CHAPTER - 10

TRANSACTION PROCESSING AND CONCURRENCY CONTROL

10.1 Introduction to Transactions

An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database is called transaction. A transaction is a logical unit of work on the database. During transaction execution the database may be inconsistent but when the transaction is committed, the database must be consistent.

There are two main issues to deal with:

i. Failures of various kinds, such as hardware failures and system crashes.

ii. Concurrent execution of multiple transactions.

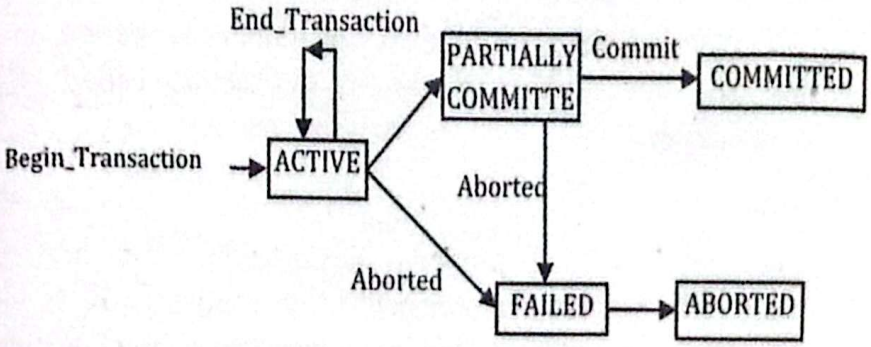


Figure: Transaction states

Transaction State

1. **Active**

   The initial state; the transaction stays in this state while it is executing.

2. **Partially committed**

   When a transaction executes its final operation it is said to be in a partially committed state.

3. **Failed**

Which occurs if the transaction can't be committed or transaction is aborted while in active state.

4. **Aborted**

If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after transaction aborts:

- restart the transaction – only if no internal logical error
- kill the transaction

5. **Committed**

If a transaction executes all its operations successfully it is said to be committed. All its effects are now permanently established on the database system

10.2 ACID Properties of Transaction

To preserve integrity of data, the database system must ensure four important properties of transaction called ACID properties.

1. **Atomicity**

The "all or nothing" property. A transaction is an invisible unit that is either performed in its entirety or is not performed at all. It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity. It states that all operations of the transaction take place at once if not the transaction is aborted.

There is no midway, i.e., transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all. Atomicity involves the following two operations:

i. **Abort**

If a transaction aborts, then all the changes made are not visible

ii. **Commit**

If a transaction commits, then all the changes made are visible

Example:

Let's assume that following transaction T consisting of T1 and T2. A consists of Rs600 and B consists of RS300. Transfer Rs 100 from account A to account B

T1	T2
Read(A)	Read(B)
A:A-100	B:B+100
Write(A)	Write(B)

After the completion of transaction, A consists of Rs 500 and B consists of Rs 400

If the transaction T fails after the completion of Transaction T1 but before completion of T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed entirety.

2. **Consistency**

The integrity constraints are maintained so that the database is consistent before and after the transaction. The execution of transaction will leave a database in either its prior stable state or a new stable state. The consistency property of the database states that every transaction sees a consistent database instance. The transaction is used to transform the database from one consistent state to another consistent state

Example:

The total amount must be maintained before or after the transaction.

Total before T occurs

600 + 300 = 900

Total after T occurs

500 + 400 = 900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur

3. **Isolation**

   It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

   In isolation, if the transaction T1 is being executed and using the data item X, then the data item can't be accessed by any other transaction until the transaction T1 ends. The concurrency control subsystem of the DBMS enforce the isolation property

4. **Durability**

   The durability property is used to indicate the performance of the database's inconsistent state. It states that the transaction made the permanent changes.

   The effects of a successfully completed(committed) transaction are permanently recorded in the database and must not be lost because of subsequent failure.

