UNIT -5

UNIT-5
Coping with System Failures & Concurrency Control

Coping with System Failures

Issues and Models for Resilient Operation

A computer system, like any other mechanical or electrical device is subject to failure. There are many causes of such failure, such as disk crash, power failure, software error, etc. In each of these cases, information may be lost. Therefore, the database system maintains an integral part known as recovery manager. It is responsible for the restore of the database to a consistent state that existed prior to the occurrence of the failures.

The recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability. It ensures atomicity by undoing the actions of transactions, that do not commit and durability by making sure that all actions of committed transactions survive system crashes and media failures.

When a DBMNS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction.

System Failure classifications:

1) Transaction Failure:

There are two types of errors that may cause a transaction failure.

i) **Logical Error:** The transaction can do longer continue with its normal execution with some internal conditions such as bad input, data not found, overflow or resource limits exceeded.

ii) **System Error:** The system has entered an undesirable state (deadlock) as a result of which a transaction cannot continue with its normal execution. This transaction can be reexecuted at a later time.

2) System Crash:

There is a hardware failure or an error in the database software or the operating system that causes the loss of the content of temporary storage and brings transaction processing to a halt. The content of permanent storage remains same and is not corrupted.

3) Disk failure:

A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks or backups on tapes are used to recover from the failure.

Causes of failures:

Some failures might cause the database to go down, some others might be trivial. On the other hand, if a data file has been lost, recovery requires additional steps. Some common causes of failures include:

1) ***System Crashes:***

It can be happen due to hardware or software errors resulting in loss of main memory.

2) ***User error:***

It can be happen due to a user inadvertently deleting a row or dropping a table.

3) ***Carelessness:***

It can be happen due to the destruction of data or facilities by operators/users because of lack of concentration.

4) ***Sabotage:***

It can be happen due to the intentional corruption or destruction of data, hardware or software facilities.

5) ***Statement failure:***

It can be happen due to the inability by the database to execute an SQL statement.

6) ***Application software errors:***

It can be happen due to the logical errors in the program to access the database, which causes one or more transactions to fail.

7) ***Network failure:***

It can be happen due to a network failure / communication software failure / aborted asynchronous connections.

8) ***Media failure:***

It can be happen due to the disk controller failure / disk head crash / disk to be lost. It is ht most dangerous failure.

9) ***Natural physical disasters****:*

It can be happen due to the natural disasters like fires, floods, earthquakes, power failure, etc.

Undo Logging

Logging is a way to assure that transactions are atomic. They appear to the database either to have executed in their entirety or not to have executed at all. A ***log*** is a sequence of ***log records***, each telling something about what some transaction has done. The actions of several transactions can "interleave," so that a step of one transaction may be executed and its effect logged, then the same happens for a step of another transaction, then for a second step of the first transaction or a step of a third transaction, and so on. This interleaving of transactions complicates logging; it is not sufficient simply to log the entire story of a transaction after that transaction completes.

If there is a system crash, the log is consulted to reconstruct what transactions were doing when the crash occurred. The log also may be used, in conjunction with an archive, if there is a media failure of a disk that does not store the log. Generally, to repair the effect of the crash, some transactions will have their work done again, and the new values they wrote into the database are written again. Other transactions will have their work undone, and the database restored so that it appears that they never executed.

The first style of logging, which is called **undo logging**, makes only repairs of the second type. If it is not absolutely certain that the effects of a transaction have been completed and stored on disk, then any database changes that the transaction may have made to the database are undone, and the database state is restored to what existed prior to the transaction.

Log Records

The log is a file opened for appending only. As transactions execute, the log manager has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as is feasible.

There are several forms of log record that are used with each of the types of logging. These are:

1. <START T> : This record indicates that transaction T has begun.

2. <COMMIT T>: Transaction T has completed successfully and will make no more changes to database elements. Any changes to the database made by T should appear on disk. If we insist that the changes already be on disk, this requirement must be enforced by the log manager.

3. <ABORT T> : Transaction T could not complete successfully. If transaction T aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is cancelled if they do.

The Undo-Logging Rules

There are two rules that transactions must obey in order that an undo log allows us to recover from a system failure. These rules affect what the buffer manager can do and also require that certain actions be taken whenever a transaction commits.

