Transaction Processing

Transaction is a collection of data manipulation statements which is considered as the smallest unit of execution by DBMS. If all the statements are executed successfully then the transaction is said to be successful. A transaction has ACID properties as follows.

Atomicity: Transaction is an atomic unit of processing. Either it will be executed entirely or not at all.

Correctness/**C**onsistency preserving: A transaction must be logically correct so that execution of the transaction takes the database from one consistent state to another.

Isolation: A transaction should appear as if it is being executed in isolation from other transactions. In other words, it is not interfered by other transactions executed concurrently.

Durability: If a transaction completely successfully (commits) then its effect must persist in the database. Changes must not be lost due to any failure.

Atomicity is ensured by transaction management module of DBMS. Correctness/consistency preserving is the responsibility of programmer. Transaction logic should be proper. Isolation and durability are the responsibility of concurrency control module and recovery module (both are part of DBMS) respectively. Concept of a transaction is the foundation for concurrency control and recovery against failure.

Suppose, certain amount of money has to be transferred between two accounts. So it is to be ensured that balance of source account will be reduced and that of destination account will be increased. Both the changes are to be made. So both the updates together will form the transaction. Atomicity ensures that either both are done or not. To ensure consistency accounts will be incremented or decremented by the same amount. The logical correctness has to be ensured by the programmer. If multiple transactions are running concurrently and working with common data item then race condition (as you have studied in OS) must not occur. Isolation property safeguards it. Once a transaction completes successfully, it is to be guaranteed that the updates are available in the database, even if certain failure occurs after that. That's what the durability is for.

Defining a transaction: In oracle, one can go on writing the DML statements and all are part of same transaction. By issuing the commands COMMIT or ROLLBACK one can **explicitly complete** a transaction. After that new transaction will start. COMMIT means we want to complete the current transaction and its effect must be durable. On the other hand, ROLLBACK means end of current transaction but we want to cancel/abort the effect. There are implicit ways of completing a transaction. Whenever a DDL statement is executed, all the statements prior to are taken as a transaction, it is automatically committed. After the DDL statement another transaction starts. If we issue EXIT command at SQL prompt then it commits the current transaction and comes out of SQL plus. But, if we abruptly close the SQL plus window (clicking on x), it is equivalent to ROLLBACK. Try to recollect, sometimes we lost data while working in the lab. Hope, we understand the reason now. In PL/SQL whole block is a transaction.

Read/Write Operations in Transaction:

Read and write are the fundamental operations behind every DML activities. Henceforth, we will describe the transactions in terms these basic operations.

Read(X) reads the database item X into the program variable X. Steps for this are as follows.

1. Find the **disk block** in the database containing X
2. Is the **disk block** already present in **memory buffer** (will refer as **data buffer**)?
   A) If yes, then copy X from **data buffer** to **program variable**
   B) If not, then read the **disk block** into **data buffer** (in memory) and copy X from **data buffer** to **program variable**.
      - If free data buffer is available then read disk block in free data buffer
      - If free data buffer is not available then as per policy a data buffer will be written back to disk and new disk block will be read into this data buffer

Write(X) writes the value of program variable X to database item X in disk. Steps are as follows.

1. Find the **disk block** in the database containing X
2. Is the **disk block** already present in **memory buffer** (will refer as **data buffer**)?
   A) If yes, then copy X from **program variable** to proper position in **data buffer**
   B) If not, then read the **disk block** into **data buffer** (in memory) and copy X from **program variable** to proper position in **data buffer**. [Note, to read the block into buffer, again availability of free buffer and policy issues may come up].
3. Store the updated **data buffer** to **disk block**. But, exactly when the buffer will be written back that depends on policy.

Why is it important to control concurrency?

Several problems may occur unless concurrency is not handled properly. Few important issues are as follows.

1. **Lost update problem:**
   Consider T1 and T2 are two transactions. Say X and Y are balance of two accounts. T1 transfers Rs. 100 from account1 whose balance is denoted by X to account2 whose balance is denoted by Y. So X is decremented by 100 and Y is incremented by

the same amount. Suppose T2 denotes deposit of Rs. 500 to account1 i.e. X is incremented by 500. Suppose before the execution of the transactions value of X is 1000. Logically, after the desired transfers value of X should be 900. Now, consider, T1 and T2 are running concurrently. So the instructions of the transactions may interleave. Suppose the instructions of the transactions are executed in the following order.

T1	**T2**
Read(X)	
X=X-100	
	Read (X)
	X=X+500
Write(X)	
Read(Y)	
	Write (X)
Y=Y+100	
Write(Y)	

T1 first reads the current value of X (i.e. 1000) into its program variable and reduces the value of its program variable. So, program variable X of T1 becomes 900. After that T2 reads X into its program variable, so it also gets 1000. T2 changes its program variable to 1500. Thereafter, T1 writes its program variable to data item X to database item and the database item X becomes 900. Then it reads Y. After that T2 writes back its program variable X to database item X which becomes 1500. It is not the desired value. See, both T1 and T2 have read **same value** of database item X, modified their program variable and written back to database. T2 writing it later, its value is retained whereas in between the update made by T1 is lost. Finally, the result is incorrect. This is similar to race problem that we studied in OS and here this is called lost update problem.

2. **Temporary update/ dirty read problem:**
   Consider the following order of execution of instructions for T1 and T2.

T1	**T2**
Read(X)	
X=X-100	
Write(X)	
	Read (X)
	X=X+500
	Write (X)
Read(Y) ←fails at this point or later	
Y=Y+100	
Write(Y)	

   T1 has modified and wrote the value of X. T2 has read the modified value of X, changed it further and wrote back. After this suppose T1 has failed and as a result it will revert back the value of X. But the changes made by it temporarily, has already been consumed by T2. This is known as temporary update/dirty read problem.

3. **Incorrect summary problem:**
   Suppose a person has two accounts. Transaction T1 transfers money from one transaction to another. Whereas T2 displays the total balance of those two accounts. So before or after the transactions total sum of the balance should be same. Consider the following order of execution for the instructions of T1 and T2.

T1	**T2**
Read(X)	
X=X-100	
Write(X)	
	Read (X)
	Read (Y)
	Display (X+Y)
Read(Y)	
Y=Y+100	
Write(Y)	

T1 has reduced X. Thereafter, T2 reads X and Y and display summary. It has read modified value of X and old value of Y which was yet to be incremented. So the summary information displayed T2 is incorrect.

