T_1) = \{d_3\},$$
$$RS(T_2) = \{d_1, d_2, d_3\}, WS(T_2) = \{d_1, d_2, d_3\}.$$

Transaction	Action	Comments
T_1	lock A	T_1 locks A
T_1	A+100 → A	T_1 adds 100 to A
T_1	lock B	T_1 locks B
T_2	lock B	T_2 tries to lock B, waits.
T_1	unlock A	T_1 unlocks A
T_1	B−100 → B	T_1 subtracts 100 from B
T_1	unlock B	T_1 unlocks B
		T_2 gets lock on B
T_2	B+50 → B	T_2 adds 50 to B
T_2	lock A	T_2 locks A
T_2	A−50 → A	T_2 subtracts 50 from A
T_2	unlock A	T_2 unlocks A
T_2	unlock B	T_2 unlocks B

FIGURE 20.3
A legal and serializable schedule of T_1 and T_2

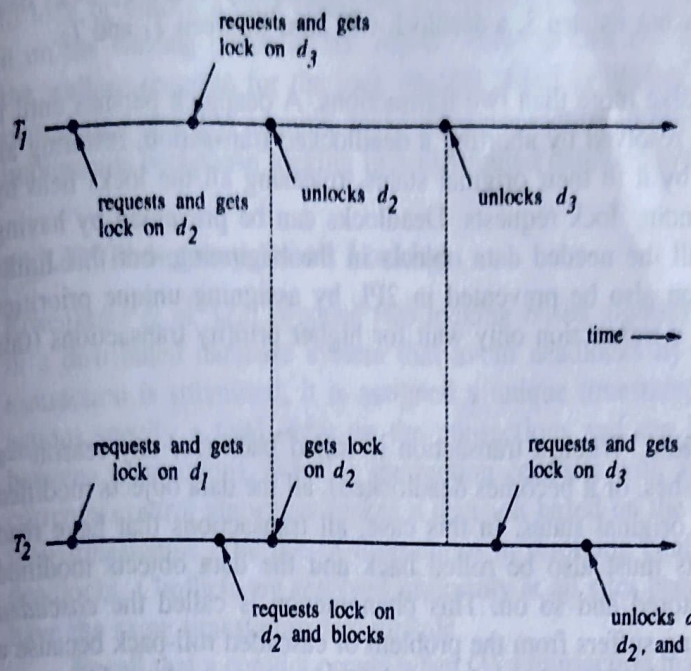

FIGURE 20.4
Concurrent execution of T_1 and T_2.

One possible execution of these transaction is shown in Fig. 20.4. T_1 begins execution first and places a read lock on d_2. T_2 begins by placing a write lock on d_1, performs some computation, and is blocked when it tries to lock d_2. T_1 locks d_3 and can now unlock d_2 since it has locked all data items it will need. This allows T_2 to lock d_2 for write and T_2 can continue computation. After T_1 has unlocked d_3, T_2 can lock d_3 and complete. Thus, T_1 and T_2 can concurrently execute in two-phase locking. In static locking, however, T_2 would not have been able to begin until T_1 had finished. Consequently, 2PL allows more concurrency in transaction execution and has better performance than static locking.

20.3.3 Problems with 2PL: Price for Higher Concurrency

Two-phase locking suffers from the problems of deadlock and cascaded aborts. These problems are not specific to 2PL; in general, any dynamic locking policy will have these problems.

DEADLOCKS. Two-phase locking is prone to deadlocks because a transaction can request a lock on a data object while holding locks on other data objects. A set of transactions are deadlocked if they are involved in a circular wait.

Example 20.3. Suppose a transaction T_1 holds lock on data object d_1 and requests lock on data object d_2 and is blocked because d_2 is already locked by transaction T_2. Later, if transaction T_2, while holding a lock on d_2 requests a lock on d_1, a deadlock will arise between T_1 and T_2.

Example 20.4. In Fig. 20.3, if T_1 locks data object A at step 1 and T_2 locks data object B before T_1 reaches its step 3, a deadlock will arise between T_1 and T_2.