U_1 : If transaction *T* modifies database element *X*, then the log record of the form *<T, X,v>* must be written to disk *before* the new value of *X* is written to disk.

U_2 : If a transaction commits, then its COMMIT log record must be written to disk only *after* all

database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules U_1 and U_2, material associated with one transaction must be written to disk in the following order:

a) The log records indicating changed database elements.
b) The changed database elements themselves.
c) The COMMIT log record.

In order to force log records to disk, the log manager needs a flush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied.
Example:

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Actions and their log entries

The transaction of undo logging to show the log entries and flushlog actions that have to take place along with the actions of the transaction *T.*

Recovery Using Undo Logging

It is the job of the **recovery manager** to use the log to restore the database state to some consistent state. The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions.

If there is a log record <COMMIT T>, then by undo rule f/2 all changes made by transaction T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

If there is a log <START T> record on the log but no <COMMIT T> record. Then there could have been some changes to the database made by T that got written to disk before the crash, while other changes by T either were not made, even in the main-memory buffers, or were made in the buffers but not copied to disk.

After making all the changes, the recovery manager must write a log record < ABORT T> for each incomplete transaction T that was not previously aborted, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

Redo Logging

While undo logging provides a natural and simple strategy for maintaining a log and recovering from a system failure, it is not the only possible approach.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging.*

The principal *differences* between *redo* and *undo logging* are:

1. While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions

2. While undo logging requires us to write changed database elements to disk before the COMMIT log record reaches disk, redo logging requires that the COMMIT record appear on disk before any changed values reach disk.

3. While the old values of changed database elements are exactly what we need to recover when the undo rules U1 and U2 are followed, to recover using redo logging, we need the new values.

The Redo-Logging Rule

Redo logging represents changes to database elements by a log record that gives the new value, rather than the old value, which undo logging uses. These records look the same as for undo logging: *<T, X, v>*. The difference is that the meaning of this record is "transaction *T* wrote new value *v* for database element *X."*

There is no indication of the old value of *X* in this record. Every time a transaction *T* modifies a database element *X*, a record of the form <*T, X, v*> must be written to the log. The order in which data and log entries reach disk can be described by a single "redo rule," called the *write-ahead logging rule.*

R_1 : Before modifying any database element *X* on disk, it is necessary that all log records pertaining to this modification of *X*, including both the update record <*T, X, v*> and the <COMMIT *T*> record, must appear on disk.

The order of redo logging associated with one transaction gets written to disk is:

1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							< START T>
2)	READ(A,t)	8	8		8	8	
3)	t : = t * 2	16	8		8	8	
4)	WRITE(A,t)		16	16		8	8 <T,A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	t : = t * 2	16	16	8	8	8	
7)	WRITE(B,t)		16	16	16	8	8 <T,B.IQ>
8)					<COMMIT T>		
9)	FLUSH LOG						
10)	OUTPUT (A)		16	16	16	16	8
11)	OUTPUT(B)		16	16	16	16	16

Actions and their log entries using redo logging

Recovery with Redo Logging:

An important consequence of the redo lule *RI* is that unless the log has a <COMMIT T> record, we know that no changes to the database made by transaction T have been written to disk.

To recover, using a redo log after a system crash, we do the following:

1. Identify the committed transactions.
2. Scan the log forward from the beginning. For each log record <T,X,v> encountered:

(a) If T is not a committed transaction, do nothing.

(b) If T is committed, write value v for database element X.

3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

The steps to be taken to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record <START CKPT (T},..., T_k)>, where T_1,...,T_k are all the active (uncommitted) transactions, and flush the log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.

3. Write an <END CKPT> record to the log and flush the log.

<STAR T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,5,10>
<START CKPT (T_2)>
<T_2,c7,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

Recovery with a Check pointed Redo Log:

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END.

i) Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. <START CKPT (T_1,..., *Tk)*> or that started after that log record appeared in the log. In searching the log, we do not look further back than the earliest of the < START T,> records. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.

ii) The last checkpoint record on the log is a <START CKPT *(T_1,... ,Tk)*> record. We must search back to the previous <END CKPT> record, find its matching <START CKPT *(S_i,..., Srn)*> record, and redo all those committed transactions that either started after that START CKPT or are among the *S_1's*.

