UNIT-5

Syllabus: Concurrency Control Techniques: Concurrency control, Locking Techniques for concurrency control, Time stamping protocols for concurrency control, validation-based protocol, multiple granularity, Multi version schemes, Recovery with concurrent transaction, case study of Oracle.

Concurrency Control:

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories:

- Lock-based protocols
- Timestamp-based protocols

Lock-based Protocols: Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

- **Shared**. If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.
- **Exclusive**. If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read andwrite Q.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix compatibility function

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

Figure 16.2 Transaction T1.

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Figure 16.3 Transaction T2.

Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A (Figure 16.2). Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B (Figure 16.3).

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 16.4 Schedule 1.

Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 16.4 is possible. In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

Figure 16.5 Transaction T3.

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Figure 16.6 Transaction T4.

UNIT-5

Granting of Locks: When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that

1. There is no other other transaction holding a lock on Q in a mode that conflicts with M.
2. There is no other transaction that is waiting for a lock on Q, and that made its lock request before Ti.

Thus, a lock request will never get blocked by a lock request that is made later

The Two-Phase Locking Protocol: One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

The Two-Phase Locking Protocol
The two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. Two-phase locking does not ensure freedom from deadlock.

Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

T_5	T_6	T_7
lock-x(A)		
read(A)		
lock-s(B)		
read(B)		
write(A)		
unlock(A)		
	lock-x(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-s(A)
		read(A)

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.

UNIT-5

Timestamp-Based Protocols: The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp-ordering scheme.

Timestamps

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).

There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj . To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.
- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues read(Q).
   - **a.** If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
   - **b.** If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).
2. Suppose that transaction Ti issues write(Q).
   - **a.** If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
   - **b.** If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.
   - **c.** Otherwise, the system executes the write operation and sets W-timestamp (Q) to TS(Ti).

If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Disadvantage of Time Stamp based Protocol:

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits
- Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction Ti to commit only after the commit of any transaction that wrote a value that Ti read. Commit dependencies can be used for this purpose.

Thomas' Write Rule

Let us consider **schedule 4 of Figure 16.14,** and apply the timestamp-ordering protocol. Since T16 starts before T17, we shall assume that TS(T16) < TS(T17). The read(Q) operation of T16 succeeds, as does the write(Q) operation of T17.When T16 attempts its write(Q) operation, we find that TS(T16) < W-timestamp(Q), since W-timestamp(Q) = TS(T17). Thus, the write(Q) by T16 is rejected and transaction T16 must be rolled back.

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Figure 16.14 Schedule 4.

Although the rollback of T16 is required by the timestamp-ordering protocol, it is unnecessary. Since T17 has already written Q, the value that T16 is attempting to write is one that will never need to be read. Any transaction Ti with TS(Ti) < TS(T17) that attempts a read(Q) will be rolled back, since TS(Ti) < W-timestamp(Q). Any transaction Tj with TS(Tj) > TS(T17) must read the value of Q written by T17, rather than the value written by T16.

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

The timestamp-ordering protocol requires that Ti be rolled back if Ti issues write(Q) and TS(Ti) < W-timestamp(Q). However, here, in those cases where TS(Ti) ≥ R-timestamp(Q), we ignore the obsolete write.

Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols.

For example, schedule 4 of Figure 16.14 is not conflict serializable and, thus, is not possible under any of two-phase locking, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T16 would be ignored. The result is a schedule that is view equivalent to the serial schedule <T16, T17>.

UNIT-5

Validation-Based Protocols: In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alterna tive scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system.

We assume that each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read phase**. During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. Transaction Ti performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write phase**. If transaction Ti succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back Ti.

Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved.
To perform the validation test, we need to know when the various phases of transactions Ti took place. We shall, therefore, associate three different timestamps with transaction Ti:

1. **Start (Ti),** the time when Ti started its execution.
2. **Validation (Ti),** the time when Ti finished its read phase and started its validation phase.
3. **Finish (Ti),** the time when Ti finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation (Ti). Thus, the value TS(Ti) = Validation (Ti) and, if TS(Tj) < TS(Tk), then any produced schedule must be equivalent to a serial schedule in which transaction Tj appears before transaction Tk. The reason we have chosen Validation (Ti), rather than Start (Ti), as the timestamp of transaction Ti is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The validation test for transaction Tj requires that, for all transactions Ti with TS(Ti) < TS(Tj), one of the following two conditions must hold:

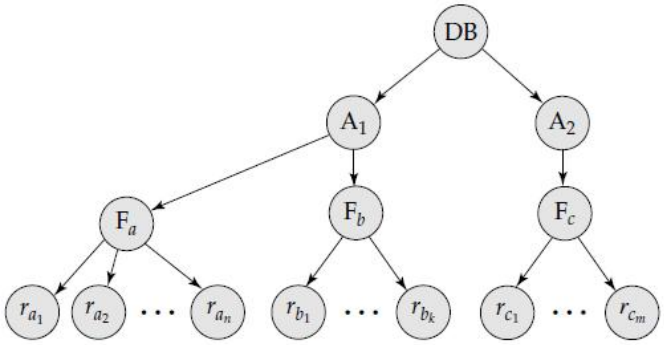
1. Finish (Ti) < Start (Tj). Since Ti completes its execution before Tj started, the serializability order is indeed maintained.
2. The set of data items written by Ti does not intersect with the set of data items read by Tj, and Ti completes its write phase before Tj starts its validation phase (Start(Tj) < Finish(Ti) < Validation(Tj)). This condition ensures that the writes of Ti and Tj do not overlap. Since the writes of Ti do not affect the read of Tj , and since Tj cannot affect the read of Ti, the serializability order is indeed maintained.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate*⟩	
display($A + B$)	
	⟨*validate*⟩
	write(B)
	write(A)

Figure 16.15: Schedule 5, a schedule produced by using validation.

This validation scheme is called the **optimistic concurrency control scheme** since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

Multiple Granularity:
In the concurrency-control schemes we have used each individual data item as the unit on which synchronization is performed.
There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit.
For example, if a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if Ti could issue a single lock request to lock the entire database. On the other hand, if transaction Tj needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.



Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.

For example, if transaction Ti gets an explicit lock on file Fc of Figure 16.16, in exclusive mode, then it has an implicit lock in exclusive mode all the records belonging to that file. It does not need to lock the individual records of Fc explicitly.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 16.17 Compatibility matrix.

The multiple-granularity locking scheme can be applied by introduce a new class of lock modes, called intention lock modes.
If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully.
There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure 16.17.

IS: Intention Shared Lock
IX: Intention Exclusice Lock
S: Shared Lock
SIX: Shared and Intention-Exclusive
X: Exclusice Lock

Multiple-Granularity Locking protocol (MGL)
The multiple-granularity locking protocol, which ensures serializability, is this:
Each transaction Ti can lock a node Q by following these rules:

1. It must observe the lock-compatibility function of Compatibility matrix
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.
4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, Ti is two phase).
6. It can unlock a node Q only if it currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in topdown (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

UNIT-5

Multiversion Schemes:

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In multiversion concurrency control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrency control manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

Multiversion Timestamp Ordering:

The most common transaction ordering technique used by multiversion schemes is time stamping. With each transaction Ti in the system, we associate a unique static timestamp, denoted by TS(Ti). The database system assigns this timestamp before the transaction starts execution.With each data item Q, a sequence of versions <Q1, Q2, . . .,Qm> is associated. Each version Qk contains three data fields:

- Content is the value of version Qk.
- W-timestamp(Qk) is the timestamp of the transaction that created version Qk.
- R-timestamp(Qk) is the largest timestamp of any transaction that successfully read version Qk.

A transaction—say, Ti—creates a new version Qk of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by Ti. The system initializes the W-timestamp and R-timestamp to TS(Ti). It updates the R-timestamp value of Qk whenever a transaction Tj reads the content of Qk, and R-timestamp(Qk) < TS(Tj).

The multiversion timestamp-ordering scheme presented next ensures serializability. The scheme operates as follows. Suppose that transaction Ti issues a read(Q) or write(Q) operation. Let Qk denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(Ti).

1. If transaction Ti issues a read(Q), then the value returned is the content of version Qk.
2. If transaction Ti issues write(Q), and if TS(Ti)<R-timestamp(Qk), then the system rolls back transaction Ti. On the other hand, if TS(Ti) = W-timestamp(Qk), the system overwrites the contents of Qk; otherwise it creates a new version of Q.

First rule stated that a transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is “too late” in doing a write. More precisely, if Ti attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed. Versions that are no longer needed are removed according.