A deadlock may involve more than two transactions. A deadlock persists until it is resolved. A deadlock is resolved by aborting a deadlocked transaction, restoring all the data objects modified by it to their original states, releasing all the locks held by it, and withdrawing its pending lock requests. Deadlocks can be prevented by having each transaction acquire all the needed data objects in the beginning, but this limits concurrency. Deadlocks can also be prevented in 2PL by assigning unique priorities to transactions and having a transaction only wait for higher priority transactions (this scheme is discussed later).

CASCADED ROLL-BACKS. When a transaction is rolled back (for any reason—a user kills it, the system crashes, or it becomes deadlocked), all the data objects modified by it are restored to their original states. In this case, all transactions that have read the backed up data objects must also be rolled back and the data objects modified by them must also be restored and so on. This phenomenon is called the *cascaded roll-back*. Two-phase locking suffers from the problem of cascaded roll-back because a transaction may be rolled back after it has released the locks on *some* data objects and other transactions have read those modified data objects.

Example 20.5. In Example 20.2, if T_1 were to abort after it released the lock on d_2 and after T_2 had read d_2, then T_2 would need to be aborted too, because now T_2 has read a value of d_2 that was never committed (T_1 did not complete).

STRICT 2PL. Cascaded roll-backs can be avoided by making all transactions strict two-phased. In strict two-phase locking, a transaction holds all its locks until it completes and releases them in a single atomic action, often called a *commit*. Strict 2PL eliminates cascaded aborts because transactions can read data objects modified by a transaction only after the transaction has completed. However, strict 2PL reduces concurrency as a transaction holds locks for a longer period than required for consistency.

Clearly, the problems of deadlock and cascaded aborts are created by the two phases of 2PL, the growing and shrinking phases, respectively. To eliminate these problems, it is necessary to avoid these two phases, and consequently return to static locking. Thus, the price of higher concurrency in 2PL are these two problems.

20.3.4 2PL in DDBS

The concurrency control problem is aggravated in a distributed database system because [18],

- Users access data objects stored in several geographically distant sites.
- A site may not have instantaneous knowledge of the state of other sites.

Two-phase locking can be implemented in a distributed database system in the following way. A DM (data manager) at a site controls the locks associated with objects

stored at that site. A TM (transaction manager) communicates with the appropriate DM to lock or unlock a data object. If a request for lock cannot be granted, the DM puts it on the waiting queue of the object. When a lock on an object is released, one of the waiting requests for the lock on that object is granted. If all the transactions are two-phased or a TM acquires locks for a transaction in the two-phased manner, then it implements two-phase locking in a distributed database system.

20.3.5 Timestamp-Based Locking

Rosenkratz et al. [22] propose two locking based algorithms for concurrency control in a distributed database system that avoid deadlocks by using timestamps. When a transaction is submitted, it is assigned a unique timestamp. The timestamps of transactions specify a total order on the transactions and can be used to resolve conflicts between transactions. When a transaction conflicts with another transaction, the concurrency control algorithm makes a decision based on the result of the comparison of their timestamps. The use of timestamps in resolving conflicts is primarily to prevent deadlocks. Conflicts are resolved uniformly at all sites because conflicting transactions have the same timestamps systemwide.

Recall that a conflict occurs when (1) a transaction makes a read request for a data object, for which another transaction currently has a write access or (2) a transaction makes a write request for a data object, for which another transaction currently has a write or read access.

CONFLICT RESOLUTION. A conflict is resolved by taking one of the following actions.

Wait. The requesting transaction is made to wait until the conflicting transaction either completes or aborts.

Restart. Either the requesting transaction or the transaction it conflicts with is aborted (all data objects modified by the aborted transaction are restored to their initial states) and started afresh. Restarting is achieved by using one of the following primitives:

Die. The requesting transaction aborts and starts afresh.

Wound. The transaction in conflict with the requesting transaction is tagged as wounded and a message "wounded" is sent to all sites that the wounded transaction has visited. If the message is received before the wounded transaction has committed at a site, the concurrency control algorithm at that site initiates an abort of the wounded transaction, otherwise the message is ignored. If a wounded transaction is aborted, it is started again. The requesting transaction proceeds after the wounded transaction completes or aborts.