Undo/Redo Logging

We have two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

i) Undo logging requires that data be written to disk immediately after a transaction finishes.

ii) Redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed.

iii) Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks.

To overcome these drawbacks we have a kind of logging called ***undo/redo logging***, that provides increased flexibility to order actions, at the expense of maintaining more information on the log.

The Undo/Redo Rules:

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record <T, X, v,w> means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR_1 : Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T,X,v,w> appear on disk.

Rule UR_1 for undo/redo logging thus enforces only the constraints enforced by both undo logging and redo logging. In particular, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8, 16$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

A possible sequence of actions and their log entries using undo/redo logging.

Recovery with Undo/Redo Logging:

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T, by restoring the old values of the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the incomplete transactions in the order latest-first.

It is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which COMMIT log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

Check pointing an Undo/Redo Log:

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a <START CKPT *(T_1,...,Tk)*> record to the log, where T_1...,*Tk* are all the active transactions, and flush the log.

2. Write to disk all the buffers that are *dirty;* i.e., they contain one or more changed database elements. Unlike redo logging, we flush all buffers, not just those written by committed transactions.

3. Write an <END CKPT> record to the log, and flush the log.

<START T_1>
<T_1, 4, 4, 5>
< START T_2>
<COMMIT T_i >
<T_2, B, 9, 10>
<START CKPT (T_2)>
<T_2, C, 14, 15>
< START T_3>
<T_3, D, 19, 20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

An undo/redo log

Suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T2 as committed but T_3 as incomplete. We redo T2 by setting *C* to 15 on disk; it is not necessary to set *B* to 10 since we know that change reached disk before the <END CKPT>.

However, unlike the situation with a redo log, we also undo T_3*;* that is, we set *D* to 19 on disk. If T3 had been active at the start of the checkpoint, we would have had to look prior to the START-CKPT record to find if there were more actions by T_3 that may have reached disk and need to be undone.

Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. The serious failures involve the loss of one or more disks. We can reconstruct the database from the log if:

a) The log were on a disk other than the disk(s) that hold the data,

b) The log were never thrown away after a checkpoint, and

c) The log were of the redo or the undo/redo type, so new values are stored on the log.

The log will usually grow faster than the database. So it is not practical to keep the log forever.

The Archive:

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk, and store them remote from the database in some secure location.

The backup would preserve the database state as it existed at this time, and if there were a media failure, the database could be restored to the state that existed then.

Since writing an archive is a lengthy process if the database is large, one generally tries to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

1. A *full dump,* in which the entire database is copied.

2. An *incremental dump,* in which only those database elements changed, the previous full of incremental dump is copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level *i"* dump copying everything changed since the last dump at level *i* or less.

We can restore the database from a full dump and its subsequent incremental dumps, in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps. Since incremental dumps will tend to involve only a small fraction of the data changed since the last dump, they take less space and can be done faster than full dumps.

Nonquiescent Archiving:

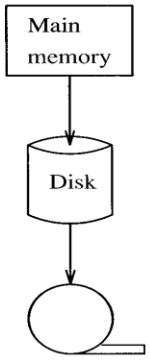
A nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started.

A nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state.

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.

Disk	Archive
	Copy A
`A := 5`	
	Copy B
`C := 6`	
	Copy C
`B := 7`	
	Copy D

Events during a nonquiescent dump



Checkpoint gets data from memory to disk; log allows recovery from system failure

Dump gets data from disk to archive; archive plus log allows recovery from media failure

The analogy between checkpoints and dumps

The process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving.

1. Write a log record < START DUMP>.
2. Perform a checkpoint appropriate for whichever logging method is being used.
3. Perform a full or incremental dump of the data disk(s), as desired; making sure that the copy of the data has reached the secure, remote site.

4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item (2) will survive a

media failure of the database.

5. Write a log record <END DUMP>.

At the completion of the dump, it is safe to throw away log prior to the beginning of the checkpoint previous to the one performed in item (2) above.

<START DUMP>
<START CKPT (T_1, T_2)>
<T_1, A, l, 5>
<T_2, C, 3, 6>
<COMMIT T_2>
<T_1, B, 2, 7>
<END CKPT>
Dump completes
<END DUMP>

Log taken during a dump

Note that we did not show T_1 committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method.

