UNIT 2 TRANSACTIONS AND CONCURRENCY MANAGEMENT

Structure **Page Nos.**

2.0 INTRODUCTION

One of the main advantages of storing data in an integrated repository or a database is to allow sharing of it among multiple users. Several users access the database or perform transactions at the same time. What if a user's transactions try to access a data item that is being used /modified by another transaction? This unit attempts to provide details on how concurrent transactions are executed under the control of DBMS. However, in order to explain the concurrent transactions, first we must describe the term transaction.

Concurrent execution of user programs is essential for better performance of DBMS, as concurrent running of several user programs keeps utilizing CPU time efficiently, since disk accesses are frequent and are relatively slow in case of DBMS. Also, a user's program may carry out many operations on the data returned from DB, but DBMS is only concerned about what data is being read /written from/ into the database. This unit discusses the issues of concurrent transactions in more detail.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the term CONCURRENCY;
- define the term transaction and concurrent transactions;
- discuss about concurrency control mechanism;
- describe the principles of locking and serialisability, and
- describe concepts of deadlock & its prevention.

2.2 THE TRANSACTIONS

A transaction is defined as the unit of work in a database system. Database systems that deal with a large number of transactions are also termed as transaction processing systems.

What is a transaction? Transaction is a unit of data processing. For example, some of the transactions at a bank may be withdrawal or deposit of money; transfer of money from A's account to B's account etc. A transaction would involve manipulation of one

or more data values in a database. Thus, it may require reading and writing of database value. For example, the withdrawal transactions can be written in pseudo code as:

Example 1:

```
; Assume that we are doing this transaction for person
;  whose account number is X.

TRANSACTION WITHDRAWAL (withdrawal_amount)
Begin transaction
        IF X exist then
                READ  X.balance
                IF        X.balance > withdrawal_amount
                          THEN  SUBTRACT withdrawal_amount
                                  WRITE X.balance
                                   COMMIT
                ELSE
                DISPLAY "TRANSACTION CANNOT BE PROCESSED"
        ELSE  DISPLAY "ACCOUNT X DOES NOT EXIST"
End transaction;
```

Another similar example may be transfer of money from Account no x to account number y. This transaction may be written as:

```
Example 2:
; transfers transfer_amount from x's account to y's account
; assumes x&y both accounts exist

TRANSACTION (x, y, transfer_amount)
Begin transaction
        IF X AND Y exist then
                READ x.balance
                IF x.balance > transfer_amount THEN
                          x.balance = x.balance – transfer_amount
                          READ y.balance
                          y.balance = y.balance + transfer_amount
                          COMMIT
                ELSE DISPLAY ("BALANCE IN X NOT OK")
                        ROLLBACK
        ELSE DISPLAY ("ACCOUNT X OR Y DOES NOT EXIST")
End_transaction
```

Please note the use of two keywords here COMMIT and ROLLBACK. Commit makes sure that all the changes made by transactions are made permanent. ROLLBACK terminates the transactions and rejects any change made by the transaction. Transactions have certain desirable properties. Let us look into those properties of a transaction.

Properties of a Transaction

A transaction has four basic properties. These are:

- Atomicity
- Consistency
- Isolation or Independence
- Durability or Permanence

Atomicity: It defines a transaction to be a single unit of processing. In other words either a transaction will be done *completely* or *not at all*. In the transaction example 1 & 2 please note that transaction 2 is reading and writing more than one data items, the atomicity property requires either operations on both the data item be performed or not at all.

Consistency: This property ensures that a complete transaction execution takes a database from one consistent state to another consistent state. If a transaction fails even then the database should come back to a consistent state.

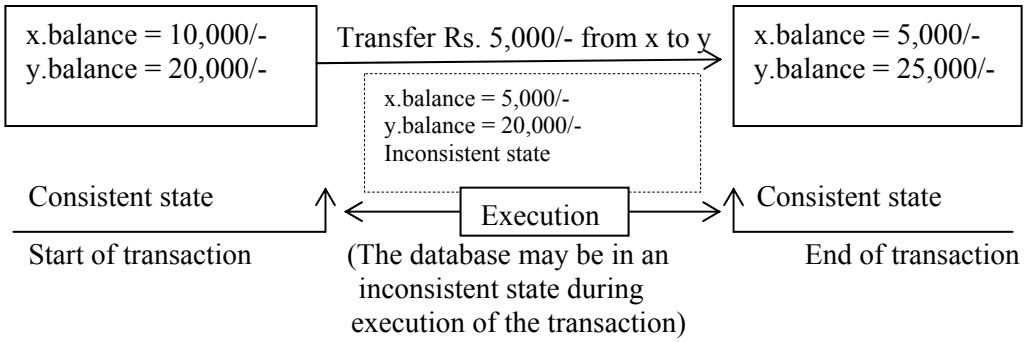


Figure 1: A Transaction execution

Isolation or Independence: The isolation property states that the updates of a transaction should not be visible till they are committed. Isolation guarantees that the progress of other transactions do not affect the outcome of this transaction. For example, if another transaction that is a withdrawal transaction which withdraws an amount of Rs. 5000 from X account is in progress, whether fails or commits, should not affect the outcome of this transaction. Only the state that has been read by the transaction last should determine the outcome of this transaction.

Durability: This property necessitates that once a transaction has committed, the changes made by it be never lost because of subsequent failure. Thus, a transaction is also a basic unit of recovery. The details of transaction-based recovery are discussed in the next unit.

A transaction has many states of execution. These states are displayed in *Figure 2*.

Error!

