Transaction Management and Concurrency Control:
Transaction, properties of transactions, transaction log, and transaction management with SQL using commit rollback and save point.

Concurrency control for lost updates, uncommitted data, inconsistent retrievals and the scheduler. Concurrency control with locking methods : lock granularity, lock types, two phase locking for ensuring serializability, deadlocks, concurrency control with time stamp ordering : Wait/Die and Wound/Wait Schemes, Database Recovery management : Transaction Recovery.

Transaction Management: is the foundation for concurrent execution and recovery from system failure in DBMS

Transaction:

- Transaction is an execution of a user program.
- DBMS considers a transaction as a series of reads and writes.
- It performs a single unit of DB processing.
- Transactions have to be managed if the system is a multiuser system.

Properties of transaction:

A DBMS must ensure 4 important properties of transactions to maintain data in the face of concurrent access and system failure
They are

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Atomicity :

- Atomicity means either all actions are carried out or none are.
- Users should not have to worry about the effect of incomplete transactions.
- Eg: Funds transfer (transfer of *Rs*. 500 from account A to account B)

```
Read(A)
A = A – 500
Write(A)
Read(B)
B = B + 500
Write(B)
```

- In the above example, either all operations should be done or none of the operations of transaction are done.
- Atomicity is maintained by the **Recovery Manager** (transaction management component of DBMS

Consistency : (correctness)

- Every user transaction must ensure that it will lead database instance from consistent state to another consistent state.
- Each transaction run by itself, with no concurrent execution of transactions, must preserve the consistency of database.
- The DBMS assumes that consistency holds for each transaction.
- Ensuring this property is the responsibility of the **user** (application developer).
- Eg: user can have consistency criterion like – fund transfer between bank accounts should not change the total amount of money in the account.

Isolation :

- In case of multiple transactions executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that, they should not create any anomaly in values stored at the sharable resource.
- When multiple transactions execute simultaneously, the net result should be equal to executing the transactions in some serial order.
- This property is ensured by **concurrency control manager** component of DBMS.

Durability:

- If the transaction has been successfully completed, its effect should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.
- Changes must not be lost due to some database failure. Changes should be permanent.
- This property is ensured by **Recovery manager** component of DBMS.

Discussion

- If each transaction maps a consistent database instance to another consistent database instance. Executing several transactions one after the other results in a consistent final database instance. User/application programmer is responsible for ensuring transaction consistency ? how?
  - By the logic of his program. By putting some criterion like total funds should be equal.
- The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order.
- Transactions may be incomplete because of
  - They can be aborted / terminated unsuccessfully
  - System may crash
  - Transaction may encounter an unexpected situation.
- If a transaction is interleaved in the middle, it leaves the database in an inconsistent state.
- DBMS must ensure transaction atomicity. How to?
  - By undoing actions of incomplete transactions.
- How DBMS will undo?
  - With the help of log. DBMS maintains a log of all write to the database.
- Log is also used to ensure durability.
- Consider that a system crash happened. Log is used to remember and restore these changes when the system restarts.

Transaction Log

- The log is a history of actions executed by the DBMS. i.e., it keeps track of all transactions that update the database.
- Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes.

- DBMS uses the information stored in a log for
  - *Recovery requirement triggered by a rollback statement.*
  - *A programs abnormal termination.*
  - *A system failure*

- Atomicity and Durability properties are ensured by the DBMS with the help of the log file.
    - *Atomicity property states that either all the operations of a transaction must be performed or none. The modifications done by an aborted transaction should not be visible to a database and the modifications done by a committed transaction should be visible. DBMS carries out undo or redo with the help of log file records.*
    - *Durability of log file can be achieved by maintaining two or more copies of the log on different disks, so that the chance of all copies of the log being lost is negligibly small.*

- Every log record is given a unique id called the **log sequence number** (LSN)
- **Log tail :** most recent portion of the log file, which is placed in main memory.
- Every log record has prevLSN, transID, and type. The set of all log records for a given transaction is maintained as a linked list going back in time, using the prevLSN field.

- A log record is written for each of the following actions.
    - ***Update :*** after modification, an update type record is appended to the log tail. Every update record contains transID, Dataitem that is being modified, old value and new values of the data item.
    - ***Start***: When a transaction starts, it is assigned a transaction id and that is written to the log file.
    - ***Commit :*** When a transaction decides to commit, it writes a commit type record containing the transaction id. That is the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record. The transaction is considered to have committed at the instant that its commit log record is written to stable storage.
    - ***Abort :*** When a transaction is aborted, an abort type log record containing the transaction is is appended to the log, and Undo is initiated for this transaction.

- With the help of the information stored in the log, system can perform redo and undo operations.
    - Undo – setting the data item to old value.
    - Redo – setting the data item to new value.

Recovery using log records :

After a system crash has occurred, the system consults the log to determine which transaction need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record <*Ti start*> but does not contain either the record <*Ti commit*> or the record <*Ti abort*>
2. Transaction Ti needs to be redone if log contains record <*Ti start*> and either the record <*Ti commit*> or the record <*Ti abort*>

Transaction management with SQL

- In an abstract model of a transaction, a transaction is a sequence of reads, writes, and abort/commit actions.
- SQL provides support for users to specify transaction-level behaviour.