Recovery Using an Archive and Log:

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive.
   (a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
   (b) If there are later incremental dumps, modify the database according to each, earliest first.

2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

PART – A [2 mark questions]

1) Transaction Management: The two principal tasks of the transaction manager are assuring recoverability of database actions through logging, and assuring correct, concurrent behavior of transactions through the scheduler (not discussed in this chapter).

2) *Database Elements:* The database is divided into elements, which are typically disk blocks, but could be tuples, extents of a class, or many other units. Database elements are the units for both logging and scheduling.

3) Logging: A record of every important action of a transaction — beginning, changing a database element, committing, or aborting — is stored on a log. The log must be backed up on disk at a time that is related to when the corresponding database changes migrate to disk, but that time depends on the particular logging method used.

4) *Recovery:* When a system crash occurs, the log is used to repair the database, restoring it to a consistent state.

5) *Logging Methods:* The three principal methods for logging are undo, redo, and undo/redo, named for the way(s) that they are allowed to fix the database during recovery.

6) Undo Logging : This method logs only the old value, each time a database element is changed. With undo logging, a new value of a database element can only be written to disk after the log record for the change has reached disk, but before the commit record for the transaction performing the change reaches disk. Recovery is done by restoring the old value for every uncommitted transaction.

7) Redo Logging : Here, only the new value of database elements is logged. With this form of logging, values of a database element can only be written to disk after both the log record of its change and the commit record for its transaction have reached disk. Recovery involves rewriting the new value for every committed transaction.

8) Undo/Redo Logging: In this method, both old and new values are logged. Undo/redo logging is more flexible than the other methods, since it requires only that the log record of a change appear on the disk before the change itself does. There is no requirement about when the commit record appears. Recovery is effected by redoing committed transactions and undoing the uncommitted transactions.

9) Check pointing: Since all methods require, in principle, looking at the entire log from the dawn of history when a recovery is necessary, the DBMS must occasionally checkpoint the log, to assure that no log records prior to the checkpoint will be needed during a recovery. Thus, old log records can eventually be thrown away and its disk space reused.

10) Nonquiescent Check pointing : To avoid shutting down the system while a checkpoint is made, techniques associated with each logging method allow the checkpoint to be made while the system is in operation and database changes are occurring. The only cost is that some log records prior to the nonquiescent checkpoint may need to be examined during recovery.

11) Archiving: While logging protects against system failures involving only the loss of main memory, archiving is necessary to protect against failures where the contents of disk are lost. Archives are copies of the database stored in a safe place.

12) Recovery from Media Failures: When a disk is lost, it may be restored by starting with a full backup of the database, modifying it according to any later incremental backups, and finally recovering to a consistent database state by using an archived copy of the log.

13) Incremental Backups : Instead of copying the entire database to an archive periodically, a single complete backup can be followed by several incremental backups, where only the changed data is copied to the archive.

14) Nonqmescent Archiving : Techniques for making a backup of the data while the database is in operation exist. They involve making log records of the beginning and end of the archiving, as well as performing a checkpoint for the log during the archiving.

Serializability

Serializability is a widely accepted standard that ensures the consistency of a schedule. A schedule is consistent if and only if it is serializable. A schedule is said to be serializable if the interleaved transactions produces the result, which is equivalent to the result produced by executing individual transactions separately.

Example:

Transaction T_1	**Transaction T_2**		**Transaction T_1**	**Transaction T_2**
read(X)			read(X)	
write(X)			write(X)	
read(Y)				read(X)
write(Y)				write(X)
	read(X)		read(Y)	
	write(X)		write(Y)	
	read(Y)			read(Y)
	write(Y)			write(Y)

Serial Schedule ***Two interleaved transaction Schedule***

The above two schedules produce the same result, these schedules are said to be serializable. The transaction may be interleaved in any order and DBMS doesn't provide any guarantee about the order in which they are executed.

There two different types of Serializability. They are,

i) Conflict Serializability
ii) View Serializability

i) Conflict Serializability:

Consider a schedule S_1, consisting of two successive instructions I_A and I_B belonging to transactions T_A and T_B refer to different data items then it is very easy to swap these instructions.