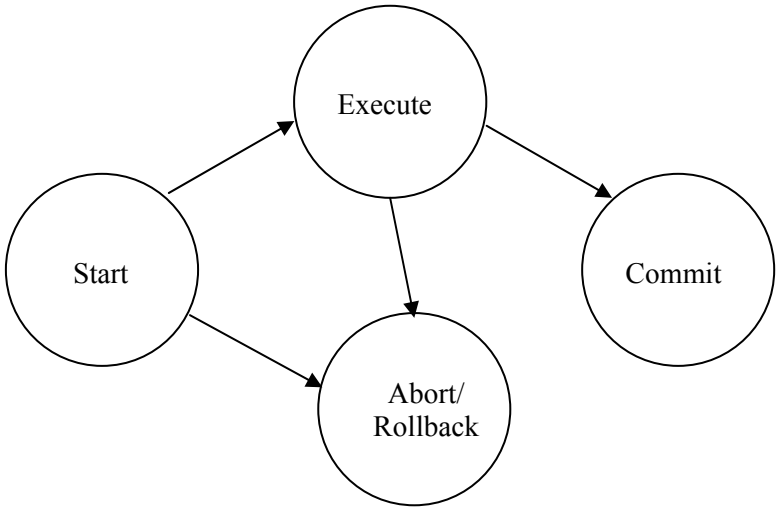


Figure 2: States of transaction execution

A transaction is started as a program. From the start state as the transaction is scheduled by the CPU it moves to the Execute state, however, in case of any system

error at that point it may also be moved into the Abort state. During the execution transaction changes the data values and database moves to an inconsistent state. On successful completion of transaction it moves to the Commit state where the durability feature of transaction ensures that the changes will not be lost. In case of any error the transaction goes to Rollback state where all the changes made by the transaction are undone. Thus, after commit or rollback database is back into consistent state. In case a transaction has been rolled back, it is started as a new transaction. All these states of the transaction are shown in *Figure 2*.

2.3 THE CONCURRENT TRANSACTIONS

Almost all the commercial DBMS support multi-user environment. Thus, allowing multiple transactions to proceed simultaneously. The DBMS must ensure that two or more transactions do not get into each other's way, i.e., transaction of one user doesn't effect the transaction of other or even the transactions issued by the same user should not get into the way of each other. Please note that concurrency related problem may occur in databases only if **two transactions are contending for the same data item and at least one of the concurrent transactions wishes to update a data value in the database**. In case, the concurrent transactions only read same data item and no updates are performed on these values, then it does NOT cause any concurrency related problem. Now, let us first discuss why you need a mechanism to control concurrency.

Consider a banking application dealing with checking and saving accounts. A Banking Transaction T1 for Mr. Sharma moves Rs.100 from his checking account balance X to his savings account balance Y, using the transaction T1:

Transaction T1:

```
A:  Read X
    Subtract 100
    Write X
B:  Read Y
    Add 100
    Write Y
```

Let us suppose an auditor wants to know the total assets of Mr. Sharma. He executes the following transaction:

Transaction T2:

```
Read X
Read Y
Display X+Y
```

Suppose both of these transactions are issued simultaneously, then the execution of these instructions can be mixed in many ways. This is also called the **Schedule**. Let us define this term in more detail.

A schedule S is defined as the sequential ordering of the operations of the 'n' interleaved transactions. A schedule maintains the order of operations within the individual transaction.

Conflicting Operations in Schedule: Two operations of different transactions conflict if they access the same data item AND one of them is a write operation.

For example, the two transactions TA and TB as given below, if executed in parallel, may produce a schedule:

TA	TB

READ X WRITE X

READ X WRITE X

SCHEDULE	TA	TB
READ X	READ X	
READ X		READ X
WRITE X		WRITE X
WRITE X	WRITE X	

One possible schedule for interleaved execution of TA and TB

Let us show you three simple ways of interleaved instruction execution of transactions T1 and T2. Please note that in the following tables the first column defines the sequence of instructions that are getting executed, that is the schedule of operations.

a) T2 is executed completely before T1 starts, then sum X+Y will show the correct assets:

Schedule	**Transaction T1**	**Transaction T2**	**Example Values**
Read X		Read X	X = 50000
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**150000**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100

b) T1 is executed completely before T2 starts, then sum X+Y will still show the correct assets:

Schedule	**Transaction T1**	**Transaction T2**	**Example Values**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100100
Display X+Y		Display X+Y	**150000**

c) Block A in transaction T1 is executed, followed by complete execution of T2, followed by the Block B of T1.

Schedule	**Transaction T1**	**Transaction T2**	**Example Values**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**149900**
Read Y	Read Y		Y= 100000

Add 100	Add 100		100100
Write Y	Write Y		Y= 100100

In this execution an incorrect value is being displayed. This is because Rs.100 although removed from X, has not been put in Y, and is thus missing. Obviously, if T1 had been written differently, starting with block B and following up with block A, even then such an interleaving would have given a different but incorrect result.

Please note that for the given transaction there are many more ways of this interleaved instruction execution.

Thus, there may be a problem when the transactions T1 and T2 are allowed to execute in parallel. Let us define the problems of concurrent execution of transaction more precisely.