4. **Unrepeatable reads:**

T1	**T2**
Read (N)	
Display (N)	
	Read (N)
	N=N-5
	Write (N)
Read (N)	
Display (N)	

T1 reads the value of N twice and in between T2 changes it. So the two reads of same item will show different values. Suppose N stands for number of tickets available for a show. Between two reads some bookings have taken place through other transaction (s) and second read by the same transaction shows different value. We may have experienced similar situation at the time of bookings. So read is not repeated.

Repeatable read is difficult to ascertain. In certain applications it may not be desired also. But the other problems like lost update, temporary update and incorrect summary problem should be addressed through proper concurrency control strategies.

Why is recovery needed?

A transaction must satisfy ACID properties. Of which **C**orrectness of logic to preserve consistency is the responsibility of programmer. But system must ensure the other properties. Either all instructions of the transaction will be executed successfully or none at all (atomicity). If executed successfully then their effects must be recorded in the database (durability). Otherwise no effect will be on database and on other transactions. It is not permitted that effect of some of the instructions is applied to database and effect of rest is ignored. But this may happen if a transaction fails after executing a part. To combat such situation recovery management is essential. There may be various reasons that a transaction may fail in the middle of execution. Some of those are as follows.

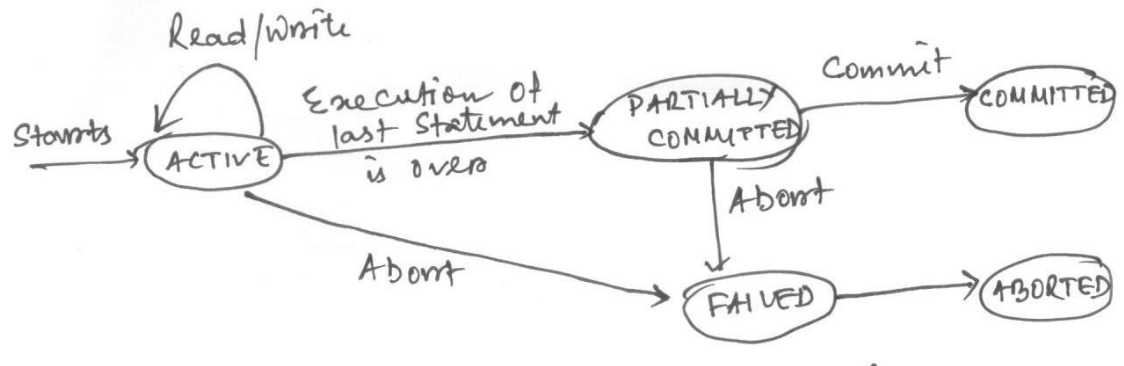
- **System failure:** Hardware, software or network error may occur.

- **Transaction error:** Certain operations in transactions like operation causing overflow may cause failure. User may interrupt the transaction also.
- **Exceptions detected by transaction:** During execution certain conditions may be detected by the transaction like, data not found. However, proper exception handling in the transaction (say, in PL/SQL block or in application) failure in such cases can be avoided.
- **Concurrency control policy:** In different situation (say, to avoid deadlock or to ensure serialization of concurrent access to an item etc.) concurrency control policy may abort transactions.

These are common causes of failure. System must maintain sufficient information to handle such failures and recover the database to consistent state. Other infrequent sources of failure may include disk failure (certain part of disk may not work), physical problems/catastrophic situation (fire, flood, overwriting of disk etc.). These will also invite recovery action.

Transaction States

Like, a process a transaction also passes through different states as shown in the figure.



State transition diagram for a transaction

When a transaction starts, it enters into active state. As long as different instructions/statements of the transactions are executed it remains in active state. One the execution of the last statement is over the transaction moves to partially committed. At this stage measures are taken to ensure durability and recovery. Once such activities are done transaction moves to committed state. When a transaction is in active or partially committed state failure may occur and transaction will be aborted. Then it moves to failed state. Note that, if a transaction reaches committed state it is guaranteed that its effects will be available in database and for a failed transaction no effect will be there in database.

Recovery

A common practice is to maintain a log (history) of activities that can be used in the recovery process.

In case the disk is damaged or catastrophic failure has occurred then only way of recovery is to the use latest backup version of database. On the database apply the log information (if available) till the point of failure to bring it into more recent updated version.

For the other non-catastrophic type of failures (remember we mentioned the same earlier) recovery can be done by **undoing** the changes made by failed transactions and **redoing** the work of committed transactions to put the database into consistent form. In this process log information will be utilized. Henceforth, we will concentrate on the recovery against non-catastrophic failure.

DBMS may follow **deferred update** or **immediate update** of database.

In **deferred update technique**, database is physically not updated till the transaction reaches commit point. Prior to that, updates are made in data buffer. Log information (what a transaction is doing) is also stored in log buffer. In partially committed state, log information in log buffer is written to log file in disk. Then only the transaction moves to committed state. Thereafter data buffer may be written to disk. Thus, if the transaction fails before it commits then it is guaranteed that changes made by it are not in database. So, there is no requirement of **undo** operation. Data buffers are not written to disk before the transaction commits. But, it may so happen that transaction has committed (log buffer written to disk). But due to certain failure, data buffers were not written to disk. Then **redo** operation has to be done by consulting the log file. So, the recovery strategy is **NO-UNDO/REDO** type.

In **immediate database update technique**, database may be updated by a transaction even before the transaction commits. It is important to note that before updating the database, corresponding log information is transferred from log buffer to log file. It is to ensure recovery. In case a transaction fails before commit, it is possible that certain changes made by it have already been posted to database. So for recovery **undo** operation is required. To ensure the posting of committed changes **redo** is also required. So, recovery strategy is **UNDO/REDO** type.

In both cases log information is essential to carry out recovery information.

Buffer management: We have already got the idea that transaction updates the data buffers. Physical update of database (i.e. when it will be done) depends on the policy adopted by DBMS. A directory is maintained to keep the track which database blocks are in which data buffer. Now remember, Read(X) checks the directory to find whether data block containing X is already in buffer or not. If not present, then data block is read into a free buffer. If there is no free buffer then one data buffer is chosen based on replacement strategy (LRU i.e. least recently used is a popular policy). It is copied into data block in disk then overwritten by the block being read into.

A **dirty bit** is associated for each data buffer (noted into directory entry) denotes that corresponding buffer has been modified or not. It is important because if the dirty bit is set (i.e. data buffer has been modified) then it must be copied back to disk before being overwritten by an incoming block.