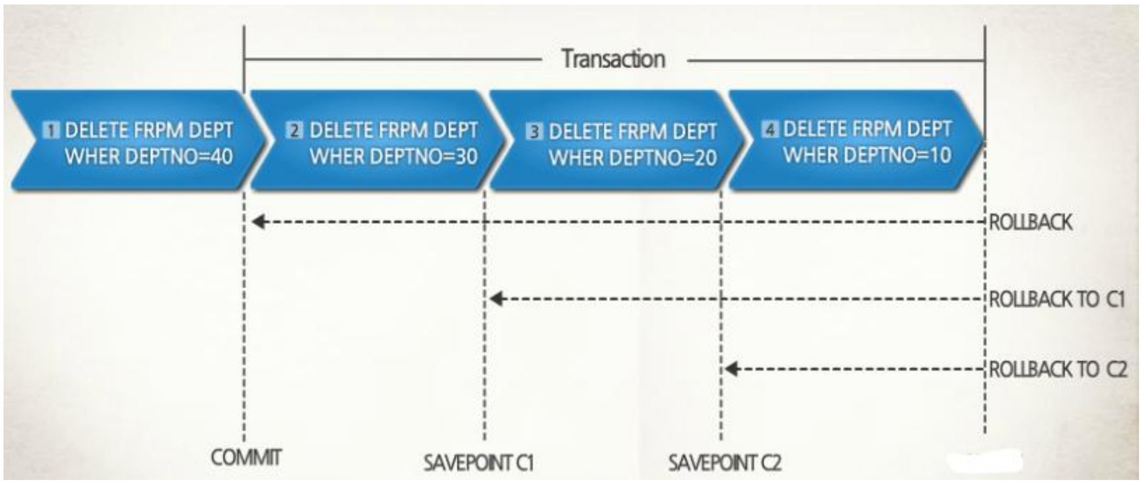
- SQL statements that provide transaction support are
    - **COMMIT**
        - Make changes done in transaction permanent.
        - Syntax : COMMIT;
    - **ROLLBACK**
        - It is an SQL keyword for abort.
        - Rollbacks the state of the database to the last commit point.
        - Syntax : ROLLBACK [to savepoint]
        - It can be used along with a savepoint to rollback upto the savepoint.
    - **SAVEPOINT**
        - Used to specify a point in transaction to which later you can rollback.
        - Syntax : SAVEPOINT name_of_savepoint

- These are considered as Transaction Control Language (**TCL**) of SQL.

- A transaction is automatically started when a user executes a statement that accesses either the database or the catalog, such as SELECT query, an UPDATE command, or a CREATE TABLE statement.
- Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either COMMIT command or a ROLLBACK command.

- Transaction sequence must continue until
  - COMMIT statement is reached.
  - ROLLBACK statement is reached.
  - End of the program is reached.
  - Program is abnormally terminated.

- Example:


Transaction
1 DELETE FRPM DEPT WHER DEPTNO=40
2 DELETE FRPM DEPT WHER DEPTNO=30
3 DELETE FRPM DEPT WHER DEPTNO=20
4 DELETE FRPM DEPT WHER DEPTNO=10
ROLLBACK
ROLLBACK TO C1
ROLLBACK TO C2
COMMIT
SAVEPOINT C1
SAVEPOINT C2

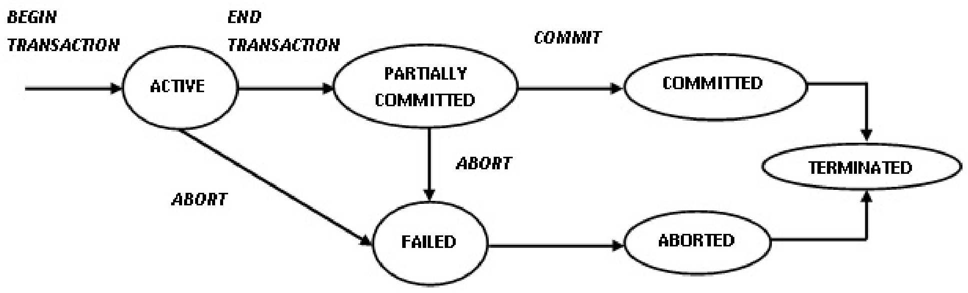
In the above example there are 4 queries.

Commit is executed after query 1, Savepoints C1 and C2 are executed after query 2 and 23.
Here a ROLLBACK undo all operations since commit, ROLLBACK to C1 will undo all operations since SAVEPOINT C1, and to C2 will undo all operations since SAVEPOINT C2.

Transaction States / Transaction State Diagram / Life Cycle of a Transaction

- A transaction goes through many different states throughout its lifetime. These states are known as transaction states.
- Transaction states are as follows-
  1. Active state
  2. Partially committed state
  3. Committed state
  4. Failed state
  5. Aborted state
  6. Terminated state

- The states of a transaction are illustrated in the following figure.



1. Active state-
- This is the first state in the lifecycle of a transaction.
- A transaction is called in an active state as long as its instructions are getting executed from first to last.
- All the changes made by the transaction are being stored in a buffer in main memory.

2. Partially committed state-
- After the last instruction of the transaction gets executed, it enters into a partially committed state.
- After the transaction has entered into this state, the transaction is now considered to be partially committed.
- It is not considered to be fully committed because all the changes made by the transaction are still stored only in the buffer in main memory and not into the database.

3. Committed state-
- After all the changes made by the transaction have been successfully stored into the database, the transaction enters into a committed state.
- NOTE- After a transaction has entered the committed state, it is not possible to roll back the transaction i.e. we can not undo the changes the transaction has made because the system has been now updated into a new consistent state.

4. Failed state-
- When a transaction is being executed or has partially committed and and some failure occurs due to some reason and it is analyzed that the normal execution is now impossible, the transaction then enters into a failed state.

5. Aborted state-
- After the transaction has failed and has entered into a failed state, all the changes made by the transaction have to be undone.
- To undo the changes made by the transaction, it is necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an aborted state.