Let us assume the following transactions (assuming there will not be errors in data while execution of transactions)

Transaction T3 and T4: T3 reads the balance of account X and subtracts a withdrawal amount of Rs. 5000, whereas T4 reads the balance of account X and adds an amount of Rs. 3000

T3
READ X
SUB 5000
WRITE X

T4
READ X
ADD 3000
WRITE X

Problems of Concurrent Transactions

1. **Lost Updates:** Suppose the two transactions T3 and T4 run concurrently and they happen to be interleaved in the following way (assume the initial value of X as 10000):

T3	T4	Value of X	
		T3	T4
READ X		10000	
	READ X		10000
SUB 5000		5000	
	ADD 3000		13000
WRITE X		5000	
	WRITE X		13000

After the execution of both the transactions the value X is 13000 while the semantically correct value should be 8000. The problem occurred as the update made by T3 has been overwritten by T4. The root cause of the problem was the fact that both the transactions had read the value of X as 10000. Thus one of the two updates has been lost and we say that a **lost update** has occurred.

There is one more way in which the lost updates can arise. Consider the following part of some transactions:

T5	T6	Value of x originally 2000	
		T5	T6
UPDATE X		3000	
	UPDATE X		4000
ROLLBACK		2000	

Here T5 & T6 updates the same item X. Thereafter T5 decides to undo its action and rolls back causing the value of X to go back to the original value that was 2000. In this case also the update performed by T6 had got lost and a lost update is said to have occurred.

2. **Unrepeatable reads:** Suppose T7 reads X twice during its execution. If it did not update X itself it could be very disturbing to see a different value of X in its next read. But this could occur if, between the two read operations, another transaction modifies X.

T7	T8	Assumed value of X=2000	
		T7	T8
READ X		2000	
	UPDATE X		3000
READ X		3000	

Thus, the inconsistent values are read and results of the transaction may be in error.

3. **Dirty Reads:** T10 reads a value which has been updated by T9. This update has not been committed and T9 aborts.

T9	T10	Value of x old value = 200	
		T9	T10
UPDATE X		500	
	READ X		500
ROLLBACK		200	?

Here T10 reads a value that has been updated by transaction T9 that has been aborted. Thus T10 has read a value that would never exist in the database and hence the problem. Here the problem is primarily of isolation of transaction.

4. **Inconsistent Analysis:** The problem as shown with transactions T1 and T2 where two transactions interleave to produce incorrect result during an analysis by Audit is the example of such a problem. This problem occurs when more than one data items are being used for analysis, while another transaction has modified some of those values and some are yet to be modified. Thus, an analysis transaction reads values from the inconsistent state of the database that results in inconsistent analysis.

Thus, we can conclude that the prime reason of problems of concurrent transactions is that a transaction reads an inconsistent state of the database that has been created by other transaction.

But how do we ensure that execution of two or more transactions have not resulted in a concurrency related problem?

Well one of the commonest techniques used for this purpose is to restrict access to data items that are being read or written by one transaction and is being written by another transaction. This technique is called locking. Let us discuss locking in more detail in the next section.

☞ Check Your Progress 1

1) What is a transaction? What are its properties? Can a transaction update more than on data values? Can a transaction write a value without reading it? Give an example of transaction.

……

……

……

2) What are the problems of concurrent transactions? Can these problems occur in transactions which do not read the same data values?

……

……

……

3) What is a Commit state? Can you rollback after the transaction commits?

……

……

……

2.4 THE LOCKING PROTOCOL

To control concurrency related problems we use locking. A lock is basically a variable that is associated with a data item in the database. A lock can be placed by a transaction on a shared resource that it desires to use. When this is done, the data item is available for the exclusive use for that transaction, i.e., other transactions are locked out of that data item. When a transaction that has locked a data item does not desire to use it any more, it should unlock the data item so that other transactions can use it. If a transaction tries to lock a data item already locked by some other transaction, it cannot do so and waits for the data item to be unlocked. The component of DBMS that controls and stores lock information is called the Lock Manager. The locking mechanism helps us to convert a schedule into a serialisable schedule. We had defined what a schedule is, but what is a serialisable schedule? Let us discuss about it in more detail:

2.4.1 Serialisable Schedules

If the operations of two transactions conflict with each other, how to determine that no concurrency related problems have occurred? For this, serialisability theory has been developed. Serialisability theory attempts to determine the **correctness** of the schedules. The rule of this theory is:

"A schedule S of n transactions is serialisable if it is equivalent to some **serial schedule** of the same 'n' transactions".

A serial schedule is a schedule in which either transaction T_1 is completely done before T_2 or transaction T_2 is completely done before T_1. For example, the following figure shows the two possible serial schedules of transactions T_1 & T_2.

Schedule A: T2 followed by T1			Schedule B: T1 followed by T2		
Schedule	**T1**	**T2**	**Schedule**	**T1**	**T2**
Read X		Read X	Read X	Read X	
Read Y		Read Y	Subtract 100	Subtract 100	
Display X+Y		Display X+Y	Write X	Write X	
Read X	Read X		Read Y	Read Y	
Subtract 100	Subtract 100		Add 100	Add 100	
Write X	Write X		Write Y	Write Y	
Read Y	Read Y		Read X		Read X
Add 100	Add 100		Read Y		Read Y
Write Y	Write Y		Display X+Y		Display X+Y

Figure 3: Serial Schedule of two transactions

Schedule C: An Interleaved Schedule		
Schedule	**T1**	**T2**
Read X	Read X	
Subtract 100	Subtract 100	
Read X		Read X

Write X	Write X	
Read Y		Read Y
Read Y	Read Y	
Add 100	Add 100	
Display X+Y		Display X+Y
Write Y	Write Y	

Figure 4: An Interleaved Schedule

Now, we have to figure out whether this interleaved schedule would be performing read and write in the same order as that of a serial schedule. If it does, then it is equivalent to a serial schedule, otherwise not. In case it is not equivalent to a serial schedule, then it may result in problems due to concurrent transactions.