While writing back a data buffer to disk the strategies may be either **in place updating** or **shadowing**. In case of in place updating, data buffer is written back to original disk block location. So a single copy of the blocks exits. Hence, log information is mandatory for recovery, In case of shadowing modified data buffer is written back to different location in disk. Thus old and new versions exist. Hence, log information is not essential for recover. We will focus on in place updating.

Like the data buffer, for log information there is log buffer. Transaction activities are noted into the log buffer. Prior to writing a data buffer to disk block (in place updating) log buffers holding the entries corresponding to the activities on the data buffer is written back to log file. Thereafter the data buffer is put to disk. It is known as **write-ahead logging (WAL)**. This is to ensure the recovery against failure.

Recovery strategy also depends on when a data buffer updated by a transaction cannot be written to disk till the transaction commits then it is **no-steal approach**. That is the case for deferred database update. For **steal approach** it is not so. Even before the transaction commits corresponding data buffers can be written back. So is the case for immediate database update. Mind it, WAL will take place always. Note that, in a concurrent environment number of transactions is active and working with number of data blocks. If no-steal approach is followed then there must be sufficient data buffers to fulfill the data requirement. It is so because the buffers cannot be written back till a transaction commits. But, in steal approach one can go with less number of buffers.

In no-steal approach, data buffers are written back after a transaction commits. But exactly at what point of time, it is not specified. In **force approach** all the data buffers updated by a transaction are written back to disk immediately after the transaction commits. It is not so for

no-force approach. Note that, multiple transactions may work on a data buffer. So each time one of the transactions commits, it will be written back to disk. It will increase the io operation.

Thus, **steal/no-force approach** is preferred as it has less buffer (memory) requirement and reduces io operation. At an instance it may so happen that certain uncommitted changes have been posted in the database (steal approach does not wait for commit of related transaction to sent back a data buffer to disk) and some committed changes have not been posted (no-force does not transfer updates every now and then a transaction commits). So at an intermediate stage the database may not be in consistent state. Finally, at the end of session, it will definitely move into consistent state.

- When a transaction commits, not necessarily updated data buffers are written back to disk. Commit of a transaction ensures following activities in sequence:
    a) Commit information of the transaction has been recorded in log buffer.
    b) All the log entries corresponding to the transaction has been transferred to log file.
    c) If force strategy is adopted then data buffers modified by the transactions are written back to disk.
- Whenever there is no free data buffer and an incoming block to be accommodated
    a) Select the data buffer to be written back. LRU may be a strategy for selection.
    b) Write back all the log entries of the transactions that modified the data buffer from log buffer to log file in disk (write-ahead logging).
    c) Write back the data buffer into disk block.
    d) Read the new block into the data buffer.

Whenever we need to write back a data buffer steps b) and c) in that order are to be followed.

Log based recovery process:

Case 1: Deferred update

Remember that in deferred update scheme database is not updated till a transaction commits (no steal case). So no uncommitted changes are transmitted to database. It is to be ensured that committed changes are reflected in database despite of failure. So for log based recovery the strategy will be **no-undo/redo.**

Suppose T_i is a transaction. Before it starts an entry **<T_i starts>** is written to log buffer. Whenever the transaction modifies a data item X in data buffer through Write statement a log entry of the form **<T_i X V_{new}>** is put into log buffer. V_{new} is the modified value of X. Thus, the log entries keep the information regarding changes made by the transaction. When the transaction reaches partially committed state (i.e. all the statements have been executed) log records of T_i are transferred from log buffer to log buffer. An entry **<T_i commits>** is sent to log file. Then the transaction enters into committed state. Actual update of database cannot take place before this. In case of failure, recovery is accomplished as follows.

- Prepare REDO-LIST (the list of transactions which have committed) as follows.

  REDO-LIST={}

  Scan the log file **from end move in the backward direction** till the start of the file is reached. If an entry like <T_i commits> is encountered then REDO-LIST=REDO-LIST U {T_i}
- Perform Redo operation

  Start from the **beginning of log file and move in forward direction** till the end of file is reached. Whenever an entry of the form <T_i X V_{new}> is encountered and T_i ϵ REDO-LIST, replace the value of data item X in database by V_{new}.

Note that, some of the committed changes may be already there in database. But doing it once again is not an issue. Redo operation if done multiple times then also effect remains same. Hence redo operation is **idempotent.**

Case2: immediate update

Unlike the deferred database modification, here the database may be modified even before the transaction commits (steal case). As a result, uncommitted changes may go to database. So for recovery not only that posting of changes made by committed transactions is to be ensured, but also the unwanted modifications by non-committed transaction sent to database has to be cancelled. So the recovery strategy will be undo/redo.

When a transaction T_i starts, an entry **<T_i starts>** is written to log buffer. Whenever the transaction modifies a data item X in data buffer through Write statement a log entry of the form **<T_i X V_{old} V_{new}>** is put into log buffer. V_{old} and V_{new} are the old and modified value of X respectively. It is essential to keep the old value to support undo (rollback) operation. When the transaction reaches partially committed state (i.e. all the statements have been executed) log records of T_i are transferred from log buffer to log buffer. An entry **<T_i commits>** is sent to log file. Then the transaction enters into committed state. In case of failure, recovery is accomplished as follows.

- Prepare REDO-LIST (the list of transactions which have committed) and UNDO-LIST (the list of transactions which could not commit) as follows.

  REDO-LIST={}

  UNDO-LIST={}

Scan the log file **from end move in the backward direction** till the start of the file is reached.

a) If an entry like <T_i commits> is encountered then REDO-LIST=REDO-LIST U {T_i}.
b) If an entry like <T_i starts> is encountered and if T_i not in REDO-LIST then UNDO-LIST=UNDO-LIST U {T_i}.

- Perform Undo (Rollback) operation

  Start scanning the log file **from end and move in backward direction** till start of the file is reached. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ UNDO-LIST, replace the value of data item X in database by V_{old}.
- Perform Redo operation

  Start from the **beginning of log file and move in forward direction** till the end of file is reached. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ REDO-LIST, replace the value of data item X in database by V_{new}.

While preparing REDO-LIST and UNDO-LIST, as we scan the log file from end in backward direction, in case a transaction has committed corresponding entry will be found before its start entry. So committed transaction will be placed into REDO-LIST prior to the start entry is encountered. Once the start entry is encountered, it is checked whether it is already in REDO-LIST or not. If not, then it is placed into UNDO-LIST. So, in a single scan of log file two lists can be prepared.