The result of swapping these instructions doesn't have any impact on the remaining instructions in the schedule. If I_a and I_B refers to same data item then the following four cases must be considered,

Case 1 : I_A = read(x), I_B = read(x),
Case 2 : I_A = read(x), I_B = write(x),
Case 3 : I_A = write(x), I_B = read(x),
Case 4 : I_A = write(x), I_B = write(x),

Case 1 : Here, both I_A and I_B are read instructions. In this case, the execution order of the instructions is not considered since the same data item x is read by both the transactions T_A and T_B.

Case 2 : Here, I_A and I_B are read and write instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_A cannot read the value written by transaction T_B in instruction I_B. but order is $I_B \rightarrow I_A$, then transaction T_A can read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

Case 3 : Here, I_A and I_B are write and read instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_B can read the value written by transaction T_A, but order is $I_B \rightarrow I_A$, then transaction T_B cannot read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

Case 1 : Here, both I_A and I_B are write instructions. In this case, the execution order of the instructions doesn't matter. If a read operation is performed before the write operation, then the data item which was already stored in the database is read.

ii) View Serializability:

Two schedules S_1 and S_1**'** consisting of some set of transactions are said to be view equivalent, if the following conditions are satisfied,

1) If a transaction T_A in schedule S_1 performs the read operation on the initial value of data item x, then the same transaction in schedule S_1**'** must also perform the read operation on the initial value of x.

2) If a transaction T_A in schedule S_1 reads the value x, which was written by transaction T_B, then T_A in schedule S_1**'** must also perform the read the value x written by transaction T_B.

3) If a transaction T_A in schedule S_1 performs the final write operation on data item x, then the same transaction in schedule S_1**'** must also perform the final write operation on x.

Example:

Transaction T_1	**Transaction T_2**
read(x) x := x -10 write(x) read(y) y := y -10 write(y)	 read(x) x := x *10 write(x) read(y) y := y / 10 write(y)

View Serializability Schedule S_1

The view equivalence leads to another notion called view serializability. A schedule say S is said to be view Serializable, if it is view equivalent with the serial schedule.

Every conflict Serializable schedule is view Serializable but every view Serializable is not conflict Serializable.

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.

Why DBMS needs a concurrency control?

In general, concurrency control is an essential part of TM. It is a mechanism for correctness when two or more database transactions that access the same data or data set are executed

concurrently with time overlap. According to Wikipedia.org, if multiple transactions are executed serially or sequentially, data is consistent in a database. However, if concurrent transactions with interleaving operations are executed, some unexpected data and inconsistent result may occur. Data interference is usually caused by a write operation among transactions on the same set of data in DBMS. For example, the lost update problem may occur when a second transaction writes a second value of data content on top of the first value written by a first concurrent transaction. Other problems such as the dirty read problem, the incorrect summary problem

Concurrency Control Techniques:

The following techniques are the various concurrency control techniques. They are:

1. concurrency control by Locks
2. Concurrency Control by Timestamps
3. Concurrency Control by Validation

1. **Concurrency control by Locks**

- A lock is nothing but a mechanism that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose.
- There are two types of operations, i.e. read and write, whose basic nature are different, the locks for read and write operation may behave differently.
- The simple rule for locking can be derived from here. If a transaction is reading the content of a sharable data item, then any number of other processes can be allowed to read the content of the same data item. But if any transaction is writing into a sharable data item, then no other transaction will be allowed to read or write that same data item.
- Depending upon the rules we have found, we can classify the locks into two types.

Shared Lock: A transaction may acquire shared lock on a data item in order to read its content. The lock is shared in the sense that any other transaction can acquire the shared lock on that same data item for reading purpose.

Exclusive Lock: A transaction may acquire exclusive lock on a data item in order to both read/write into it. The lock is excusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item.

The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

Two Phase Locking Protocol

The use of locks has helped us to create neat and clean concurrent schedule. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks.

The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

Growing Phase:

- In this phase the transaction can only acquire locks, but cannot release any lock
- The transaction enters the growing phase as soon as it acquires the first lock it wants.
- It cannot release any lock at this phase even if it has finished working with a locked data item.
- Ultimately the transaction reaches a point where all the lock it may need has been acquired. This point is called **Lock Point**.