WAIT-DIE ALGORITHM. The WAIT-DIE algorithm is a nonpreemptive algorithm because a requesting transaction never forces the transaction holding the requested data object to abort. The algorithm works as follows. Suppose requesting transaction T_1 is

in conflict with a transaction T_2. If T_1 is older (i.e., has a smaller timestamp), then T_1 waits, otherwise T_1 dies (and starts afresh).

WOUND-WAIT ALGORITHM. The WOUND-WAIT algorithm is a preemptive algorithm and works as follows. Suppose a requesting transaction T_1 is in conflict with a transaction T_2. If T_1 is older, it wounds T_2, otherwise it waits.

Both these algorithms produce serializable logs and guarantee that no transaction waits forever to prevent deadlocks.

COMPARISON BETWEEN THE ALGORITHMS

Waiting Time. In the WAIT-DIE algorithm, an older transaction is made to wait for younger ones. Hence, the older a transaction becomes, the higher the number of younger transactions it waits for and the more it tends to slow down.

In the WOUND-WAIT algorithm, an older transaction never waits for younger ones and wounds all the younger transactions that conflict with it. Hence, the older a transaction becomes, the less it tends to slow down.

Number of Restarts. In the WAIT-DIE algorithm, the younger requester dies and is restarted. If this younger transaction is restarted with the same timestamp, it might again conflict with the older transaction (if still running) and again die. Thus, a younger transaction may die and restart several times before it completes.

In the WOUND-WAIT algorithm, if the requester is younger, it waits rather than continuously dying and restarting.

20.3.6 Non-Two-Phase Locking

When the data objects of a database system are hierarchically organized (i.e., hierarchical database systems [29]), a non-two-phase locking protocol can ensure serializability and freedom from deadlock [24]. In non-two-phase locking, a transaction can request a lock on a data object even after releasing locks on some data objects. However, a data object cannot be locked more than once by the same transaction.

In order to access a data object, a transaction must first lock it. If a transaction attempts to lock a data object that is already locked, the transaction is blocked. When a transaction unlocks a data object, one of the transactions waiting for it gets a lock on it and resumes. When a transaction T_i starts, it selects a data object (denoted by $E(T_i)$) in the database tree for locking and can subsequently lock the data objects only in the subtree with root node $E(T_i)$. Moreover, a transaction can lock a data object only if its immediate ancestor is also currently locked by it.

Example 20.6. Figure 20.5 shows a hierarchical database system and Fig. 20.6 shows two non-two-phased transactions T_1 and T_2. Transaction T_1 must lock the node B first so that it can lock all the required data objects (D, G, and I). Likewise, T_2 must lock node A before accessing any other node. Before T_2 can lock node H, it must first lock node C.

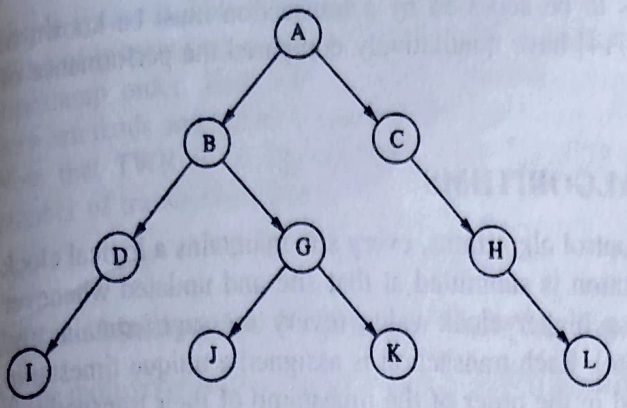

FIGURE 20.5
A hierarchical database system.

THE LOCKING PROTOCOL

1. T_i can lock data object R $\neq$ E(T_i) iff T_i is holding a lock on R's ancestor.
2. After unlocking a data object, T_i cannot lock it again.
3. T_i can only access those data objects for which it is holding a lock.