Note that, undo operation is done in reverse order. It will be clear once we go through concurrency. Just to get the idea, suppose a transaction has changed an item value from 10 to 20. Later another transaction has made it from 20 to 50. Log records will also maintain similar chronology. Both the transactions are active (so, a concurrent scenario). Later, say, both the transactions are to be rolled back. Undo operation in reverse order will make the data item 20 at first step and then to 10. But if the log records are applied in forward direction it would have been 20 and that's an error.

Checkpoint:

Drawbacks of log based recovery are as follows.

- Log file is quite large and scanning the whole file is time consuming.
- Most of the Redo operations are redundant as in most of the cases changes are already posted to database

To address the problems, checkpoint concept can be utilized. DBMS periodically performs checkpoints and as a result following actions are taken in sequence.

- All log records in log buffers are sent to log file in disk.
- All modified data buffers are sent to database.
- A log record **<checkpoint>** is sent to log file.

Checkpoint operation ensures that all the changes made by committed transactions at that point of time are sent to disk. While writing data buffers to disk failure can take place, hence log records are sent prior to that. It also sends the changes made by active transactions to database. Such transactions may fail after checkpoint and their effect has to be rolled back. For non-concurrent situation such transaction is at most one and for the time being we will restrict ourselves with in this scenario. Log based recovery with checkpoints is similar to the algorithm that we have already gone through. But there is no need to go through the whole log file. Move from the end of the log file in the backward direction till we reach the most recent <checkpoint> and move further till we encounter an entry of the form <T**i** starts>. From this point to end of log file will be utilized in recovery process as follows.

- Prepare REDO-LIST (the list of transactions which have committed) and UNDO-LIST (the list of transactions which could not commit) as follows.

  REDO-LIST={}

  UNDO-LIST={}

  Scan the log file **from end move in the backward direction till <checkpoint>** is reached.

  a) If an entry like <T_i commits> is encountered then REDO-LIST=REDO-LIST U {T_i}.
  b) If an entry like <T_i starts> is encountered and if T_i not in REDO-LIST then UNDO-LIST=UNDO-LIST U {T_i}.

  Move further **beyond the <checkpoint> in backward direction till <T_i starts>** is found. If T_i not in REDO-LIST then UNDO-LIST=UNDO-LIST U {T_i}.

- Perform Undo (Rollback) operation

  Start scanning the log file **from end and move in backward direction till first <T_j starts> beyond most recent <checkpoint>** is reached. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ UNDO-LIST, replace the value of data item X in database by V_{old}.

- Perform Redo operation

Start from the position that has been reached after undo operation (**i.e. first <T_j starts> prior to most recent <checkpoint>) and move in forward direction** till the end of file is reached. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ REDO-LIST, replace the value of data item X in database by V_{new}.

It has been assumed that immediate database update has been followed. However, if deferred database update is followed then UNDO-LIST is not required and undo operation will also be dropped. Rest of the steps will carry out the recovery.

Concurrency Control

In a concurrent execution scenario multiple transactions may be in active state and processor will be shared. It improves processor utilization and throughput (amount of work done in a time interval) increases. However, response time may increase. Moreover, concurrency gives rise to certain issues as discussed earlier. Hence proper strategy for concurrency control is essential.

Schedule: Suppose a number of transactions are running concurrently. A schedule represents the chronological order in which statements of the participating transactions are executed by the system. A schedule must satisfy the following:

- All the statements of the participating transactions must appear in the schedule
- Statements of the transactions must appear in the schedule preserving the order in which they appear in the individual transaction.

In a transaction T_i if statement S_j appears before S_k then in the schedule also S_j will come before $S_{k.}$

In a schedule if for every participating transaction the statements appear together (not interleaved by the statements of other transactions) then it is called a **serial schedule**. Otherwise, if statements of different transactions appear in interleaved fashion then it is called a **non-serial** or **concurrent schedule**.

Consider T1 and T2 are two transactions. Consider the schedule S1 in which all the statements of T1 appears first and followed by those of T2. It is a serial schedule where T1 is followed by T2. Another serial schedule could have been other way around where T1 would have followed T2.

Schedule S1:

T1	T2
Read(X)	
X=X-100	
Write(X)	
Read(Y)	
Y=Y+100	
Write(Y)	
	Read (X)
	X=X-500
	Write(X)
	Read (Y)
	Y=Y+500
	Write(Y)

Consider the schedule S2. Here, statements of the two transactions are in interleaved fashion. Hence, it is a non serial schedule.

Schedule S2:

T1	T2
Read(X)	
X=X-100	
Write(X)	
	Read (X)
	X=X-500
	Write(X)
Read(Y)	
Y=Y+100	
Write(Y)	
	Read (Y)
	Y=Y+500
	Write(Y)

If there are n transactions participating in a schedule then it is possible to have n! different serial schedules. But number of possible non-serial schedule is quite huge and it depends on number of participating transactions, number of statements and interleaving pattern. However, all such schedules will not result into correct state. Consider the schedule S3. Note, both the participating transactions reducing the value of X by an amount and increasing Y by same. So, in any consistent state sum of X and Y will remain same. As per schedule of S3, effectively X will be reduced by 100 (instead if 600 as in S2) and Y is increased by 500 (not by 600). So, S3 does not preserve consistency. So, all the concurrent schedules are not permissible.

Schedule S3:

T1	**T2**
Read(X)	
X=X-100	
	Read (X)
	X=X-500
	Write(X)
	Read (Y)
Write(X)	
Read(Y)	
Y=Y+100	
Write(Y)	
	Y=Y+500
	Write(Y)

Remember, we have discussed the necessity of concurrency control to avoid the problems like, lost update, dirty read, incorrect summary etc. It is important to follow proper policy of concurrent control.

Conflict serializable schedule: In a schedule ordering of two consecutive instructions can be interchanged provided they are non-conflicting and such interchange is known as **non-conflicting swap**.

Suppose in a schedule S, I_i and I_j are two **consecutive instructions** such that

- they belong to two different transactions T1 and T2 respectively
- both the instructions work on same data item X

I_i and I_j conflict if at least one of them is Write(X) else they are non-conflicting.