Concurrent Execution

Multiple transactions are allowed to run concurrently in the system in concurrent execution. The process of managing simultaneous operations on the database without having them interfere with one another is called concurrency control. The operations of transactions are interleaved to achieve concurrent execution.

Advantages of concurrent execution are:

i. Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk.

ii. Reduced average response time for transactions; short transactions need not wait behind long ones.

Concurrent control schemas

Concurrent control schemas are mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

10.3 Schedules and Serializability

10.3.1 Schedule

A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each for the individual transactions is called schedule. A schedule for a set of transactions must consists of all instructions of those transactions. A schedule must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have a commit instruction as the last statement. A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Types of schedule:

1. Serial schedule
2. Non serial schedule

- **Serial Schedule**

  Serial schedule is a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions. There is no interference between transactions

Example:

Schedule 1

T1	T2
Read(A) A:=A-50 Write(A) Read(B) B:=B+50 Write(B)	
	Read(A) Temp:=A*0.1 A:A-temp Write(A) Read(B) B:=B+temp Write(B)

Figure: A serial schedule where T1 is followed by T2

Schedule 2

T1	T2
	Read(A) Temp:=A*0.1 A:=A-temp Write(A) Read(B) B:=B+temp Write(B)
Read(A) A:=A-50 Write(A) Read(B) B:=B+50 Write(B)	

Figure: A serial schedule where T2 is followed by T1

- **Non serial schedule**

A schedule where the operations from a set of concurrent transactions are interleaved

Let T1 and T2 be the transactions defined previously. The following schedule is not serial schedule but it is equivalent to schedule 1. In schedules 1,2 and 3 the sum A+B is preserved.

Schedule 3

T1	T2
Read(A) A:=A-50 Write(A)	
	Read(A) Temp:=A*0.1 A:=A-temp Write(A)
Read(B) B:=B+50 Write(B)	
	Read(B) B:=B+temp Write(B)

Figure: A non-serial schedule equivalent to schedule 1

The following concurrent schedule does not preserve the value of A+B:

Schedule 4

T1	T2
Read(A) A:=A-50	
	Read(A) Temp:=A*0.1 A:A-temp Write(A) Read(B)
Write(A) Read(B) B:=B+50 Write(B)	
	B:B+temp Write(B)

Figure: Non-serial schedule that does not preserve A+B

10.3.2 Serializability

Serializability is a concurrency scheme where the concurrent transaction is equivalent to one that executes the transactions serially. A schedule is list of transactions. A schedule S of n transactions is serializable if it is equivalent to some serial schedule of same n transactions. Two schedules are called result equivalent if they produce the same final state of the database. A schedule s is serial if, for every transaction T, participating in the schedule. All the operations of T are executed consecutively in the schedule.

A schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to notions of:

- Conflict serializability
- View serializability

The main objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interference and produce a database state that could be produced by a serial execution

Conflict Serializability

Two actions Ai and Aj executed on the same data object by Ti and Tj conflicts if either of them is write operation. Conflict serializability defines two instructions of different transactions accessing the same data item to perform a read/write operation.

Let Ai and Aj are consecutive non conflicting actions that belong to different transactions, we can swap Ai and Aj without changing the result. If two transactions are both read operation then they are not in conflict.

If one transaction wants to perform a read operation and other transaction wants to perform a write operation then they are in conflict and cannot be swapped. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read value updated by each other.

If a schedule s can be transformed into schedule s' by a series of swaps of non-conflicting instructions we say that s and s' are conflict equivalent. We can say that a schedule s is conflict equivalent to a serial schedule Actions I_i and I_j of transactions T_i and T_j respectively conflict if and only if there exist some item Q accessed by both I_i and I_j and at least one of these instructions wrote Q.

I_i=read(Q), I_j=read(Q)	I_i and I_j don't conflict
I_i=read(Q), I_j=write(Q)	They conflict
I_i=write(Q), I_j=read(Q)	They conflict
I_i=write(Q), I_j=wirite(Q)	They conflict

Intuitively a conflict between I_i and I_j forces a(logical temporal) order between them. That is replacing their order will change the result. If I_i and I_j are consecutive in a schedule and

they do not conflict, their results would remain same even if they had been interchanged in the schedule

Example:

Schedule 5 can be transformed into schedule 6, a serial schedule where T2 follows T1 by series of swaps of non-conflicting instructions. Schedule 5 is conflict serializable.