Silberschatz and Kedem [24] show that the non-two-phase locking protocol guarantees serializability and is deadlock free. Intuitively, serializability is achieved because data objects are locked in ascending order (i.e., from the root to leaves) in a tree that is acyclic. Deadlocks are avoided because the tree structure puts an order on data objects and the first rule of the protocol guarantees that data objects are requested (locked) in ascending order.

ADVANTAGES. Non-two-phase locking has two advantages over two-phase locking. First, it is free from deadlocks and hence, no transaction is aborted to resolve deadlocks. Second, a lock can be released when it is no longer needed (rather than waiting for a moment when all the required locks are set). Hence, the availability of data objects to other transactions is higher. However, the database must be organized as a tree and

T_1	T_2
lock D	lock H
D+100 → D	H+200 → H
lock I	unlock H
I−50 → I	lock D
unlock D	D−100 → D
lock G	unlock D
G*2 → G	
unlock I	
unlock G	

FIGURE 20.6
Non-two-phased transactions T_1 and T_2.

a super set of all the data objects to be accessed by a transaction must be known in advance. Singhal and Joergensen [14] have qualitatively compared the performance of 2PL and N2PL algorithms.

20.4 TIMESTAMP BASED ALGORITHMS

In timestamp based concurrency control algorithms, every site maintains a logical clock that is incremented when a transaction is submitted at that site and updated whenever the site receives a message with a higher clock value (every message contains the current clock value of its sender site). Each transaction is assigned a unique timestamp and conflicting actions are executed in the order of the timestamp of their transactions. Recall that a timestamp is generated by appending the local clock time with the site identifier [17].

Timestamps can be used in two ways. First, they can be used to determine the currency or outdatedness of a request with respect to the data object it is operating on. Second, they can be used to order events (read-write requests) with respect to one another. In timestamp based concurrency control algorithms, the serialization order of transactions is selected a priori (decided by their timestamps) and transactions are forced to follow this order.

We next describe a series of timestamp based concurrency control algorithms [4]. We assume that the TM attaches an appropriate timestamp to all read and write operations. All DMs process conflicting operations in timestamp order. The timestamp order execution of conflicting operations results in their serialization.

20.4.1 Basic Timestamp Ordering Algorithm

In the basic timestamp ordering algorithm (BTO), the scheduler at each DM keeps track of the largest timestamp of any read and write processed thus far for each data object. Let us denote these timestamps by R-ts(object) and W-ts(object), respectively. Let read(x, TS) and write(x, v, TS) denote a read and a write request with timestamp TS on a data object x. (In a write operation, v is the value to be assigned to x.)

A read(x, TS) request is handled in the following manner: If TS < W-ts(x), then the read request is rejected and the corresponding transaction is aborted, otherwise it is executed and R-ts(x) is set to max{R-ts(x), TS}. A write(x, v, TS) request is handled in the following manner: If TS < R-ts(x) or TS < W-ts(x), then the write request is rejected, otherwise it is executed and W-ts(x) is set to TS.

If a transaction is aborted, it is restarted with a new timestamp. This method of restart can result in a cyclic restart where a transaction can repeatedly restart and abort without ever completing. This algorithm has storage overhead for maintaining timestamps (note that two timestamps must be kept for every data object).

20.4.2 Thomas Write Rule (TWR)

The Thomas write rule (TWR) is suitable only for the execution of write actions [28]. For a write(x, v, TS), if TS < W-ts(x), then TWR says that instead of rejecting the write,

simply ignore it. This is sufficient to enforce synchronization among writes because the effect of ignoring an obsolete write request is the same as executing all writes in their timestamp order. However, an additional mechanism is needed for synchronization between reads and writes because TWR takes care of only write-write synchronization. Note that TWR is an improvement over the BTO algorithm because it reduces the number of transaction aborts.

20.4.3 Multiversion Timestamp Ordering Algorithm

In the multiversion timestamp ordering (MTO) algorithm, a history of a set of R-ts's and < W-ts, value > pairs (called *versions*) is kept for each data object at the respective DM's. The R-ts's of a data object keep track of the timestamps of all the executed read operations, and the versions keep track of the timestamp and the value of all the executed write operations. Read and write actions are executed in the following manner:

- A read(x, TS) request is executed by reading the version of x with the largest timestamp less than TS and adding TS to the x's set of R-ts's. A read request is never rejected.