Basically, non-conflicting swap does not alter the result. Note that, even if interchanging the instructions of a single transaction keeps the result unaltered still it is not done as the ordering of instructions of an individual transaction has to be preserved. So for non-conflicting swap the instructions must be of different transactions. If I_i and I_j work on different data item then also their ordering is non-issue.

Suppose, I_i and I_j are Read(X) and Write(X) respectively. Their ordering is important. If I_i precedes I_j then I_i will read old value of X. otherwise it reads the new value of X if Write(x) comes before Read(X). So they are conflicting. The ordering of conflicting instructions has impact on the outcome.

Similarly, if I_i and I_j are Write(X) and Read(X) respectively then also they conflict.

If both are Write(X) then their ordering has impact on subsequent Read(x).

If both are Read(x) then they do not conflict.

A schedule S is a **conflict serializable schedule** if it is conflict equivalent to a serial schedule S'. In other words, S can be transformed into S' through a series of non-conflicting swaps of instructions. Such a concurrent schedule is acceptable.

Schedule S4:

T1	**T2**
Read(X)	
Write(X)	
	Read (X)
	Write(X)
Read(Y)	
Write(Y)	
	Read (Y)
	Write(Y)

Consider the schedule S4. Here, only Read/Write statements have been shown. Change of program variables (like, X=X+100 etc.) has been dropped. Here, Read(Y) of T1 and Write(X) of T2 are non-conflicting. So, Read(Y) of T1 goes up and Write(X) of T2 can come down. After this swap, Read(Y) of T1 and Read(X) of T2 become consecutive statements which are not conflicting and therefore can be swapped. So the schedule becomes as follows.

T1	**T2**
Read(X)	
Write(X)	
Read(Y)	

	Read (X)
	Write(X)
Write(Y)	
	Read (Y)
	Write(Y)

Now, Write(Y) of T1 and Write(X) of T2 are considered. Again through successive swaps, finally the schedule gets transformed into S'.

Schedule S':

T1	**T2**
Read(X)	
Write(X)	
Read(Y)	
Write(Y)	
	Read (X)
	Write(X)
	Read (Y)
	Write(Y)

S' is a serial schedule. Thus, schedule S4 can be transformed into a serial schedule S' through a series of non-conflicting swap. So, S4 is conflict equivalent to the serial schedule S' and it is a conflict serializable schedule.

Testing for conflict serializability: In order to test whether a schedule S is conflict serializable or not we can form a **precedence graph**. It is a **directed graph** G consisting of pair (V, E). V is the set of vertices and E is the set of vertices. Corresponding to each participating transaction there is a vertex. An edge, $T_i \rightarrow T_j$ (from transaction T_i to transaction T_j) is there if any of the following situation exists:

- In S, T_i executes Read(X) before T_j executes Write(X)
- In S, T_i executes Write(X) before T_j executes Read(X)
- In S, T_i executes Write(X) before T_j executes Write(X)

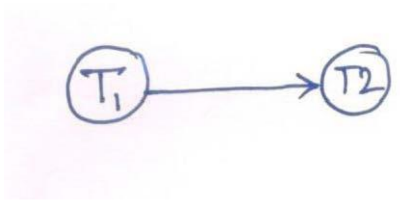
In other words, $T_i \rightarrow T_j$ implies that in a serial schedule S' which is equivalent to S, T_i must appear before T_j.

In the precedence graph, if there exists a cycle then the corresponding schedule S is not conflict serializable. If there is no cycle then S is conflict serializable.

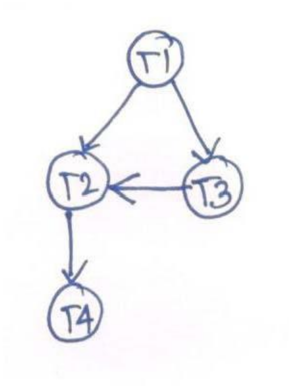
Say, in the precedence graph edges like, $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$ exist. They form a cycle. Thus, in the equivalent serial schedule $T_i \rightarrow T_j$ demands that T_i must appear before T_j and $T_j \rightarrow T_i$ demand that T_j must appear before T_i. This is not possible to fulfill.

To test the conflict serializability of a schedule, prepare the precedence graph and detect the cycle. If there are n transactions in the schedule (number of nodes in the graph), the complexity of cycle detection algorithm is of order n^2.

Consider the schedule S1 and corresponding precedence graph is as follows.



The following graph is consists of four transactions T1, T2, T3 and T4. As there is no cycle



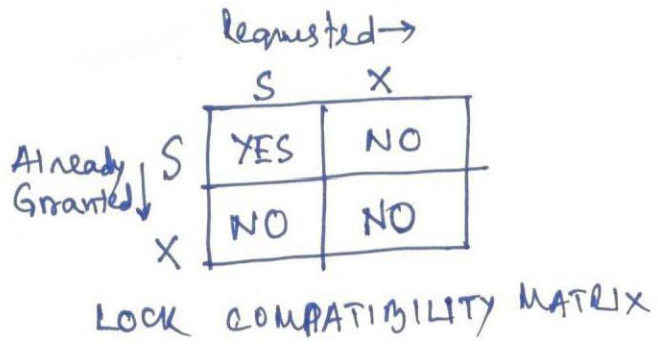
Corresponding schedule is conflict serializable. Conflict equivalent serial schedule will have the instructions of participating transactions in the following order: T1-T3-T2-T4. It is so because the graph shows T2 precedes T4. T3 precedes T2 and T1 precedes both T2 and T3.

<u>Ensuring Conflict Serializability:</u>

1. **Lock based protocol**
   In order to ensure serializability, it is necessary to control the access to data item. A transaction will acquire lock on a data item prior to accessing the same. There are different modes of lock. Transactions request a mode depending on its

requirement. In the simplest form locks may be **exclusive** or **shared**. Holding exclusive lock on a data item means that the transaction can do both read and write on data item. For shared lock only read is possible. If a transaction acquires exclusive lock on the data item, no other transaction can get access to it till the transaction holding the exclusive lock releases the same. Number of transactions can have shared lock on a data item simultaneously. In this read operation is possible. So, simultaneous read of a data item is possible. If a transaction is holding shared lock on a data item no other transaction can acquire exclusive lock on it. Note that any update operation is a combination of both read and write. Hence, to modify a data item the transaction must acquire exclusive lock on the data item.