Serialisability

Any schedule that produces the same results as a serial schedule is called a serialisable schedule. But how can a schedule be determined to be serialisable or not? In other words, other than giving values to various items in a schedule and checking if the results obtained are the same as those from a serial schedule, is there an algorithmic way of determining whether a schedule is serialisable or not?

The basis of the algorithm for serialisability is taken from the notion of a serial schedule. There are two possible serial schedules in case of two transactions (T1- T2 OR T2 - T1). Similarly, in case of three parallel transactions the number of possible serial schedules is 3!, that is, 6. These serial schedules can be:

T1-T2-T3	T1-T3-T2	T2-T1-T3
T2-T3-T1	T3-T1-T2	T3-T2-T1

Using the notion of precedence graph, an algorithm can be devised to determine whether an interleaved schedule is serialisbale or not. In this graph, the transactions of the schedule are represented as the nodes. This graph also has directed edges. An edge from the node representing transactions Ti to node Tj means that there exists a **conflicting operation** between Ti and Tj and Ti precedes Tj in some conflicting operations. It has been proved that a serialisable schedule is the one that contains no cycle in the graph.

Given a graph with no cycles in it, there must be a serial schedule corresponding to it.

The steps of constructing a precedence graph are:

1. Create a node for every transaction in the schedule.
2. Find the precedence relationships in conflicting operations. Conflicting operations are (read-write) or (write-read) or (write–write) on the same data item in two different transactions. But how to find them?

    2.1 For a transaction Ti which *reads* an item A, find a transaction Tj that *writes* A later in the schedule. If such a transaction is found, draw an edge from Ti to Tj.
    2.2 For a transaction Ti which has *written* an item A, find a transaction Tj later in the schedule that *reads* A. If such a transaction is found, draw an edge from Ti to Tj.
    2.3 For a transaction Ti which has *written* an item A, find a transaction Tj that *writes* A later than Ti. If such a transaction is found, draw an edge from Ti to Tj.
3. If there is any cycle in the graph, the schedule is not serialisable, otherwise, find the equivalent serial schedule of the transaction by traversing the transaction nodes starting with the node that has no input edge.

Let us use this algorithm to check whether the schedule as given in Figure 4 is Serialisable. *Figure 5* shows the required graph. Please note as per step 1, we draw the two nodes for T1 and T2. In the schedule given in *Figure 4*, please note that the transaction T2 reads data item X, which is subsequently written by T1, thus there is an edge from T2 to T1 (clause 2.1). Also, T2 reads data item Y, which is subsequently written by T1, thus there is an edge from T2 to T1 (clause 2.1). However, that edge already exists, so we do not need to redo it. Please note that there are no cycles in the graph, thus, the schedule given in *Figure 4* is serialisable. The equivalent serial schedule (as per step 3) would be T2 followed by T1.

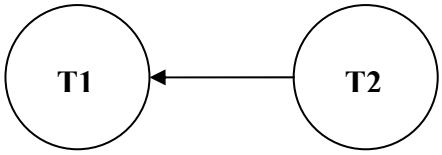


Figure 5: Test of Serialisability for the Schedule of Figure 4

Please note that the schedule given in part (c) of section 2.3 is not serialsable, because in that schedule, the two edges that exist between nodes T1 and T2 are:

- T1 writes X which is later read by T2 (clause 2.2), so there exists an edge from T1 to T2.
- T2 reads X which is later written by T1 (clause 2.1), so there exists an edge from T2 to T1.

Thus the graph for the schedule will be:

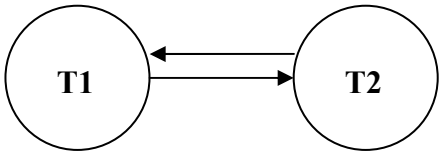


Figure 6: Test of Serialisability for the Schedule (c) of section 2.3

Please note that the graph above has a cycle T1-T2-T1, therefore it is not serialisable.

2.4.2 Locks

Serialisabilty is just a test whether a given interleaved schedule is ok or has a concurrency related problem. However, it does not ensure that the interleaved concurrent transactions do not have any concurrency related problem. This can be done by using locks. So let us discuss about what the different types of locks are, and then how locking ensures serialsability of executing transactions.

Types of Locks

There are two basic types of locks:

- Binary lock: This locking mechanism has two states for to a data item: locked or unlocked
- Multiple-mode locks: In this locking type each data item can be in three states read locked or shared locked, write locked or exclusive locked or unlocked.

Let us first take an example for binary locking and explain how it solves the concurrency related problems. Let us reconsider the transactions T1 and T2 for this purpose; however we will add to required binary locks to them.

Schedule	T1	T2
Lock X	Lock X	
Read X	Read X	
Subtract 100	Subtract 100	
Write X	Write X	
Unlock X	Unlock X	
Lock X		Lock X
Lock Y		Lock Y
Read X		Read X
Read Y		Read Y
Display X+Y		Display X+Y
Unlock X		Unlock X
Unlock Y		Unlock Y
Lock Y	Lock Y	
Read Y	Read Y	
Add 100	Add 100	
Write Y	Write Y	
Unlock Y	Unlock Y	

Figure 7: An incorrect locking implementation

Does the locking as done above solve the problem of concurrent transactions? No the same problems still remains. Try working with the old value. Thus, locking should be done with some logic in order to make sure that locking results in no concurrency related problem. One such solution is given below:

Schedule	T1	T2
Lock X	Lock X	
Lock Y	Lock Y	
Read X	Read X	
Subtract 100	Subtract 100	
Write X	Write X	
Lock X (issued by T2)	Lock X: denied as T1 holds the lock. The transaction T2 Waits and T1 continues.	
Read Y	Read Y	
Add 100	Add 100	
Write Y	Write Y	
Unlock X	Unlock X	
	The lock request of T2 on X can now be granted it can resumes by locking X	
Unlock Y	Unlock Y	
Lock Y		Lock Y
Read X		Read X
Read Y		Read Y
Display X+Y		Display X+Y
Unlock X		Unlock X
Unlock Y		Unlock Y

Figure 8: A correct but restrictive locking implementation

Thus, the locking as above when you obtain all the locks at the beginning of the transaction and release them at the end ensures that transactions are executed with no concurrency related problems. However, such a scheme limits the concurrency. We will discuss a two-phase locking method in the next subsection that provides sufficient concurrency. However, let us first discuss multiple mode locks.