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule 5

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Schedule 6

Figure: Transformation of non-serial schedule into serial schedule by swapping non-conflicting instructions.

Example of a schedule that is not conflict serializable.

T3	T4
Read(Q)	
	Write(Q)
Write(Q)	

Figure: Schedule that is not conflict serializable

We are not able to swap instructions in the above schedule to obtain either serial schedule <T3, T4> or the serial schedule <T4, T3>. Schedule s1 is conflict serializable schedule with s3 but not with s2.

S1

T1	T2
Read(A)	
	Write(A)
Read(B)	
	Write(B)

S2

T1	T2
Write(A)	
Write(B)	
	Read(A)
	Read(B)

S3

T1	T2
Read(A)	
Read(B)	
	Write(A)
	Write(B)

Figure: Schedule S1 is conflict serializable schedule with S3 but not with S2

Precedence graph for testing conflict serializability

Precedence graph or serialization graph is used commonly to test Conflict Serializability of a schedule. It is a directed Graph (V, E) consisting of a set of nodes V = {T1, T2, T3, ...Tn} and a set of directed edges E = {e1, e2, e3, ...em}. The graph contains one node for each Transaction T_i. An edge e_i is of the form T_j -> T_k where T_j is the starting node of e_i and T_k is the ending node of e_i. An edge e_i is constructed between nodes T_j to T_k if one of the operations in T_j appears in the schedule before some conflicting operation in T_k.

Algorithm

1. Create a node T in the graph for each participating transaction in the schedule.
2. For the conflicting operation read_item(X) and write_item(X) – If a Transaction T_j executes a read_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
3. For the conflicting operation write_item(X) and read_item(X) – If a Transaction T_j executes a write_item (X) after T_i executes a read_item (X), draw an edge from T_i to T_j in the graph.

4. For the conflicting operation write_item(X) and write_item(X)
   - If a Transaction T_i executes a write_item (X) after T_j executes a write_item (X), draw an edge from T_i to T_j in the graph.
5. The Schedule S is serializable if there is no cycle in the precedence graph.
   - If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S.
   - The serial schedule S' can be found by Topological Sorting of the acyclic precedence graph. Such schedules can be more than 1.

Example:

Consider the schedule S :

S : r1(x) r1(y) w2(x) w1(x) r2(y)

Creating tabular representation of above schedule.

T1	T2
r(x)	
r(y)	
	w(x)
w(x)	
	r(y)

graph

sponding to Transaction T1 and T2.

w2(x), where r1(x) happ
T1 to T2.

w2(x) hap

4. For the conflicting operation write_item(X) and write_item(X) – If a Transaction T_j executes a write_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
5. The Schedule S is serializable if there is no cycle in the precedence graph.
   - If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S.
   - The serial schedule S' can be found by Topological_Sorting of the acyclic precedence graph. Such schedules can be more than 1.

Example:

Consider the schedule S :

S : r1(x) r1(y) w2(x) w1(x) r2(y)

Creating tabular representation of above schedule.

T1	T2
r(x)	
r(y)	
	w(x)
w(x)	
	r(y)

Creating Precedence graph

i. Make two nodes corresponding to Transaction T1 and T2.

ii. For the conflicting pair r1(x) w2(x), where r1(x) happens before w2(x), draw an edge from T1 to T2.

iii. For the conflicting pair w2(x) w1(x), where w2(x) happens before w1(x), draw an edge from T2 to T1.

Since the graph is cyclic, we can conclude that it is not conflict serializable to any schedule serial schedule.

Let us try to infer a serial schedule from this graph using topological ordering.

The edge $T_1 \rightarrow T_2$ tells that T_1 should come before T_2 in the linear ordering.