There is a **lock compatibility matrix** that specifies whether mode of lock granted on certain data item and mode of lock requested on same data item are compatible or not. If lock in certain mode (say, M1) on a data item Q has been currently granted to a transaction T_i and lock in certain mode (say, M2) on the same data item has been sought by another transaction T_j are not compatible then lock cannot be granted to T_j and it will have to wait. The matrix shows that if either of M1 and M2 is exclusive (X) then they are not compatible. Compatible only if both are shared (S) mode.



Mere use of locks is not sufficient to control the concurrency to ensure consistency of the database. Let us look into the following schedule S5.

Schedule S5:

T1	**T2**
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	
Unlock(A)	

	Lock-S(A)
	Read(A)
	Unlock(A)
	Lock-S(B)
	Read(B)
	Unlock(B)
	Display(A+B)
Lock-X(B)	
Read(B)	
B=B-100	
Write(B)	
Unlock(B)	

Lock-X(Q) denotes the request to have exclusive lock on Q whereas Lock-S() is the request for shared lock. Unlock() denotes release of the lock. Whenever a transaction reaches terminal state (committed/aborted) then all the locks held by it are automatically relinquished. In schedule S5 T1 modifies the data items A and B. One is increased and other one is decreased by the same amount. So, in a consistent state sum of A+B should remain unchanged. T1 intends to modify A and B. Hence seeks exclusive locks. As per schedule T1 gets the lock on A, modifies A and releases the lock. So, T2 can acquire locks on A and B. Once it releases the locks T1 will proceed. But T2 reads modified value of A and old value of B giving inconsistent output. **So, mere use of locks is not the solution**.

In order to overcome the problem one may think **that instead of releasing the locks as and when the operation is over** (as it was the case in S5. Exclusive lock on A was released immediately after Write(A)) **retain the same till the end of transaction**. Schedule S6 shows the changed version.

Schedule S6:

T1	**T2**
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	
	Lock-S(A)
	Read(A)
	Lock-S(B)

	Read(B)
	Unlock(A)
	Unlock(B)
	Display(A+B)
Lock-X(B)	
Read(B)	
B=B-100	
Write(B)	
Unlock(A)	
Unlock(B)	

Now, after T1 performs Write(A), T2 tries to get the lock on A in shared mode. As per lock compatibility matrix it cannot be granted as T1 still holds exclusive lock on A. So, T2 waits. At some point T1 will resume and once it unlocks T2 will proceed. It will display consistent result.

Let us modify schedule S6 further. Say, T2 works with A and B in reverse order. S7 shows the changed schedule. Now, after T1 performs Write(A), T2 gets the shared lock on B. It reads B. But T2 will have to wait for the lock on A. Now T1 also cannot proceed as it cannot acquire the exclusive lock on B (as shared lock on B is with T2). T1 waits for the lock on B that it can get only when execution of T2 is over. T2 also waits for the completion of T1. So, there is a cyclic wait (arising out of hold (a lock) and wait (for another) nature of the transactions and it's a deadlock situation. *[To come out of it, transaction(s) may have to forcibly aborted – a cause of failure][It is advised that if the transactions are working with sane data items then they must be used in same order. Here in schedule S7, T1 first uses A and then B. It is reverse for T2. But in schedule S6, both the transactions used in same order.]*

Schedule S7:

T1	**T2**
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	
	Lock-S(B)
	Read(B)

	Lock-S(A)
	Read(A)
	Unlock(B)
	Unlock(A)
	Display(A+B)
Lock-X(B)	
Read(B)	
B=B-100	
Write(B)	
Unlock(A)	
Unlock(B)	

To summarize the observations: lock and unlock are to be used judiciously. Early release of locks may increase the concurrency at the cost of consistency. Retaining of locks till end of transaction can maintain consistency sacrificing the concurrency. Moreover, it can lead to deadlock also. So we need a locking protocol to decide about the timings for locking and unlocking of data items.

Let {T_1, T_2,..T_n} be a set of transactions taking part in a schedule. Like the precedence graph, a similar graph can be constructed for the schedule. Here, **$T_i \rightarrow T_j$ denotes that: In the schedule, T_i holds lock on a data item A in mode M1, T_j will hold lock on A in mode M2 at a later point of time and M1 and M2 are not compatible**. So, in an equivalent serial schedule T_i must precede T_j. Conflict between instructions in precedence graph is mapped to non compatibility of lock modes.

If a schedule is possible following the rule of a locking protocol then it is said to be legal w.r.t. the locking protocol. It is conflict serializable if the associated $\rightarrow$ relation is acyclic.

Two phase locking protocol: Two phase locking protocol ensures conflict serializability. Each transaction requests to lock or unlock in two phases namely, **growing phase** and **shrinking phase**.

When a transaction starts it enters into growing phase and it can ask for lock only when it is in growing phase. As if it can grow and acquiring data to work on. When the transaction releases a lock it enters into shrinking phase. In this phase it can ask for any lock and can only give up the locks. As if, it is moving towards completion, no more additional data will be required (hence, shrinking) and will go on releasing the locks that it held.

Schedule S5 again:

T1	T2
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	
Unlock(A)	
	Lock-S(A)
	Read(A)
	Unlock(A)
	Lock-S(B)
	Read(B)
	Unlock(B)
	Display(A+B)
Lock-X(B)	
Read(B)	
B=B-100	
Write(B)	
Unlock(B)	

Look into the schedule S5. It gives rise to incorrect summary problem. Neither T1 nor T2 follows two phase locking protocol. T1 has released lock on A, thereby enters into shrinking phase and after that asking for lock on B. It violates two phase locking protocol. Same is the situation for transaction T2. It releases lock on A and request for lock on B. So the schedule is not a legal one under two phase locking protocol.

Schedule S6 again:

T1	T2
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	

	Lock-S(A)
	Read(A)
	Lock-S(B)
	Read(B)
	Unlock(A)
	Unlock(B)
	Display(A+B)
Lock-X(B)	
Read(B)	
B=B-100	
Write(B)	
Unlock(A)	
Unlock(B)	

Let us revisit schedule S6. It is a legal one under two phase locking protocol and it is also conflict serializable (equivalent to a serial schedule where T1 is followed by T2). Not, it overcomes incorrect summary problem.

Look into the schedule S7. The transactions are following two phase locking protocol. But we have seen earlier it gives rise to deadlock.

To summarize, two phase locking protocol ensures conflict serializability. But, it is not free from deadlock.