6. Terminated state-
- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction then finally enters into a terminated state where the transaction life cycle finally comes to an end.

Schedule

- A transaction is seen by the DBMS, as series, or list, of actions.
- The actions that can be executed by a transaction includes
    - reads and writes of db objects.
    - In addition each transaction must specify as its final action either commit or abort.

Denotion of actions of a transaction:

— ***$R_T(O)$***	:	Read operation on database object O by transaction T
— ***$W_T(O)$***	:	Write operation on database object O by transaction T
— ***$Abort_T$***	:	Unsuccessful completion by transaction T (terminate and undo all the actions carried out thus far)
— ***$Commit_T$***	:	Successful completion by transaction T (complete successfully)

- **Def :** A Schedule is a list of actions of several transactions.
- The order in which two actions of transaction T appear in a schedule must be the same as the order in which they appear in T.
- Following schedule shows an execution order for actions of 2 transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- A schedule that contains either an abort or a commit for each transaction is called a ***complete schedule***.
- A schedule in which all actions of a transaction are executed from start to finish, one by one, without any interleaving is called ***serial schedule***.

Disadvantages of Serial Scheduling:

1. It limits concurrency.
2. It causes wastage of CPU time
3. Smaller transactions may need to wait long.

Because of this we will adopt non-serial scheduling.

Concurrent Execution of Transactions

- DBMS executes several transactions concurrently for performance reasons. i.e., DBMS interleaves the actions of different transactions.
- But not all interleaving should be allowed.
- DBMS should allow only safe interleaving of transactions (i.e., serializable schedules)

Motivations / reasons for concurrent execution :

- DBMS executes transactions concurrently for the following performance reasons.
  - Improved throughput
  - Improved response time / Reduced waiting time.
- The average number of transactions completed per unit amount of time is called system throughput. Concurrent execution of transactions improves system throughput. When one transaction is waiting for I/O, another transaction can be executed on CPU, (as CPU and I/O unit can work in parallel)
- Concurrent execution of transactions also improves response time, or the average waiting time taken to complete a transaction.
  Assume that in a serial execution of transactions, if a short transaction started after long transaction, short transaction could get stuck behind a long transaction. Interleaving of these two transactions usually allows the short transaction to complete quickly.

Serializability :

- A serializable schedule over a set **S** of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S.
- i.e., the database instance that results from executing the given schedule is identical to the database instance that results from executing the transaction in some serial order.
  Eg: A serializable schedule

even though, the actions of T1 and T2 are interleaved, the result of this schedule is equal to running T1 and then running T2

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

Anomalies due to interleaved execution:

- DBMS interleaves the actions of several transactions for performance reasons, but not all interleaving are allowed.
- Two actions on the same data object conflict if at least one of them is a write.
- There are three main ways in which a schedule could run against a consistent database and leave it in an inconsistent state.
- Three anomalous situations due to interleaved execution of transactions are
  1. Write – Read conflict (WR)
  2. Read – Write Conflict (RW)
  3. Write – Write Conflict (WW)

1. *Reading uncommitted Data (WR conflicts) :*

- A transaction T2 could read a database object A that has been modified by another transaction T1 which has not yet been committed. Such a read is called 'Dirty Read'.
- ** **Dirty Read** – occurs when a transaction reads a database object that has been modified by another not yet committed transaction. **

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Schedule – 1
Reading uncommitted Data (Dirty Read)

The interleaved execution shown in schedule – 1 can lead to an inconsistency. Here, T2 performed a dirty read on database object A.

Eg: Let A = 500, B = 400
T1 – transfers $100 from A to B
T2 – increments both A and B by 10%.
when T1 and T2 runs alone, they preserve database consistency.

Execution of T1;T2 leads to a database state --> A= 440 B = 550

Execution of T2;T1 leads to a database state --> A= 450 B = 540

but interleaved execution leads to a database state --> A = 440, B = 540

This result is different from any result that we would get by running one of the two transactions first and then the other.
The problem is – The value of A written by T1 is read by T2 before T1 has completed all its changes.

T_1	T_2
R(A)	
A = A - 100	
W(A)	
	R(A)
	A = A * 1.1
	W(A)
	R(B)
	B = B * 1.1
	W(B)
	Commit
R(B)	
B = B + 100	
W(B)	
Commit	

2. *Unrepeatable Reads (RW conflicts) :*

- A Transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress.
- It T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the mean time. It is called an <u>unrepeatable read</u>.
- This situation could not arise in serial execution of two transactions.

- Example : This example illustrates how this situation can lead to inconsistency.

T_1	T_2
R(A)	
A=A-1	
	R(A)
	A=A-1
	W(A)
	Commit
W(A)	
Commit	

Schedule-2 (unrepeatable read)

  - Transactions are trying to book flight tickets.
  - No.of tickets available (A) = 1
  - T1 - read the value of A as 1 and proceeds to book the ticket.
  - T2 – also reads the value of A as 1, booked the ticket (Written)
  - T1 – trying to perform write operation.
  - This leads to incorrect execution of transactions, there by inconsistency.

3. *Overwriting uncommitted data (WW conflicts) :*

- A transaction T2 could overwrite the value of an object A, which has been modified by a transaction T1, while T1 is still in progress.
- This is also called as Blind Write, because here the transaction writes to an object without ever reading the object.
- Example: There are two employees Motu and Patlu. Their salaries must be kept equal. Transaction T1 sets their salaries to $2000 and transaction T2 set their salaries to $1000

T_1	T_2
W(M)	
	W(M)
	W(P)
	Commit
W(P)	
Commit	

Schedule – 3(Blind Write)

- If we execute the transactions serially, then we will get both salaries as $1000 (T1;T2) or as $2000 (T2;T1)
- If we consider the interleaving shown in schedule-3, which consists of blind write leads to inconsistency.
- The above schedule makes Motu salary as 1000 and patlu salary as 2000.