Multiple-mode locks: It offers two locks: shared locks and exclusive locks. But why do we need these two locks? There are many transactions in the database system that never update the data values. These transactions can coexist with other transactions that update the database. In such a situation multiple reads are allowed on a data item, so multiple transactions can lock a data item in the shared or read lock. On the other hand, if a transaction is an updating transaction, that is, it updates the data items, it has to ensure that no other transaction can access (read or write) those data items that it wants to update. In this case, the transaction places an exclusive lock on the data items. Thus, a somewhat higher level of concurrency can be achieved in comparison to the binary locking scheme.

The properties of shared and exclusive locks are summarised below:

a) **Shared lock or Read Lock**

- It is requested by a transaction that wants to just read the value of data item.
- A shared lock on a data item does not allow an exclusive lock to be placed but permits any number of shared locks to be placed on that item.

b) **Exclusive lock**

- It is requested by a transaction on a data item that it needs to update.
- No other transaction can place either a shared lock or an exclusive lock on a data item that has been locked in an exclusive mode.

Let us describe the above two modes with the help of an example. We will once again consider the transactions T1 and T2 but in addition a transaction T11 that finds the total of accounts Y and Z.

Schedule	T1	T2	T11
S_Lock X		S_Lock X	
S_Lock Y		S_Lock Y	
Read X		Read X	
S_Lock Y			S_Lock Y
S_Lock Z			S_Lock Z
			Read Y
			Read Z
X_Lock X	X_Lock X. The exclusive lock request on X is denied as T2 holds the Read lock. The transaction T1 Waits.		
Read Y		Read Y	
Display X+Y		Display X+Y	
Unlock X		Unlock X	
X_Lock Y	X_Lock Y. The previous exclusive lock request on X is granted as X is unlocked. But the new exclusive lock request on Y is not granted as Y is locked by T2 and T11 in read mode. Thus T1 waits till both T2 and T11 will release the read lock on Y.		
Display Y+Z			Display Y+Z
Unlock Y		Unlock Y	
Unlock Y			Unlock Y
Unlock Z			Unlock Z
Read X	Read X		
Subtract 100	Subtract 100		
Write X	Write X		
Read Y	Read Y		
Add 100	Add 100		

Write Y	Write Y		
Unlock X	Unlock X		
Unlock Y	Unlock Y		

Figure 9: Example of Locking in multiple-modes

Thus, the locking as above results in a serialisable schedule. Now the question is can we release locks a bit early and still have no concurrency related problem? Yes, we can do it if we lock using two-phase locking protocol. This protocol is explained in the next sub-section.

2.4.3 Two Phase Locking (2PL)

The two-phase locking protocol consists of two phases:

Phase 1: The lock acquisition phase: If a transaction T wants to read an object, it needs to obtain the S (shared) lock. If T wants to modify an object, it needs to obtain X (exclusive) lock. No conflicting locks are granted to a transaction. **New locks on items can be acquired but no lock can be released till all the locks required by the transaction are obtained.**

Phase 2: Lock Release Phase: The existing locks can be released in any order but no new lock can be acquired **after a lock has been released**. The locks are held only till they are required.

Normally the locks are obtained by the DBMS. Any legal schedule of transactions that follows 2 phase locking protocol is guaranteed to be serialisable. The two phase locking protocol has been proved for it correctness. However, the proof of this protocol is beyond the scope of this Unit. You can refer to further readings for more details on this protocol.

There are two types of 2PL:

(1) The Basic 2PL
(2) Strict 2PL

The basic 2PL allows release of lock at any time after all the locks have been acquired. For example, we can release the locks in schedule of *Figure 8*, after we have Read the values of Y and Z in transaction 11, even before the display of the sum. This will enhance the concurrency level. The basic 2PL is shown graphically in *Figure 10.*

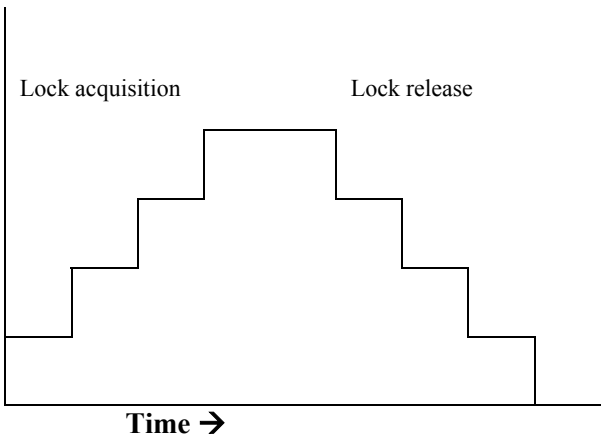


Figure 10: Basic Two Phase Locking

However, this basic 2PL suffers from the problem that it can result into loss of atomic / isolation property of transaction as theoretically speaking once a lock is released on a

data item it can be modified by another transaction before the first transaction commits or aborts.