  Example 20.7. In Fig. 20.7(a), a read(x, 18) is executed by reading the version <12, V2> and the resulting history is shown in Fig. 20.7(b).

- A write(x, v, TS) request is executed in the following way: If there exists a R-ts(x) in the interval from TS to the smallest W-ts(x) that is larger than TS, then the write is rejected, otherwise it is accepted and a new version of x is created with timestamp TS.

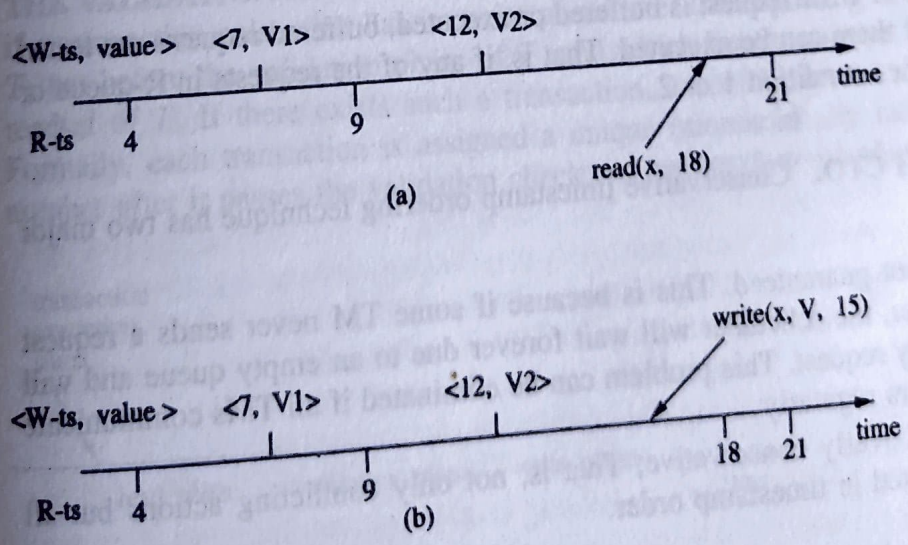

FIGURE 20.7
An example of MTO.

Example 20.8. In Figure 20.7(b), write(x, V, 15) is rejected because a read with timestamp 18 has already been executed. However, a write(x, V, 22) is accepted and is executed by creating a version <22, V> in the history.

It can be shown that the MTO algorithm is correct; i.e., every execution is equivalent to a serial execution in timestamp order. The MTO algorithm reduces the number of transaction aborts over the BTO and TWR algorithms. It does, however, require a huge amount of storage, as a set of R-ts's and multiple versions of data objects are kept for data objects. It is not practical to keep all versions of data objects—techniques exist to delete old versions.

20.4.4 Conservative Timestamp Ordering Algorithm

The conservative timestamp ordering algorithm (CTO) altogether eliminates aborts and restarts of transactions by executing the requests in strict timestamp order at all DM's. A scheduler processes a request when it is sure that there is no other request with a smaller (older) timestamp in the system.

Each scheduler maintains two queues—a R-queue and a W-queue—per TM. These queues, respectively, hold read and write requests. A TM sends requests to schedulers in timestamp order and the communication medium is order preserving. A scheduler puts a new read or write request in the corresponding queue in timestamp order. This algorithm executes read and write actions in the following way:

1. A read(x, TS) request is executed in the following way. If every W-queue is nonempty and the first write on each W-queue has a timestamp greater than TS, then the read is executed, otherwise the read(x, TS) request is buffered in the R-queue.
2. A write(x, v, TS) request with timestamp TS is executed in the following manner. If all R-queues and all W-queues are nonempty and the first read on each R-queue has a timestamp greater than TS and the first write on each W-queue has a timestamp greater than TS, then the write is executed, otherwise the write(x, v, TS) request is buffered in the W-queue.
3. When any read or write request is buffered or executed, buffered requests are tested to see if any of them can be executed. That is, if any of the requests in R-queue or W-queue satisfies condition 1 or 2.