The edge $T_2 \rightarrow T_1$ tells that T_2 should come before T_1 in the linear ordering.

So, we cannot predict any particular order (when the graph is cyclic). Therefore, no serial schedule can be obtained from this graph

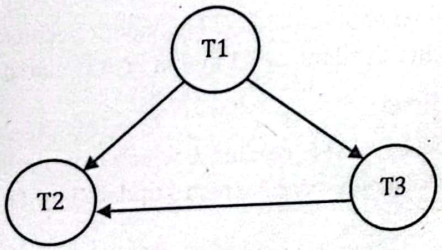
Consider the another schedule S1 :

S1: r1(x) r3(y) w1(x) w2(y) r3(x) w2(x)

Creating tabular representation of above schedule.

T1	T2	T3
r(x)		
		r(y)
w(x)		
	w(y)	
		r(x)
	w(x)	

The graph for above schedule is:



Since the graph is acyclic, the schedule is conflict serializable. Performing Topological Sort on this graph would give us a possible serial schedule that is conflict equivalent to schedule S1.

In Topological Sort, we first select the node with in-degree 0, which is T1. This would be followed by T3 and T2 So, S1 is conflict serializable since it is conflict equivalent to the serial schedule T1 T3 T2.

View Serializability

A schedule will be view serializable if it is view equivalent to a serial schedule. If a schedule is conflict serializable it will be view serializable. The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedule S1 and S2 are said to be view equivalent if they satisfy the following conditions.

1. **Initial Read**

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule 1

T1	T2
	Write(A)
Read(A)	

Schedule 2

Figure: Initial read

Above two schedules are view equivalent because initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. **Updated Read**

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also Ti should read A which s updated by Tj.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
		Write(A)
	Read(A)	
Write(A)		

Schedule S2

Figure: Updated read

Above two schedules are not view equivalent because in S1, T3 is reading A updated by T2. In S2, T3 is reading A updated by T1

3. **Final Write**

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final write operation should also be done by T1

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule S1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule S2

Figure: Final write

Above two schedules are view equivalent because final write operation in S1 is done by T3 and in S2, then final write operation is also done by T3.

Example:

Schedule S

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

With 3 transactions the total number of possible schedule is 3!=6.

S1=<T1, T2, T3>

S2=<T1, T3, T2>

S3=<T2, T3, T1>

S4=<T2, T1, T3>

S5=<T3, T1, T2>

S6=<T3, T2, T1>

Taking first schedule S1

T1	T2	T3
Read(A)		
Write(A)		
	Write(A)	
		Write(A)

Step 1: Updated Read

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

Hence, view equivalent serial schedule is:

T1 → T2 → T3

Differences between conflict serializability and view serializability:

- Conflict serializability is easy to achieve but view serializability is difficult to achieve.
- Every conflict serializable schedule is view serializable but the reverse is not true.
- It is easy to test conflict serializability but expensive to test view serializability.
- Most of the concurrency control schemas used in practice are conflict serializability.

10.4 Concepts of Locking for Concurrency Control

The management and coordination of many concurrent transactions that access and modify the same data simultaneously is referred to as concurrency control in DBMS. By controlling concurrency, transactions are prevented from interfering with one another or yielding inaccurate results, and the database is kept in a consistent state.

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

i. Shared lock
ii. Exclusive lock

i. Shared Lock

It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction. It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item. If a transaction T_i has obtained a shared lock on item Q, then T_i can read but cannot write Q.

ii. Exclusive Lock

In the exclusive lock, the data item can be both reads as well as written by the transaction. If a transaction T_i has obtained exclusive lock on item Q, then T_i can both read and write Q. Lock requests are made to concurrency control manager. Transaction can proceed only after request is granted.