To avoid such a situation we use strict 2PL. The strict 2PL is graphically depicted in *Figure 11*. However, the basic disadvantage of strict 2PL is that it restricts concurrency as it locks the item beyond the time it is needed by a transaction.

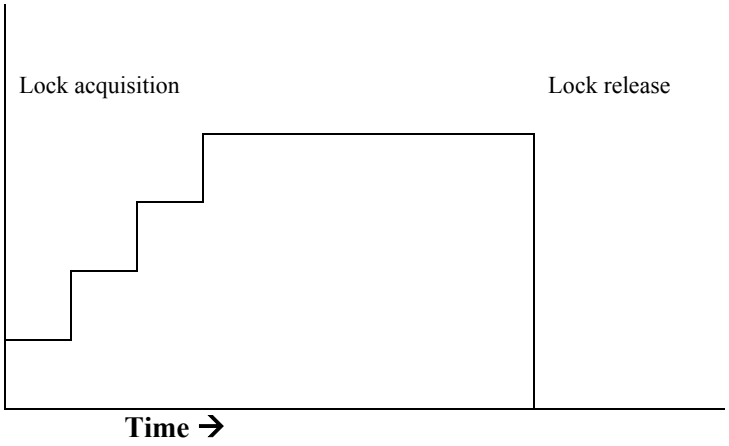


Figure 11: Strict Two Phase Locking

Does the 2PL solve all the problems of concurrent transactions? No, the strict 2PL solves the problem of concurrency and atomicity, however it introduces another problem: "Deadlock". Let us discuss this problem in next section.

☞ Check Your Progress 2

1) Let the transactions T1, T2, T3 be defined to perform the following operations:

   T1: Add one to A
   T2: Double A
   T3: Display A on the screen and set A to one.

   Suppose transactions T1, T2, T3 are allowed to execute concurrently. If A has initial value zero, how many possible correct results are there? Enumerate them.

   ……

   ……

2) Consider the following two transactions, given two bank accounts having a balance A and B.

   Transaction T1: Transfer Rs. 100 from A to B

   Transaction T2: Find the multiple of A and B.

   Create a non-serialisable schedule.

   ……

   ……

3) Add lock and unlock instructions (exclusive or shared) to transactions T1 and T2 so that they observe the serialisable schedule. Make a valid schedule.

   ……

   ……

2.5 DEADLOCK AND ITS PREVENTION

As seen earlier, though 2PL protocol handles the problem of serialisability, but it causes some problems also. For example, consider the following two transactions and a schedule involving these transactions:

TA	TB
X_lock A	X_lock A
X_lock B	X_lock B
:	:
:	:
Unlock A	Unlock A
Unlock B	Unlock B

Schedule

T1: X_lock A
T2: X_lock B
T1: X_lock B
T2: X_lock A

As is clearly seen, the schedule causes a problem. After T1 has locked A, T2 locks B and then T1 tries to lock B, but unable to do so waits for T2 to unlock B. Similarly, T2 tries to lock A but finds that it is held by T1 which has not yet unlocked it and thus waits for T1 to unlock A. At this stage, neither T1 nor T2 can proceed since both of these transactions are waiting for the other to unlock the locked resource.

Clearly the schedule comes to a halt in its execution. The important thing to be seen here is that both T1 and T2 follow the 2PL, which guarantees serialisability. So whenever the above type of situation arises, we say that a deadlock has occurred, since two transactions are **waiting for a condition that will never occur**.

Also, the deadlock can be described in terms of a directed graph called a "*wait for*" graph, which is maintained by the lock manager of the DBMS. This graph G is defined by the pair (V, E). It consists of a set of vertices/nodes V is and a set of edges/arcs E. Each transaction is represented by node and an arc from Ti → Tj, if Tj holds a lock and Ti is waiting for it. When transaction Ti requests a data item currently being held by transaction Tj then the edge Ti → Tj is inserted in the "wait for" graph. This edge is removed only when transaction Tj_j is no longer holding the data item needed by transaction Ti.

A deadlock in the system of transactions occurs, if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, a periodic check for cycles in graph can be done. For example, the "wait-for" for the schedule of transactions TA and TB as above can be made as:

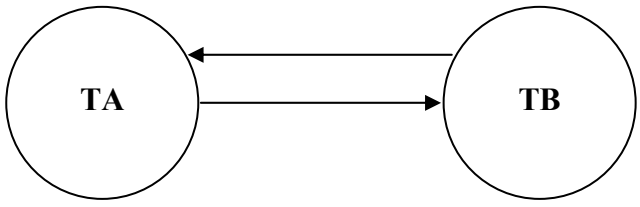


Figure 12: Wait For graph of TA and TB

In the figure above, TA and TB are the two transactions. The two edges are present between nodes TA and TB since each is waiting for the other to unlock a resource held by the other, forming a cycle, causing a deadlock problem. The above case shows a direct cycle. However, in actual situation more than two nodes may be there in a cycle.

A deadlock is thus a situation that can be created because of locks. It causes transactions to wait forever and hence the name deadlock. A deadlock occurs because of the following conditions:

a) Mutual exclusion: A resource can be locked in exclusive mode by only one transaction at a time.
b) Non-preemptive locking: A data item can only be unlocked by the transaction that locked it. No other transaction can unlock it.
c) Partial allocation: A transaction can acquire locks on database in a piecemeal fashion.
d) Circular waiting: Transactions lock part of data resources needed and then wait indefinitely to lock the resource currently locked by other transactions.