Schedule S7 again:

T1	**T2**
Lock-X(A)	
Read(A)	
A=A+100	
Write(A)	
	Lock-S(B)
	Read(B)
	Lock-S(A)
	Read(A)
	Unlock(B)
	Unlock(A)
	Display(A+B)

Lock-X(B)
Read(B)
B=B-100
Write(B)
Unlock(A)
Unlock(B)

Consider the following schedule that gives rise to lost update problem as Write(X) by T1 is lost as T2 consumes old value of X (lost in the sense, modification made by T1 is not consumed by anybody, as if it did not take place at all). Let us try to put the lock and unlock following two phase locking protocol. Schedule S8 is transformed into S9.

Schedule S8:

T1	**T2**
Read(X)	
X=X-100	
	Read (X)
	X=X+500
Write(X)	
Read(Y)	
	Write (X)
Y=Y+100	
Write(Y)	

Schedule S9:

T1	**T2**
Lock-X(X)	
Read(X)	
X=X-100	
	Lock-X(X)
	Read (X)
	X=X+500
Write(X)	

Lock-X(Y)	
Unlock(X)	
Read(Y)	
	Write (X)
	Unlock(X)
Y=Y+100	
Write(Y)	
Unlock(Y)	

Note that, S9 is legal under two phase locking protocol. Unless T1 executes Write(X) and acquires lock on Y it cannot relinquish lock on X. So, T2 will wait and it will consume the new value of X caused by T1. **So, lost update problem is also addressed**.

2. **Timestamp based protocol**

   A timestamp $TS(T_i)$ is assigned by DBMS to every transaction T_i in the system. Let T_i and T_j are two transactions with timestamps $TS(T_i)$ and $TS(T_j)$ respectively. If T_i enters into the system earlier then $TS(T_i) < TS(T_j)$. Timestamps can be assigned using the system clock. It can also be done by maintaining a counter that will be incremented after assigning every timestamp.

   Serializability order is determined by the timestamps of participating transactions. If $TS(T_i) < TS(T_j)$ then it is to ensured that produced schedule is equivalent to a serial schedule where T_i is followed by T_j.

   With every data item X, two timestamps, W-TS(X) and R-TS(X) are maintained. W-$TS(T_i)$ stands for write timestamp and it is the largest timestamp of the transaction that successfully executed Write(X). Similarly, R-$TS(T_i)$ stands for read timestamp and it is the largest timestamp of the transaction that successfully executed Read(X). These two timestamps are updated after every write/read operations.

   Timestamp ordering protocol:

   Suppose transaction **T_i issues Read(X)**. It is allowed if X has not been overwritten by a transaction that has started later (i.e. $TS(T_i)$<W-TS(X)). As if, T_i was supposed read the value of X that is already lost. Hence, it must be cancelled and if required will be restarted with higher timestamp. Thus, **T_i issues Read(X)** is evaluated as follows.

   - If $TS(T_i)$<W-TS(X) then reject Read(X) and rollback T_i
   - If $TS(T_i)$>=W-TS(X) then allow Read(X) and set R-TS(X)=max(R-TS(X), $TS(T_i)$)

Note, in assessing "T_i issues Read(X)", it is immaterial whether another transaction that started earlier/later has read the value of X. The fate is decided based on W-TS(X).

If **T_i issues Write(X)** then it is allowed if no other transaction that started later has already read/written the value. As if, the value of X supposed to be written by T_i would have been read by a transaction that started later. But such transaction has already worked with old value. So there is no use of this write by T_i. Write(X) by a transaction that started later is supposed to prevail by overwriting the value written by T_i. As such transaction has already written then it should be allowed to prevail. Thus, **T_i issues Write(X)** is evaluated as follows.

- If $TS(T_i)<R\text{-}TS(X)$ then reject Write(X) and rollback T_i
- If $TS(T_i)<W\text{-}TS(X)$ then reject Write(X) and rollback T_i
- Otherwise allow Write(X) by T_i and set W-TS(X)=max(W-TS(X), TS(Ti))

Timestamp ordering protocol ensures conflict serializability. Also note that in this protocol there is no wait situation. Either a transaction is allowed to perform the requested operation or it is rolled back/cancelled. Hence, unlike lock based protocol (it had wait situation), time stamp ordering protocol is deadlock free.

Recovery in Concurrent Execution

Cascading Rollback: In a concurrent environment, for recovery from a transaction failure it may be necessary to roll back number of transactions. It happens when data written by the failed transaction is used by others. Consider the schedule S10 as follows.

Schedule S10:

T1	**T2**	**T3**
Read(X)		
Write(X)		
Read(Y)		
Read(Z)		
	Read(X)	
	Write(X)	
		Read(X)

In the schedule, part of statements of the participating transactions has been shown. T1 has changed the value of X. That has been used by T2 and T3. Suppose once T2 and T3 have used the changed value of X then T1 fails. Not only T1 but also T2 and T3 are to rolled back. This is cascading rollback. It is the phenomenon where failure of single transaction results into rollback of number of transactions.

Cascading rollback can also happen under two phase locking protocol as shown in schedule S11 which is nothing but schedule S10 under two phase locking protocol.

Schedule S11:

T1	**T2**	**T3**
Lock-X(X)		
Read(X)		
Write(X)		
Lock-X(Y)		
Read(Y)		
Lock-X(Z)		
Read(Z)		
Unlock(X)		
	Lock-X(X)	
	Read(X)	
	Write(X)	
	Unlock(X)	
		Lock-X(X)
		Read(X)

Timestamp-ordering protocol is also not free from cascading rollback. Consider the schedule S10. Say the transactions enter into the system following the order as T1, T2, and T3. So, TS(T1)<TS(T2)<TS(T3). So Read(X) and Write(X) by T1 are allowed. Read and write operations on X by T2 are also valid as per protocol. Same with Read(X) by T3. So, this partial schedule is also possible under time-stamp ordering protocol. So, failure of T1 leads to roll back of all the three transactions.

Cascading rollback is not desirable as it requires a lot of undoing. Hence, the concurrency control protocols must be free from this phenomenon.

Recoverable Schedule: More critical issue is to ensure recoverability. A schedule must ensure recoverability.

Schedule S12:

T1	T2
Read(X)	
Write(X)	
Read(Y)	
	Read(X)
	Write(X)
Write(Y)	

Consider the schedule S12. After Read(Y) by transaction T1, the statements of T2 are executed. Suppose T2 commits and after that T1 fails. So for recovery, not only T1 has to be rolled back T2 also has to be rolled back as it has worked with the value of X that was set by T1. But, T2 has already committed and its effect cannot be removed. So, schedule S12 is not a recoverable one.