Incorrect Summary problem:

- This **problem** is caused when one of the transactions is executing an aggregate operation on several data items, and other transactions are updating one or more of those data items. This causes a inconsistent database state.
- T2 is updating the value of C and is used in average calculation as one of the argument.

T_1	T_2
R(A)	
R(B)	
R(C)	
Sum=0	
Sum = sum + A	
	R(C)
	C=C+100
	W(C)
	Commit
Sum = sum + B	
Sum = sum + C	
Average()	
Commit	

Phantom Read and Phantom Tuple

When transaction T executes a query that retrieves a set of tuple from a relation satisfying a certain condition, reexecuting the query at a later time but find the retrieved set contains an additional (phantom) tuple that has been inserted by another transaction in the mean time. This is referred as phantom read.

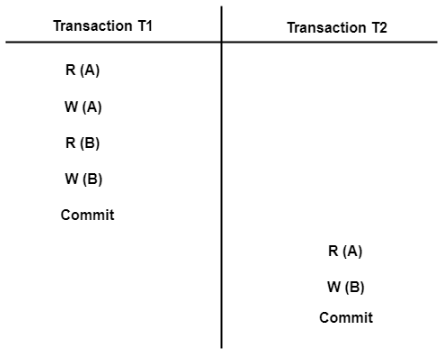
Schedule

- List of actions of several transactions as seen by the Database system is called as schedule.
- The various types of schedules are discussed below.

Serial Schedule:

- A schedule in which actions of different transactions are not interleaved.
- i.e., transactions are executed from start to finish, one by one.
- Serial schedules are always-
    - Consistent
    - Recoverable
    - Cascadeless
    - Strict

In the schedule shown in figure, we have two transactions T1 and T2 where transaction T1 executes first and after it complete its execution, transaction T2 begins its execution. So, this schedule is an example of a **serial schedule**.



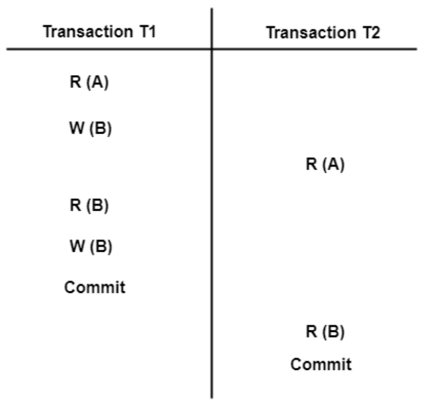
Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

Serial Schedule

Non-Serial Schedule :

- Non-serial schedules are those schedules in which the operations of all transactions are interleaved or mixed with each other.
- Non-Serial schedules are **NOT** always-
    - Consistent
    - Recoverable
    - Cascadeless
    - Strict

In the schedule shown next, we have two transactions T1 and T2 where the operations of T1 and T2 are interleaved. So, this schedule is an example of a **non-serial schedule**.



Transaction T1	Transaction T2
R (A)	
W (B)	
	R (A)
R (B)	
W (B)	
Commit	
	R (B)
	Commit

Non-Serial Schedule

Complete Schedule:

A schedule that contains either an abort or commit for each transaction that listed in it.

T_1	T_2
R(A)	
	R(B)
	W(B)
	Commit
W(A)	
Commit	

Irrecoverable Schedule:

- If in a schedule, a transaction performs a dirty read operation (reads a value from an uncommitted transaction) and commits before the transaction from which it has read the value, then such a schedule is known as an **Irrecoverable schedule**.
- If T1 is aborted because of some reasons, then action of T1 will be rolled back. If T1 is rolled back, then T2 should also be rolled back, which is not possible here. This example schedule is not recoverable.

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
	Commit[1]
Abort[2]	

Irrecoverable Schedule

[1] A committed transaction should never be rolled back.
[2] All actions of aborted transactions are to be undone.

Recoverable Schedule:

Def: A schedule is recoverable if each transaction commits only after all transactions from which it has read has committed.

Or

A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j

- It is a schedule in which no committed transaction needs to be rolled back.
- If in a schedule, a transaction performs a dirty read operation (reads a value from an uncommitted transaction) and its commit operation is delayed till the time, the transaction from which it has read the value commits or roll back, then such a schedule is known as a **Recoverable schedule**.
- By delaying the commit operation of the transaction which has performed a dirty read, it is ensured that it still has a chance to roll back if the transaction which updated the value aborts or roll backs later.

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
Commit	
	Commit

Recoverable Schedule

Cascading abort:

Consider the following schedule

T_1	T_2	T_3
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
		W(X)
Abort		

- It T1 aborts, T2 also aborts, as it is reading data changed by T1 and T3 also aborts.
- It is a strictly avoidable case.
- It is called **cascading abort.**
- Due to cascading abort, cascading rollback happens.
- The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **cascading rollback**.
- Cascading rollback is undesirable, since it leads to the undoing of significant amount of work.

Cascadeless Schedule:

- It is a schedule which avoids cascading rollbacks.
- If transactions in a schedule read only the changes of committed transactions, then it is called cascadeless schedule.

T_1	T_2	T_3
R(X)		
W(X)		
Commit		
	R(X)	
	W(X)	
	Commit	
		R(X)
		W(X)
		Commit

Cascadeless Schedule

Transaction T2 and T3 are reading X, after commit. i.e, T2 and T3 are reading a data item which is not dirty (safe)
This kind of schedule is called cascadeless schedule.