PROBLEMS WITH CTO. Conservative timestamp ordering technique has two major problems.

- Termination is not guaranteed. This is because if some TM never sends a request to some scheduler, the scheduler will wait forever due to an empty queue and will never execute any request. This problem can be eliminated if all TMs communicate with all schedulers regularly
- The algorithm is overly conservative; That is, not only conflicting actions but all actions are executed in timestamp order.

These problems have been addressed in [4] in detail.

20.5 OPTIMISTIC ALGORITHMS

Optimistic concurrency control algorithms are based on the assumption that conflicts among transactions are rare. In optimistic algorithms, no synchronization is performed when a transaction is executed, but at the end of the transaction's execution, a check is performed to determine if the transaction has conflicted with any other concurrently running transaction. In case of a conflict, the transaction is aborted, otherwise it is committed. When conflicts among transactions are rare, very few transactions need to be rolled back. Thus, transaction roll-backs can be effectively used as a concurrency control mechanism rather than locking.

20.5.1 Kung-Robinson Algorithm

Kung and Robinson were the first to propose an optimistic method for concurrency control [16]. In their technique, a transaction always executes (tentatively) concurrently with other transactions without any synchronization check, but before its writes are written in the database (and become accessible to other transactions), it is validated. In the validation phase, it is determined whether actions of the transaction have conflicted with those of any other transaction. If found in conflict, then the tentative writes of the transaction are discarded and the transaction is restarted. The basic algorithm is as follows:

THE ALGORITHM. The execution of a transaction is divided into three phases: read phase, validation phase, and write phase. In the read phase, appropriate data objects are read, the intended computation of the transaction is done, and writes are made on a temporary storage. In the validation phase, it is checked if the writes made by the transaction violate the consistency of the database. If the check passes, then in the write phase, all the writes of the transaction are made to the database. A typical transaction execution in the optimistic approach is shown in Fig. 20.8.

THE VALIDATION PHASE. In the validation phase of a transaction T, it is checked if a transaction exists that has its write phase after the beginning of the read phase of T, but before the validation phase of T, and which has its writeset intersected by the readset of T. If there exists such a transaction, a conflict occurs and T is restarted. Formally, each transaction is assigned a unique (monotonically increasing) sequence number after it passes the validation check and before its write phase starts. Let t_s be

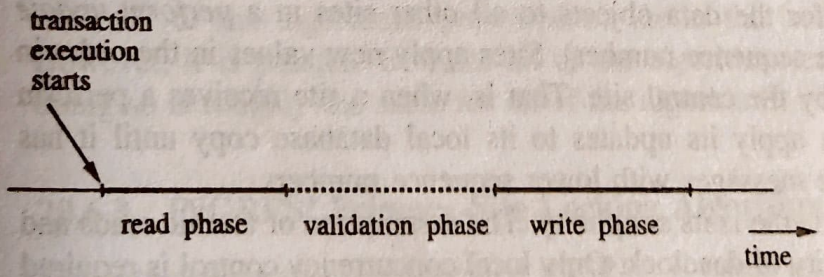

FIGURE 20.8
Transaction execution in the optimistic approach.

the highest sequence number at the start of T and t_f be the highest sequence number at the beginning of its validation phase. After the read phase of transaction T, the following algorithm is executed in a mutually exclusive manner [16] (which consists of the validation phase and a possible write phase of T):

```
<valid: = t ue;
for t:= t_s + 1 to t_f do
        if (writeset[t] ∩ readset[T] ≠ Φ) then
                valid: = false;
if valid then {write phase; increment counter,
        assign T a sequence number} >
```

A read-only transaction does not have a write phase, but it still has to be validated using the above validation algorithm. The optimistic approach is suitable only in environments where conflicts are unlikely to occur, as in a query dominant system.

Schlageter proposed an improvement to the Kung-Robinson method wherein a read transaction always proceeds without validation check and thus without the risk of restarts [23]. In the Kung-Robinson method, read transactions are treated in the same way as update transactions, and thus, are subject to a validation check with the risk of restart [16].

20.6 CONCURRENCY CONTROL ALGORITHMS: DATA