Shrinking Phase:

- After Lock Point has been reached, the transaction enters the shrinking phase. In this phase the transaction can only release locks, but cannot acquire any new lock.
- The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point.

Two Phase Locking Protocol:

- There are two different versions of the Two Phase Locking Protocol. They are:

1. Strict Two Phase Locking Protocol
2. Rigorous Two Phase Locking Protocol
   - In this protocol, a transaction may release all the shared locks after the Lock Point has been reached, but it cannot release any of the exclusive locks until the transaction commits. This protocol helps in creating cascade less schedule.

A **Cascading Schedule** is a typical problem faced while creating concurrent schedule. Consider the following schedule once again.

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
Unlock (A)	
	Lock-S (A)
	Read A;
	Temp = A * 0.1;
	Unlock (A)
	Lock-X(C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	
Unlock (B)	

- The schedule is theoretically correct, but a very strange kind of problem may arise here.
- T1 releases the exclusive lock on A, and immediately after that the Context Switch is made.
- T2 acquires a shared lock on A to read its value, perform a calculation, update the content of account C and then issue COMMIT. However, T1 is not finished yet. What if the remaining portion of T1 encounters a problem (power failure, disc failure etc) and cannot be committed?

- In that case T1 should be rolled back and the old BFIM value of A should be restored. In such a case T2, which has read the updated (but not committed) value of A and calculated the value of C based on this value, must also have to be rolled back.
- We have to rollback T2 for no fault of T2 itself, but because we proceeded with T2 depending on a value which has not yet been committed. This phenomenon of rolling back a child transaction if the parent transaction is rolled back is called Cascading Rollback, which causes a tremendous loss of processing power and execution time.
- Using Strict Two Phase Locking Protocol, Cascading Rollback can be prevented.
- In Strict Two Phase Locking Protocol a transaction cannot release any of its acquired exclusive locks until the transaction commits.
- In such a case, T1 would not release the exclusive lock on A until it finally commits, which makes it impossible for T2 to acquire the shared lock on A at a time when A's value has not been committed. This makes it impossible for a schedule to be cascading.

Rigorous Two Phase Locking Protocol

In Rigorous Two Phase Locking Protocol, a transaction is not allowed to release any lock (either shared or exclusive) until it commits. This means that until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock; but cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock.

2. Concurrency Control by Timestamps

Timestamp ordering technique is a method that determines the serializability order of different transactions in a schedule. This can be determined by having prior knowledge about the order in which the transactions are executed.

Timestamp denoted by $T_S(T_A)$ is an identifier that specifies the start time of transaction and is generated by DBMS. It uniquely identifies the transaction in a schedule. The timestamp of older transaction (T_A) is less than the timestamp of a newly entered transaction (T_B) i.e., $T_S(T_A) < T_S(T_B)$.

In timestamp-based concurrency control method, transactions are executed based on priorities that are assigned based on their age. If an instruction I_A of transaction T_A conflicts with an instruction I_B of transaction T_B then it can be said that I_A is executed before I_B if and only if $T_S(T_A) < T_S(T_B)$ which implies that older transactions have higher priority in case of conflicts.

Ways of generating Timestamps:
Timestamps can be generated by using,

i) System Clock : When a transaction enters the system, then it is assigned a timestamp which is equal to the time in the system clock.
ii) Logical Counter: When a transaction enters the system, then it is assigned a timestamp which is equal to the counter value that is incremented each time for a newly entered transaction.
Every individual data item x consists of the following two timestamp values,

i) WTS(x) (W-Timestamp(x)) : It represents the highest timestamp value of the transaction that successfully executed the *write* instruction on x.

ii) RTS(x) (R-Timestamp(x)) : It represents the highest timestamp value of the transaction that successfully executed the *read* instruction on x.

Timestamp Ordering Protocol

This protocol guarantees that the execution of read and write operations that are conflicting is done in timestamp order.

Working of Timestamp Ordering Protocol:

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

1) **If TA executes read(x) instruction**, then the following two cases must be considered,
   i) $T_S(T_A) < WTS(x)$
   ii) $T_S(T_A)$ □□WTS(x)

Case 1 : If a transaction T_A wants to read the initial value of some data item x that had been overwritten by some younger transaction then, the transaction T_A cannot perform the read operation and therefore the transaction must be rejected. Then the transaction T_A must be rolled back and restarted with a new timestamp.