Lock compatibility matrix

		Current State of Locking of Data Items		
		Unlocked	Shared(S)	Exclusive(X)
Lock Mode of Request	Unlocked		Yes	Yes
	Shared(S)	Yes	Yes	No
	Exclusive(X)	Yes	No	No

Figure: Lock compatibility matrix

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

10.4.1 Two-Phase Locking (2PL)

This is a protocol which ensures conflict serializable schedules. The two-phase locking protocol divides the execution phase of the transaction into three parts. In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires. In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock. In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

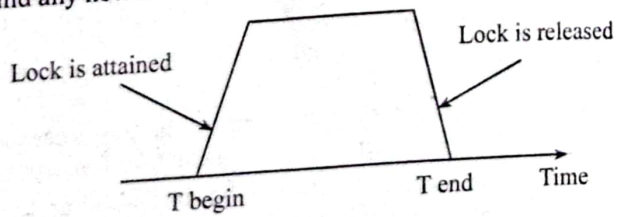


Figure: Two phase locking

There are two phases of 2PL:

i. Growing phase

ii. Shrinking phase

i. Growing Phase

In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

ii. Shrinking Phase

In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired. In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	T2
0	Lock-S(A)	
1		Lock-S(A)
2	Lock-X(B)	
3	-	-
4	Unblock(A)	
5		Lock-X(C)
6	Unblock(B)	
7		Unblock(A)
8		Unblock(C)
9	-	-

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

Growing phase

From step 0-2

Shrinking phase

From step 4-6

Lock point

At 3

Transaction T2:

Growing phase

From step 1-5

Shrinking phase

From step 7-8

Lock point

At 6

10.4.2 Strict Two-Phase Locking (Strict-2PL)

The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally. The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it. Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time. Strict-2PL protocol does not have shrinking phase of lock release. It does not have cascading abort as 2PL does.

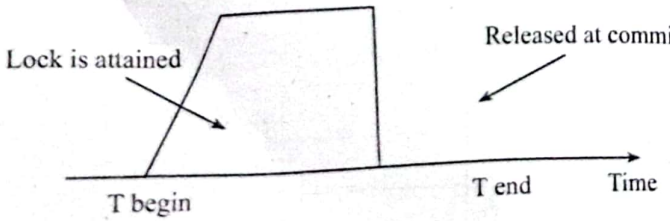


Figure: Strict two-phase locking

Lock conversion

Two phase locking with lock conversions.

First phase:

- Can acquire a Lock-S on item.
- Can acquire a Lock-x on item.
- Can convert a Lock-s to a Lock-X (upgrade)

Second phase

- Can release a Lock-S
- Can release a Lock-X
- Can convert a Lock-X to Lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Pitfalls of lock based protocol

Consider the partial schedule

T3	T4
lock-x(B)	
Read(B)	
B:=B-50	
Write(B)	
	Lock-S(A)
	Read(A)
	Lock-S(B)
Lock-X(A)	

Neither T3 nor T4 can make progress. Executing lock-S(B) causes T4 to wait for T3 to release its lock on B, while executing Lock-X(A) causes T3 to wait T4 to release its lock on A. Such situation is called a Deadlock.

To handle a deadlock one of T3 or T4 must be rolled back and its locks released. The potential for deadlock exists in most locking protocols. Deadlocks are necessary evil. Starvation is also possible if concurrency control manager is badly designed.

Example:

A transaction may be waiting for an X-Lock on an item, while a sequence of other transactions request and are granted on S-Lock on the same item. The same transaction is repeatedly rolled back due to deadlocks. Concurrency control manager can be designed to prevent starvation.

10.5 Deadlock

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a halt.