- In cascadeless schedule, a write is allowed after write, but a read is allowed only after commit.

T_1	T_2
R(X)	
W(X)	
	W(X)
Commit	
	R(X)
	Commit

** Every cascadeless schedule is also recoverable schedule.

Strict Schedule:

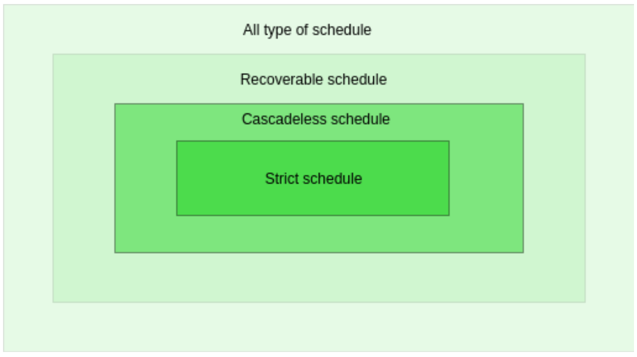
- A schedule is said to be strict if a value written by a transaction T is not read or overwritten by other transaction until T either aborts or commits.

T_1	T_2
R(X)	
	R(Y)
W(X)	
Commit	
	R(X)
	W(X)
	Commit

Strict Schedule

- Strict Schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified objects.
- Transaction T2 can neither perform read/write on X before T1 commits / aborts.
- If you are working on same data item, then strict schedule is serial schedule.
- If you are working with multiple data items strict schedules are not serial schedules.
- All strict schedules are cascadeless schedules.

Following figure illustrations the relation between Strict, cascadeless and recoverable schedules.



Serializable Schedule:

A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of Committed transactions in S.

Or

(in simple terms) A schedule whose result is equal to some serial schedule.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Serializable Schedule

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Non-serializable Schedule

- Serial Scheduling has lot of problems in terms of performance.
- To overcome those, DBMS interleaves the execution of transactions (non-serial Scheduling)
- But some of the orders in non-serial scheduling will give incorrect results.
- Serializability helps us to check whether the order is correct or not. It takes the order, checks whether it is equal to some serial schedule, then we say that the schedule is serializable.
- A schedule **S** of **n** transactions is serializable if it is equivalent to some serial schedule of the same **n** transactions.
- Based on equivalence
  - Conflict Serializability
  - View Serializability

Conflict Serializable schedule:

- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
- Two schedules are said to be conflict equivalent if they involve the (same set of) actions of the same transactions, and they order every pair of conflict actions of two committed transactions in the same way.

Example 1: Determine whether the following 2 schedules are conflict equivalent or not.

T_1	T_2
R(A)	
	R(A)
	R(E)
R(B)	
R(C)	
W(C)	
	R(C)
W(D)	
R(E)	
	R(D)
	R(F)
	W(F)

T_1	T_2
R(A)	
R(B)	
	R(A)
R(C)	
W(C)	
	R(E)
	R(C)
W(D)	
	R(D)
	R(F)
	W(F)
R(E)	

Answer : Both Schedules are conflict equivalent, as they have the same set of operations in both schedules, they order the conflicting operations in the same order.

Example 2: Determine whether the given schedule is conflict serializable or not.

T_1	T_2
R(A)	
	R(A)
	R(E)
R(B)	
R(C)	
W(C)	
	R(C)
W(D)	
R(E)	
	R(D)
	R(F)
	W(F)

Answer: we can say that the given schedule is conflict serializable, if it is conflict equivalent to some serial schedule. The given schedule is conflict equivalent to the serial schedule <T1, T2>. Therefore the given schedule is conflict serializable

Example 3: Check whether the given schedule is conflict serializable or not.

$$S: R_1(X)\ R_2(X)\ W_1(X)\ R_1(Y)\ W_2(X)\ W_1(Y)$$

Answer :

It is hard and time consuming to check whether the given schedule is conflict serializable or not in this way.

There is another method, in which we can construct precedence graph & check for cycles. If the precedence graph is acyclic, we can say that given schedule is conflict serializable.

<u>Precedence Graph (Serializability Graph):</u>

- Precedence graph is the best method to test for conflict Serializability.
- Precedence graph (or Serializability graph) is useful to capture all potential conflict between the transactions in a schedule.
- The precedence graph for a schedule S contains
  - A node for each committed transaction S
  - An arc from T_i to T_j if an action of T_i precedes and conflict with one of the T_j's actions

****if the precedence graph (of a given schedule) is acyclic, we can say that the given schedule is conflict serializable.

**No.of nodes in a precedence graph = no.of transactions in a schedule.

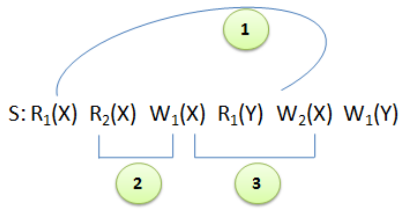
Example : Check whether the following Schedule is conflict serializable or not.

S: $R_1(X)$ $R_2(X)$ $W_1(X)$ $R_1(Y)$ $W_2(X)$ $W_1(Y)$

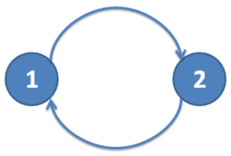
There are 3 conflicting operations.

R1(X) W2(X)
R2(X) W1(X)
W1(X) W2(X)

They were shown in the following picture.



Precedence graph for the above conflict operations is



The precedence graph is cyclic, therefore the given schedule is not conflict serializable.