Case 2 : If a transaction T_A wants to read the initial value of some data item x that had not been updated then the transaction can execute the read operation. Once the value has been read, changes occur in the read timestamp value (RTS(x)) which is set to the largest value of RTS(x) and T_S

2) If TA executes write(x) instruction, then the following two cases must be considered,

i) $T_S(T_A) < RTS(x)$

ii) $T_S(T_A) < WTS(x)$

iii) $T_S(T_A) > \square WTS(x)$

Case 1 : If a transaction T_A wants to write the value of some data item x on which the read operation has been performed by some younger transaction, then the transaction cannot execute the write operation. This is because the value of data item x that is being generated by T_A was required previously and therefore, the system assumes that the value will never be generated. The write operation is thereby rejected and the transaction T_A must be rolled back and should be restarted with new timestamp value.

Case 2 : If a transaction T_A wants to write a new value to some data item x, that was overwritten by some younger transaction, then the transaction cannot execute the write operation as it may lead to inconsistency of data item. Therefore, the write operation is rejected and the transaction should be rolled back with a new timestamp value.

Case 3 : If a transaction T_A wants to write a new value on some data item x that was not updated by a younger transaction, then the transaction can executed the write operation. Once the value has been written, changes occur on WTS(x) value which is set to the value of $T_S(T_A)$.

Example:

T_1	**T_2**
read(y)	
	read(y)
	y:= y + 100
	write(y)
read(x)	
	read(x)
show(x+y)	
	x:= x – 100
	write(x)
	show(x+y)

The above schedule can be executed under the timestamp protocol when $T_S(T_1) < T_S(T_2)$.

3. Concurrency Control by Validation

- Validation techniques are also called as Optimistic techniques.
- If read only transactions are executed without employing any of the concurrency control mechanisms, then the result generated is in inconsistent state.
- However if concurrency control schemes are used then the execution of transactions may be delayed and overhead may be resulted. To avoid such issue, optimistic concurrency control mechanism is used that reduces the execution overhead.
- But the problem in reducing the overhead is that, prior knowledge regarding the conflicting transactions will not be known. Therefore, a mechanism called "monitoring" the system is required to gain such knowledge.

Let us consider that every transaction T_A is executed in two or three-phases during its life-time. The phases involved in optimistic concurrency control are,

1) Read Phase
2) Validation Phase and
3) Write Phase

1) Read phase: In this phase, the copies of the data items (their values) are stored in local variables and the modifications are made to these local variables and the actual values are not modified in this phase.

2) Validation Phase: This phase follows the read phase where the assurance of the serializability is checked upon each update. If the conflicts occur between the transaction, then it is aborted and restarted else it is committed.

3) Write Phase : The successful completion of the validation phase leads to the write phase in which all the changes are made to the original copy of data items. This phase is applicable only to the read-write transaction. Each transaction is assigned three timestamps as follows,

i) When execution is initiated I(T)
ii) At the start of the validation phase V(T)
iii) At the end of the validation phase E(T)

Qualifying conditions for successful validation:

Consider two transactions, transaction T_A, transaction T_B and let the timestamp of transaction T_A is less than the timestamp of transaction T_B i.e., $T_S(T_A) < T_S(T_B)$ then,

1) Before the start of transaction T_B, transaction T_A must complete its execution. i.e., $E(T_A) < I(T_B)$

2) The values written by transaction T_A must not be necessarily matched with the values read by transaction T_B. T_A must execute the write phase before T_B initiate the execution of validation phase, i.e., $I(T_B) < E(T_A) < V(T_B)$

3) If transaction T_A starts its execution before transaction T_B completes, then the write phase of transaction T_B must be finished before transaction T_A starts the validation phase.

Advantages:

i) The efficiency of optimistic techniques lie in the scarcity of the conflicts.

ii) It doesn't cause the significant delays.

iii) Cascading rollbacks never occurs.

Disadvantages:

i) Wastage in processing time during the rollback of aborting transactions which are very long.

ii) Hence, when one process is in its critical section (a portion of its code), no other process is allowed to enter. This is the principal of mutual exclusion.