Example:

In the student table, Transaction T1 holds a lock on some rows in the Students table and needs to update some rows in the Grades table. Simultaneously, Transaction T2 holds locks on those very rows (Which T1 needs to update) in the Grades table but needs to update the rows in the Student table held by Transaction T1

Now, Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

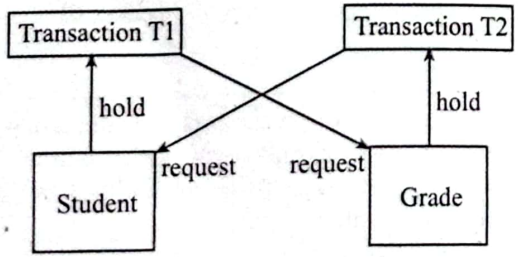


Figure: Deadlock

In multi process system deadlock is an unwanted situation that arises in a shared resource environment, where a process

indefinitely waits for a resource that is held by another process. System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set. Following schedule is a schedule with deadlock:

T1	T2
Lock-x on X	
Write(X)	
	Lock-x on Y
	Write(Y)
	Wait for Lock-x on X
	Write(X)
Wait for Lock-x on Y	
Write(Y)	

To prevent any deadlock situation in the system the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If they can create a deadlock situation might occur, then that transaction is never allowed to be executed. Deadlock prevention protocols ensure that the system will never enter into a deadlock state.

10.5.1 Deadlock Avoidance

When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restarting the database. This is a waste of resource. Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

10.5.2 Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is

involved in a deadlock or not. The lock manager maintains a wait graph to detect the deadlock cycle in the graph.

Wait for graph

This is suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop then there is a deadlock. The wait for graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph. The wait for graph for the above scenario is shown below:

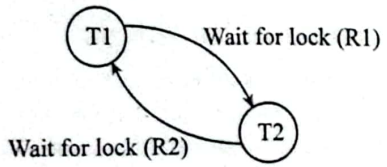


Figure: Wait for graph

Deadlocks can be described as a wait for graph which consists of a pair G=(V, E). V is a set of vertices (all the transactions in the system). E is a set of edges; each element is an ordered pair Ti -> Tj. If Ti -> Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item. When Ti requests a data item currently being held by Tj, then edge Ti->Tj is inserted in the wait for graph. This edge is removed only when Tj is no longer holding data item needed by Tj.

10.5.3 Deadlock Prevention

Deadlock prevention is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then deadlock can be prevented. The Database Management System analyzes the operations of the transactions whether they can create a deadlock situation or not. If they do, then the DBMS never allow that transaction to be executed. Two approach to deadlock prevention are:

1. One approach ensures that no cyclic waits can occur by ordering the requests for locks, requiring all locks to be acquired together.
2. Another approach performs transaction rolled back instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

The first approach requires that each transaction locks all its data items before execution in one step or none are locked. There are two main disadvantages:

1. It is often hard to predict, before the transaction begins, what data items need to be locked.
2. Data item utilization maybe very slow. Since many of the data items may be locked but unused for a long time

The second approach for preventing deadlocks is to use preemption and transaction rollback. These schemas use timestamps just for deadlock prevention

Wait-Die Schema

In this schema, if a transaction requests for a resource which is already held with a conflicting lock by another transaction, then the DBMS simply checks the timestamp of both transactions. It allows older transaction to wait until the resource is available for execution

When Transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than tha of Tj. Otherwise, Ti is rolled back (dies). Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

i. Check if TS(Ti)<TS(j)-If Ti is older transaction and Tj has held some resource, then Ti is allowed to wait until the data item is available for execution. This means if the older transaction is waiting for a resource which is locked by the younger transaction, then older transaction is allowed to wait for resource until its available.

ii. Check if TS(Ti)<TS(Tj)-If Ti is older transaction and has held some resource and if Tj is waiting for it then Tj is killed and restarted later with the random delay but with the same timestamp

Example:

Suppose that transaction T1, T2 and T3 have timestamps 20,30 and 40 respectively. If T1 requests a data item held by T2 then T1 will wait. If T3 requests a data item held by T2 then T3 will be rolled back.

Wound Wait Scheme

In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.

If the older transaction has held a resource which is requested by the younger transaction, then the younger transaction is asked to wait until older release it. When transaction Ti requests a data item currently held by Tj, Tj is allowed to wait only if it has timestamp larger than that of T otherwise Ti is rolled back.