Check yourself.

1. Check whether the given schedule is conflict serializable or not.
   S: R2(Z) R2(Y) W2(Y) R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z)
2. Check whether the given schedule is conflict serializable or not.
   S: R2(Z) R2(Y) W2(Y) R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z) R2(X) R1(Y) W1(Y) W2(X)

View Serializable Schedule:

- A schedule is called view serializable if it is view equivalent to some serial schedule.

- View equivalence: Two schedules S1 and S2 over the same set of transactions are said to be view equivalent, iff the following conditions are satisfied.
  1. *Initial read :* if Ti reads the initial value of object A in S1, then it must also read the initial value of A in S2.
  2. *Updated read :* If Ti reads a value of A written by Tj in S1, it must also read the value of A written by Tj in S2.
  3. *Final Write operation :* For each data object A, the transaction (if any) that performs the final write on A in S1 must also perform the final write on A in S2.

Note:

- If the given schedule is conflict serializable, then it is surely view serializable.
- If the given schedule is not conflict serializable, then it may or maynot be view serializable.
- No blind write means, not a view serializable schedule.

Example : Check whether the given two schedules are view equivalent or not.

T_1	T_2	T_3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit

Schedule - 1

T_1	T_2	T_3
R(A)		
W(A)		
Commit		
	W(A)	
	Commit	
		W(A)
		Commit

Schedule – 2

These two schedules are view equivalent to each other.
Schedule-1 is view serializable, as it is view equivalent to the serial schedule-2

Check your understanding:

1. Determine whether the following schedule is (1) conflict serializable (2) view serializable.

T_1	T_2
R(X)	
	R(X)
W(X)	
	W(X)

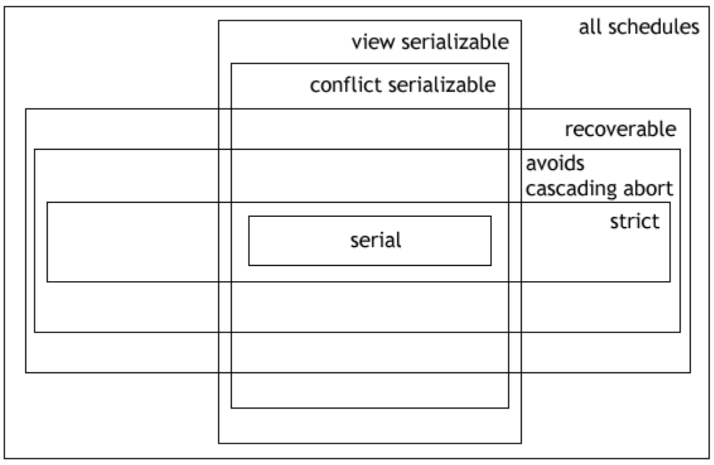
2. Determine whether the following schedule is (1) conflict serializable (2) view serializable

T_1	T_2	T_3
	R(B)	
	W(A)	
R(A)		
		R(A)
W(B)		
	W(B)	
		W(B)

Important points :

- Every conflict serializable schedule is view serializable, but every view serializable schedule need not be conflict serializable.
- Strict 2PL allows only conflict serializable schedules.
- Schedule which is view serializable, but not conflict serializable will have Blind Writes.

The relationship among various schedules is shown below.



Concurrency Control

- Concurrency control is the simultaneous execution of transactions to ensure serializability in multiuser database.
- DBMS executes several transactions concurrently for performance reasons. All interleaving are not safe. DBMS should allow only safe interleavings.
- DBMS must be able to ensure important properties of schedules like serializability and recoverability.
- To ensure these DBMS uses several concurrency control mechanisms, some of them are
    - Lock based protocols,
    - Time-stamp based protocols.

Lock-based concurrency control

- It is the most widely used concurrency control mechanism by DBMS.
- Locking protocol is typically used by DBMS to ensure only serializable and recoverable schedules.
- A lock is a small book keeping object associated with a database object.

- Locking is a procedure used to control concurrent access to data (to ensure serializability of concurrent transactions)
- In order to use a ‘resource’ (table, row, etc) a transaction must first acquire a lock on that resource
- This may deny access to other transactions to prevent incorrect results
- A **locking protocol** is a set of rules to be followed by each transaction to ensure that, even thought actions of several transactions might be interleaved; the net effect is identical to executing all transactions in some serial order.

Lock Granularity:

- The size of the database object being locked is called lock granularity. It can be coarse or fine.
- If a small database object is locked, then that is referred as fine level granularity.
- If a large database object is locked, then that is referred as coarse level granularity.
- Finer granularity improves concurrency.
- Lock granularity indicates the level of access to a database defined by a lock manager.
- There are five different levels of access in locking granularity. They are

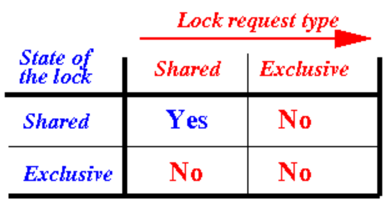
1. **Database level:**
   Only one transaction is allowed to access the entire database at a time.
   It is coarse level of locking.
   It ensures serializability, but decreases concurrency.
2. **Table level**
   Each table allows only one transaction to interact at a time.
   Other transactions are allowed to access other tables in the database.
   It is less restricted than database level locking.
3. **Page level**
   A transaction can lock only stipulated number of rows from one or more tables.
   A page is a virtual table on one or many tables.
4. **Row level**
   Row level locking allows more than one transaction on a table as long as their rows are different.
   Each locked row allows one transaction to interact with it at a time.
   It is less restrictive than the previous ones.
5. **Field level**
   This locking allows concurrency control to put locking on fields.
   Each locked attribute allows only one transaction to interact with at a time.
   It improves concurrency.
   It is most flexible and least restrictive.