In order to prevent a deadlock, one has to ensure that at least one of these conditions does not occur.

A deadlock can be prevented, avoided or controlled. Let us discuss a simple method for deadlock prevention.

Deadlock Prevention

One of the simplest approaches for avoiding a deadlock would be to acquire all the locks at the start of the transaction. However, this approach restricts concurrency greatly, also you may lock some of the items that are not updated by that transaction (the transaction may have if conditions).Thus, better prevention algorithm have been evolved to prevent a deadlock having the basic logic: *not to allow circular wait to occur*. This approach rolls back some of the transactions instead of letting them wait.

There exist two such schemes. These are:

"Wait-die" scheme: The scheme is based on non-preventive technique. It is based on a simple rule:

```
If Ti requests a database resource that is held by Tj
        then if Ti has a smaller timestamp than that of Tj
                it is allowed to wait;
        else Ti aborts.
```

A timestamp may loosely be defined as the system generated sequence number that is unique for each transaction. Thus, a smaller timestamp means an older transaction. For example, assume that three transactions T1, T2 and T3 were generated in that sequence, then if T1requests for a data item which is currently held by transaction T2, it is allowed to wait as it has a smaller time stamping than that of T1. However, if T3 requests for a data item which is currently held by transaction T2, then T3 is rolled back (die).

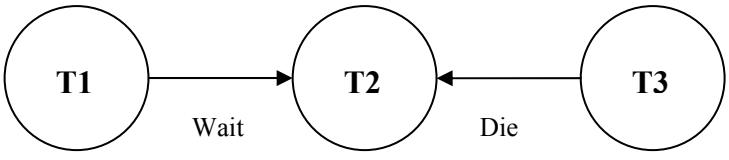


Figure 13: Wait-die Scheme of Deadlock prevention

"Wound-wait" scheme: It is based on a preemptive technique. It is based on a simple rule:

If Ti requests a database resource that is held by Tj
then if Ti has a larger timestamp (Ti is younger) than that of Tj
it is allowed to wait;
else Tj is wounded up by Ti.

For example, assume that three transactions T1, T2 and T3 were generated in that sequence, then if T1requests for a data item which is currently held by transaction T2, then T2 is rolled back and data item is allotted to T1 as T1 has a smaller time stamping than that of T2. However, if T3 requests for a data item which is currently held by transaction T2, then T3 is allowed to wait.

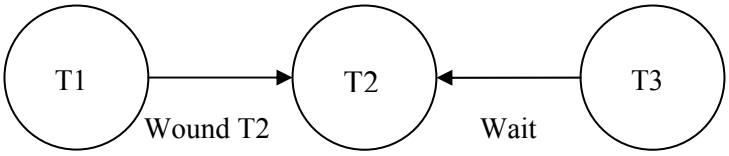


Figure 14: Wound-wait Scheme of Deadlock prevention

It is important to see that whenever any transaction is rolled back, it would not make a starvation condition, that is no transaction gets rolled back repeatedly and is never allowed to make progress. Also both "wait-die" & "wound-wait" scheme avoid starvation. The number of aborts & rollbacks will be higher in wait-die scheme than in the wound-wait scheme. But one major problem with both of these schemes is that these schemes may result in unnecessary rollbacks. You can refer to further readings for more details on deadlock related schemes.

2.6 OPTIMISTIC CONCURRENCY CONTROL

Is locking the only way to prevent concurrency related problems? There exist some other methods too. One such method is called an Optimistic Concurrency control. Let us discuss it in more detail in this section.

The basic logic in optimistic concurrency control is to allow the concurrent transactions to update the data items assuming that the concurrency related problem will not occur. However, we need to reconfirm our view in the validation phase. Therefore, the optimistic concurrency control algorithm has the following phases:

a) READ Phase: A transaction T reads the data items from the database into its private workspace. All the updates of the transaction can only change the local copies of the data in the private workspace.

b) VALIDATE Phase: Checking is performed to confirm whether the read values have changed during the time transaction was updating the local values. This is performed by comparing the current database values to the values that were read in the private workspace. In case, the values have changed the local copies are thrown away and the transaction aborts.

c) WRITE Phase: If validation phase is successful the transaction is committed and updates are applied to the database, otherwise the transaction is rolled back.

Some of the terms defined to explain optimistic concurrency contents are:

- write-set(T): all data items that are written by a transaction T
- read-set(T): all data items that are read by a transaction T

- Timestamps: for each transaction T, the start-time and the end time are kept for all the three phases.

More details on this scheme are available in the further readings. But let us show this scheme here with the help of the following examples:

Consider the set for transaction T1 and T2.

T1		T2	
Phase	**Operation**	**Phase**	**Operation**
-	-	Read	Reads the read set (T2). Let say variables X and Y and performs updating of local values
Read	Reads the read set (T1) lets say variable X and Y and performs updating of local values	-	-
Validate	Validate the values of (T1)	-	-
-	-	Validate	Validate the values of (T2)
Write	Write the updated values in the database and commit	-	-
-	-	Write	Write the updated values in the database and commit

In this example both T1 and T2 get committed. Please note that Read set of T1 and Read Set of T2 are both disjoint, also the Write sets are also disjoint and thus no concurrency related problem can occur.