Example:

Suppose that transaction T1, T2 and T3 have timestamps 20,30 and 40 respectively. If T1 requests a data item held by T2, then the data item will be preempted from T2 and T2 will be rolled back. If T3 requests a data item held by T2, then T3 will wait. Both in wait die and in wound wait schemas a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer one's ad starvation is hence avoided.

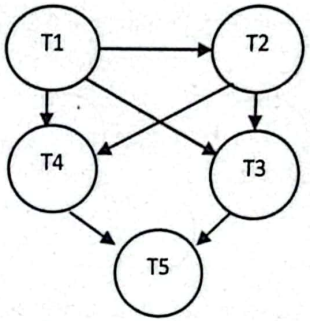
Timeout-Based schemas

A transaction waits for a lock only for specified amount of time. After that the wait times out and transaction is rolled back. Thus, deadlocks are not possible. Simple to implement; but starvation is possible. Also, difficult to determine good value of the timeout interval.

SOLUTION TO EXAMS' AND OTHER IMPORTANT QUESTIONS

1. Since every conflict serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability? Consider the precedence graph given below. Is the corresponding schedule conflict serializable? Explain your answer.

ANS:

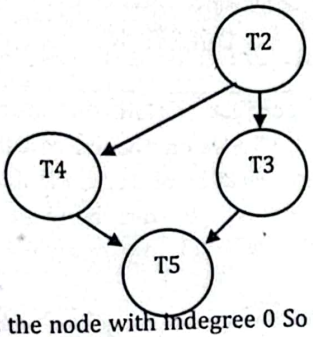


Since every conflict serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability because conflict-serializability needs in simple algorithms for its checking, while checking of view-serializability belongs to NP-complete problems. NP-complete problems are any of a class of computational problems for which no efficient solution algorithm has been found.

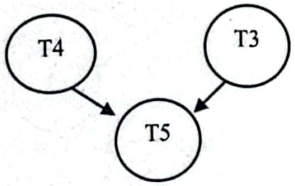
Since there are no efficient solution, we emphasize conflict serializability. Also, every conflict serializable schedule is view serializable we could only concentrate on view serializability if a schedule is not conflict serializable.

Since the graph is acyclic, the schedule is conflict serializable. Performing Topological Sort on this graph would give us a possible serial schedule that is conflict equivalent to given schedule .

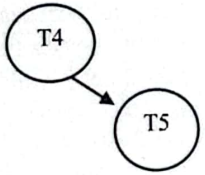
In Topological Sort, we first select the node with in-degree 0, which is T1. So T1 comes at first in order.



- Now T2 is the node with indegree 0 So T2 comes after T1 in order



- Now T4 and T3 are node with indegree 0 So T3 or T4 can come after T2 in order



- Now T4 is the node with indegree 0 So T4 can come after T3 in order

- Hence given schedule is conflict serializable since it is conflict equivalent to the serial schedule T1 T2 T3 T4 T5.

◆◆◆

CHAPTER - 11

ADVANCED DATABASE CONCEPTS

11.1 Object Oriented Model

This data model is a method of representing real world objects. Object oriented model considers each object in the world as objects and isolates it from each other. Object oriented model groups its related functionalities together and allows inheriting its functionality to others related sub groups.

Element of Object Oriented Model

- **Objects**

  The real world entities and situations are represented as objects in the object oriented database model.

- **Attributes and Method**

  Every object has certain characteristics. These are represented using attributes. The behavior of the object is represented using methods.

- **Class**

  Similar attributes and methods are grouped together using a class. An object can be called as an instance of the class.

- **Inheritance**

  A new class can be derived from the original class. The derived class contains attributes and methods of the original class as its own.

  Example:

  Shape, circle, Rectangle and Triangle are all objects in this model.

  Circle has the attributes center and radius. Rectangle has the attributes length and breadth. Triangle has the attributes base and height. The objects circle, rectangle and triangle inherit from the object shape

11.2 Object Relational Model

A data modeling and manipulation technique that connects object-oriented programming and conventional relational databases is called the Object-Relational Model (ORM).