Types of locks:

- Any transaction cannot read or write data until it acquires an appropriate lock on it.
- The various types of locks are

1. **Shared lock / Read Lock (S)** : If a transaction wants to perform read operation on a database object, first it has to acquire shared lock on that object.
2. **Exclusive lock / Write Lock (X) :** If a transaction wants to perform write operation on a database object, first it has to acquire exclusive lock on it.

The DBMS grants locks to transactions according to the following table when a transaction request the DBMS for locks.


Lock request type

State of the lock	*Shared*	*Exclusive*
Shared	Yes	No
Exclusive	No	No

Lock compatibility matrix

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

Lock Management:

- **Lock Manager :**
  - The part of the DBMS that keeps track of the locks issued to transactions.
  - It maintains a lock table.
- **Lock Table:**
  - It is a hash table with the data object identifier as the key.
  - A lock table entry for an object contains
    - The number of transactions currently holding a lock on the object,
    - The nature of the lock (shared or exclusive) and
    - A pointer to a queue of lock requests.
- **Transaction table:**
  - It contains details of transactions.
  - Every entry contains a pointer to a list of locks held by the transaction.
  - This list is checked before requesting a lock, to ensure that a transaction does not request the same lock again.

- When a transaction needs a lock on an object, it issues a lock request to the lock manager.
  - If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object.
  - If an exclusive lock is requested and no transaction currently holds a lock on the object, the lock manager grants the lock and updates the lock table entry.
  - Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object.

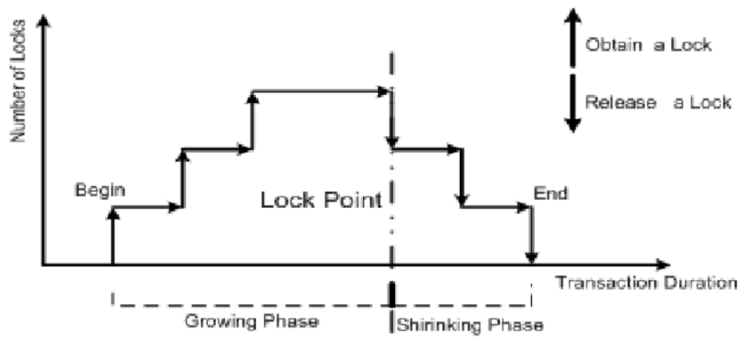
- When a transaction aborts or commits, it releases all its locks.

Two Phase Locking Protocol (2PL):

2PL for concurrency control:

- A transaction is said to follow two phase locking protocol if locking and unlocking can be done in two phases.

- **Growing phase:** new locks on data items can be acquired but none can be released.
- **Shrinking Phase:** existing locks may be released but no new locks can be acquired.



- **Lock point** is the point at which the growing ends (i.e., when the transaction takes final lock it needs to carry out its work)

- 2PL ensures serializability. If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Skeleton transaction implementing 2-PL

	T_1	T_2
1	LOCK-S(A)	
2		LOCK-S(A)
3	LOCK-X(B)	
4		
5	UNLOCK(A)	
6		LOCK-X(C)
7	UNLOCK(B)	
8		UNLOCK(A)
9		UNLOCK(C)

Transaction T1:

Growing Phase	:	steps 1 to 3
Shrinking Phase	:	steps 5 to 7
Lock Point	:	step 3

Transaction T2:

Growing Phase	:	steps 2 to 6
Shrinking Phase	:	steps 8 to 9
Lock Point	:	step 6

- Drawbacks of 2PL are
  - Unnecessary wait due to early locks
  - Deadlocks and starvation is possible.
  - Cascading rollback is possible in 2PL.

1) <u>Unnecessary locks due to early locks</u>

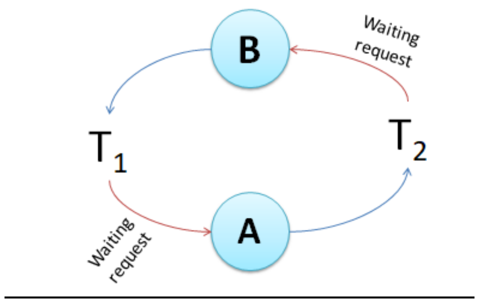
T_1	T_2
L(A)	
L(B)	L(A)
L(C)	

In the above schedule, lock point is where it locked C. From lock point onwards unlocking starts.
Assume that T1 completes its work on data object A, at the very beginning. Even though T2 has to wait for A until shrinking phase of T1 starts and T1 releases A.

2) <u>Deadlock:</u>

- Transaction T1 acquired an exclusive lock on data object B.
- Transaction T2 acquired a shared lock on data object A.
- Transaction T2 requested an exclusive lock on object B, which can't be granted until T1 releases the lock. So it is waiting.
- Transaction T1 requested an exclusive lock on object A, which can't be granted until T2 releases the lock. So it is waiting.
- This is a deadlock.

T_1	T_2
Lock-X(B)	
R(B)	
W(B)	
	Lock-S(A)
	R(A)
	Lock-X(B)
Lock-X(A)	



Therefore, deadlocks may come in 2PL

3) <u>Cascading rollback:</u>
Consider the following schedule. If T1 aborts, then it rollbacks the actions of T2 and T3. Therefore cascading rollback happens in 2PL.