T1	T2	T3
Operation	Operation	Operation
Read R(A)	--	--
--	Read R(A)	--
--	--	Read (D)
--	--	Update(D)
--	--	Update (A)
--	--	Validate (D,A) finds OK Write (D,A), COMMIT
--	Validate(A):Unsuccessful Value changed by T3	--
Validate(A):Unsuccessful Value changed by T3	--	--
ABORT T1	--	--
--	Abort T2	--

In this case both T1 and T2 get aborted as they fail during validate phase while only T3 is committed. Optimistic concurrency control performs its checking at the transaction commits point in a validation phase. The serialization order is determined by the time of transaction validation phase.

☞ Check Your Progress 3

1) Draw suitable graph for following locking requests, find whether the transactions are deadlocked or not.

T1: S_lock A	--	--
--	T2: X_lock B	--
--	T2: S_lock C	--

--	--	T3: X_lock C
--	T2: S_lock A	--
T1: S_lock B	--	--
T1: S_lock A	--	--
--	--	T3: S_lock A
All the unlocking requests start from here		

……

……

……

……

……

……

……

……

2) What is Optimistic Concurrency Control?

……

……

……

……

……

…….

2.7 SUMMARY

In this unit you have gone through the concepts of transaction and Concurrency Management. A transaction is a sequence of many actions. Concurrency control deals with ensuring that two or more users do not get into each other's way, i.e., updates of transaction one doesn't affect the updates of other transactions.

Serializability is the generally accepted criterion for correctness for the concurrency control. It is a concept related to concurrent schedules. It determines how to analyse whether any schedule is serialisable or not. Any schedule that produces the same results as a serial schedule is a serialisable schedule.

Concurrency Control is usually done via locking. If a transaction tries to lock a resource already locked by some other transaction, it cannot do so and waits for the resource to be unlocked.

Locks are of two type a) shared lock b) Exclusive lock. Then we move on to a method known as Two Phase Locking (2PL). A system is in a deadlock state if there exist a set of transactions such that every transaction in the set is waiting for another transaction in the set. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state.

Finally we have discussed the method Optimistic Concurrency Control, another concurrency management mechanism.

2.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) A transaction is the basic unit of work on a Database management system. It defines the data processing on the database. IT has four basic properties:

   a. Atomicity: transaction is done completely or not at all.
   b. Consistency: Leaves the database in a consistent state
   c. Isolation: Should not see uncommitted values
   d. Durability: Once committed the changes should be reflected.

   A transaction can update more than one data values. Some transactions can do writing of data without reading a data value.

   A simple transaction example may be: Updating the stock inventory of an item that has been issued. Please create a sample pseudo code for it.

2) The basic problems of concurrent transactions are:

   - Lost updates: An update is overwritten
   - Unrepeatable read: On reading a value later again an inconsistent value is found.
   - Dirty read: Reading an uncommitted value
   - Inconsistent analysis: Due to reading partially updated value.

   No these problems cannot occur if the transactions do not read the same data values. The conflict occurs only if one transaction updates a data value while another is reading or writing the data value.

3) Commit state is defined as when transaction has done everything correctly and shows the intent of making all the changes as permanent. No, you cannot rollback after commit.

Check Your Progress 2

1) There are six possible results, corresponding to six possible serial schedules:

Initially:	A = 0
T1-T2-T3:	A = 1
T1-T3-T2:	A = 2
T2-T1-T3:	A = 1
T2-T3-T1:	A = 2
T3-T1-T2:	A = 4
T3-T2-T1:	A = 3

2)

Schedule	T1	T2
Read A	Read A	
A = A - 100	A = A - 100	
Write A	Write A	
Read A		Read A
Read B		Read B

Read B	Read B	
Result = A * B		Result = A * B
Display Result		Display Result
B = B + 100	B = B + 100	
Write B	Write B	

Please make the precedence graph and find out that the schedule is not serialisable.

3)

Schedule	**T1**	**T2**
Lock A	Lock A	
Lock B	Lock B	
Read A	Read A	
A = A - 100	A = A - 100	
Write A	Write A	
Unlock A	Unlock A	
Lock A		Lock A: Granted
Lock B		Lock B: Waits
Read B	Read B	
B = B + 100	B = B + 100	
Write B	Write B	
Unlock B	Unlock B	
Read A		Read A
Read B		Read B
Result = A * B		Result = A * B
Display Result		Display Result
Unlock A		Unlock A
Unlock B		Unlock B

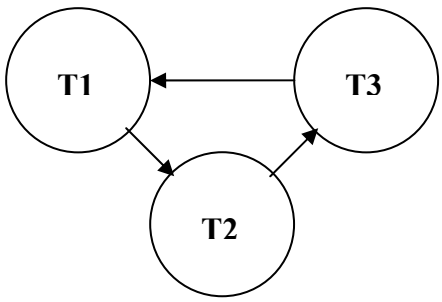
You must make the schedules using read and exclusive lock and a schedule in strict 2PL.

Check Your Progress 3

1) The transaction T1 gets the shared lock on A, T2 gets exclusive lock on B and Shared lock on A, while the transactions T3 gets exclusive lock on C.

- Now T2 requests for shared lock on C which is exclusively locked by T3, so cannot be granted. So T2 waits for T3 on item C.
- T1 now requests for Shared lock on B which is exclusively locked by T2, thus, it waits for T2 for item B. The T1 request for shared lock on C is not processed.
- Next T3 requests for exclusive lock on A which is share locked by T1, so it cannot be granted. Thus, T3 waits for T1 for item A.

The Wait for graph for the transactions for the given schedule is:



Since there exists a cycle, therefore, the schedule is deadlocked.

2) The basic philosophy for optimistic concurrency control is the optimism that nothing will go wrong so let the transaction interleave in any fashion, but to avoid any concurrency related problem we just validate our assumption before we make changes permanent. This is a good model for situations having a low rate of transactions.