In a concurrent schedule, suppose a transaction, T_j works with a value modified by another transaction (say, T_i). If the schedule is such that T_j can commit before the completion of T_i then the failure of T_i after the completion of T_j cannot be recovered. Such a schedule is a non-recoverable one. **A concurrency control protocol must ensure recoverability.**

Schedule S12 is also possible under two phase locking protocol as shown in Schedule S13. Thus, it is clear that **two phase locking protocol cannot ensure recoverability**.

Schedule S13:

T1	T2
Lock-X(X)	
Read(X)	
Write(X)	
Lock-X(Y)	
Read(Y)	
Unlock(X)	
	Lock-X(X)
	Read(X)
	Write(X)
	Unlock(X)
Write(Y)	

Unlock(Y)

However, it is possible to modify two phase locking protocol and cascading rollback can be avoided. Recoverability can also be achieved. The change to be made is that all the **acquired exclusive locks must be held by the transaction are to be retained till it commits**. Thus, no transaction can use the uncommitted changes made by other transaction. Under this modified protocol the schedule will be as follows (schedule S14).

Schedule S14:

T1	**T2**
Lock-X(X)	
Read(X)	
Write(X)	
Lock-X(Y)	
Read(Y)	
	Lock-X(X)
	Read(X)
	Write(X)
	Unlock(X)
Write(Y)	
Unlock(X)	
Unlock(Y)	

Compare schedules S13 and S14. In S14, Note that T1 performs Unlock(X) only after it executes Write(Y). Now, T2 cannot get the lock on X till T1 completes. It will have to wait. So, the issue of cascading rollback is avoided and recoverability is also ensured.

Timestamp-ordering protocol also does not ensure recoverability. Look into the schedule S12 again. Assuming T1 enters into the system before T2 (i.e. TS(T1)<TS(T2)), schedule S12 is legal under time-stamp ordering protocol. Thus, it can also give rise to the discussed scenario resulting into non-recoverability. However, **timestamp-ordering protocol can be modified to ensure recoverability**. It is as follows.

- With each transaction a commit bit is associated. It is initially false and set to true when the transaction commits.
- Suppose a transaction T wants to perform Read (X) then If TS(T)<W-TS(X) then reject Read(X) and rollback T
- If TS(T)>=W-TS(X) then

```
{
  If **commit bit (b) of last transaction performing Write(X)** is true then
    { allow Read(X) by T and set R-TS(X)=max(R-TS(X), TS(T)) }
  else
      { **wait till b is true** }
}
```

Thus, here also uncommitted changes made by a transaction are not transmitted to others. Note that, wait state has now been introduced. Follow carefully that a transaction can wait for another transaction with lower time-stamp. Hence, cyclic wait is never possible. So the modified protocol is still deadlock free.

Log based Recovery and Checkpoint: As the part of recovery mechanism we have gone through log based recovery. To avoid scanning large log file and also to minimize redundant redo operation concept of checkpoint was discussed. In the context of checkpoint, one major difference arises in case of concurrent environment. In case of non concurrent scenario, only one transaction may be in active state. But now number of transaction may be in active state when system performs checkpoint. When system does check point along with other operations (i.e. writing back log buffers and modified data buffers) it writes **<checkpoint L>** in the log file. L stands for the list of active transactions. In case of non-concurrent scenario, recovery operation is carried out considering the **log record from the immediate past <T_i strats> of most recent checkpoint entry to the end of log file.** That <T_i starts> may correspond to the transaction that was in active state. Now, **prior to <checkpoint L>, we will have to consider all the log entries until <T_i starts> for all T_i ϵ L are encountered.** Considering immediate update modification, the recovery strategy will be as follows.

- Prepare REDO-LIST (the list of transactions which have committed) and UNDO-LIST (the list of transactions which could not commit) as follows.

  REDO-LIST={}

  UNDO-LIST={}

  Scan the log file **from end move in the backward direction till <checkpoint L>** is reached.

  a) If an entry like <T_i commits> is encountered then REDO-LIST=REDO-LIST U {T_i}.
  b) If an entry like <T_i starts> is encountered and if T_i not in REDO-LIST then UNDO-LIST=UNDO-LIST U {T_i}.

  For every transaction T_i ϵ L, if T_i not in REDO-LIST then UNDO-LIST=UNDO-LIST U {T_i}

- Move further **beyond the <checkpoint L> in backward direction till <T_i starts> for all $T_i \in$ L is reached**.
- Perform Undo (Rollback) operation
- Start scanning the log file **from end and move in backward direction beyond most recent <checkpoint L> till <T_i starts> for all $T_i \in$ L is reached**. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ UNDO-LIST, replace the value of data item X in database by V_{old}.
- Perform Redo operation

  Start from the position that has been reached after undo operation (**i.e. <T_j starts> of the oldest transaction among L of most recent <checkpoint L>) and move in forward direction** till the end of file is reached. Whenever an entry of the form <T_i X V_{old} V_{new}> is encountered and $T_i \in$ REDO-LIST, replace the value of data item X in database by V_{new}.

Note, undo has to be done in backward direction. Suppose log entry shows one transaction change the value of a data item from v to v1. Later another entry shows that for the same data item the value is changed from v1 to v2. Say, both are to be cancelled out to make the data item value V. But if we undo in forward direction at first step data item value will be v and due to subsequent entry it will be v1 which is incorrect. But, undo operation done in reverse order will make the value correct.

If deferred database update is adhered then there will be no undo operation. Hence in the algorithm certain changes are to be made. As it will be no-undo/redo strategy, there is no need to prepare UNDO-LIST and undo activity is to be dropped. Furthermore, log records will not have old value as its part.

[*Note, locking/unlocking and recovery as and when required are done by DBMS itself. In oracle for example, during shutdown also it considers log records to maintain consistency of the database. In case there is abrupt termination of oracle, then during start up process it performs recovery. Oracle has variety of lock modes at the row level and table level. These are automatically applied as per the policy. For Select operation no lock is required. For update operation exclusive locks at the row level are applied. So, two transactions can go for updating two different sets of rows concurrently. It enhances concurrency. As user, in case of special requirement can explicitly go for lock acquisition and this is at table level. There is a command LOCK TABLE. Normally use of locks by explicit command is not advisable. Better to rely on the system.*]