T_1	T_2	T_3
LOCK-X(A)		
R(A)		
W(A)		
UNLOCK(A)		
	LOCK-X(A)	
	R(A)	
	W(A)	
	UNLOCK(A)	
		LOCK-X(A)
		R(A)
abort		

Following are the variants which improves the 2PL

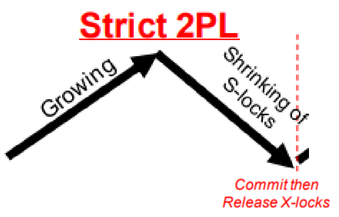
1. Strict 2PL
2. Rigorous 2PL
3. Conservative 2PL

<u>Conservative 2PL:</u>

- This locking protocol acquires locks on all the desired data items before transaction begins its execution.
- There is no growing phase.
- It guarantees serializability.
- It will not guarantee strict schedule, as some transaction can make a dirty read from T, before T has committed.
- It is deadlock free, as it acquires all locks before starting its execution.
- It can have cascading rollback problem.
- Practical implementation of this protocol is difficult, as it needs early prediction.

<u>Strict 2PL:</u>

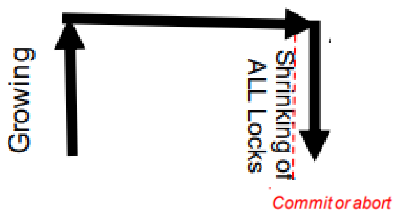
- In this protocol
  - Exclusive locks are only released after commit or abort.
  - Shared locks are released earlier.



- It is practically possible, in fact It is most commonly used 2PL algorithm.
- It guarantees serializability, but it is not deadlock-free.
- As dirty-reads are not allowed, it ensures strict schedules for recoverability.
- It guarantees strict schedules.

<u>Rigorous 2PL:</u>

- It is even more restrictive than strict 2PL.
- In this protocol, exclusive and shared locks are only released after commit or abort.



- It is deadlock free
- It gurantees serializability
- Leads to strict schedules for recoverability.

Problems of locking:

- Problems that may occur when we use locking
    1. Deadlock
    2. Starvation

Deadlock:

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T^1 in the set.
- Hence each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock.

- To prevent deadlocks the time stamp concept is used.

Transaction Timestamp TS(t)

- It is an increasing variable (integer) indicating the age of an a transaction.
- It is a unique identifier assigned to each transaction
- The timestamps are typically based on the time at which the transaction is started
- A larger timestamp indicates a more recent transaction
- If TS(t1) < TS(t2) then t2 is younger than t1

- Two schemes to prevent deadlocks based on timestamps are
    1. Wait – Die
    2. Wait – Wound

Wait – Die

- Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rule followed by Wait – Die scheme is

```
If   TS(Ti) < TS(Tj) then
        Ti is allowed to wait;
Otherwise
        Abort Ti and restart it later with the same timestamp.
```

Wound – Wait

- Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rule followed by Wound – Wait scheme is

```
If   TS(Ti) < TS(Tj) then
        abort Tj and restart it later with the same timestamp;
Otherwise
        Ti is allowed to wait.
```

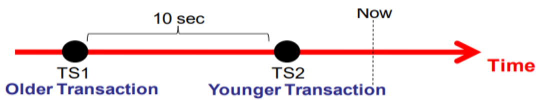
Starvation

- Occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- It occurs if the waiting scheme for locked items is unfair.
- Solutions for starvation:
    - **A fair waiting scheme (FCFS**) – transactions are enabled to lock an item in the order in which they originally requested the lock.
    - **Aging** – Allowing transactions to have priority, and increasing the priority of the transactions the longer it waits.

Concurrency control based on timestamp ordering

What is a Timestamp?

- It is a unique identifier created by the DBMS to identify a transaction.
- TS values are assigned in the order in which the transactions are submitted to the system.
- A timestamp can be thought of as the transaction start time.
- TS(t) – refers to timestamp of T
- If TS(t1) < TS(t2) then t2 is younger than t1



- Concurrency control techniques based on timestamps do not use locks; hence, deadlocks cannot occur.

Timestamps are assigned in two ways

1. **Using current date and time** : The current date or time value of the system clock can be used as timestamp. Here, it ensures that no two timestamp values are generated during the same tick of the clock.
2. **Using a logical counter** : A counter can be used which is incremented each time its value is assigned to a transaction. Transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to 0.

Basic TimeStamp Ordering Alogirthm:

- This algorithm based on timestamps and ensure serializability using the ordering of timestamps
- **Idea** – to enforce the equivalent serial order on the transactions based on their timestamps. Transactions are ordered on the basis of arrival time.
- The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the Timestamp order.

- To do this, the algorithm associates with each database item X, two timestamp values
    1. Read time stamp of X RTS(X)
    2. Write time stamp of X WTS(X)
- **RTS(X)** – the read timestamp of item X is the largest timestamp among all timestamps of transactions that have successfully read item X.
- **WTS(X)** – the write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X.

This algorithm uses the following assumptions:

- A data item X in the database has a RTS(X) and WTS(X) (recorded when the object was last accessed for the given action)
- A transaction T attempts to perform some action (read or write) on data item X on timestamp TS(T)
- Problem: We need to decide whether T has to be aborted or whether T can continue execution.

- This algorithm utilizes Timestamps to guarantee serializability of concurrent transactions.

Time Stamp Ordering Rule :

If Pi(X) and Qj(X) are conflicting operations then
Pi(X) is processed before Qj(X) iff **TS(Ti) < TS(Tj)**

- If the TO rule is enforced in a schedule then the schedule is (conflict) serializable.
- Why? Because cycles are not possible in the Conflict Precedence Graph

Algorithm:

Whenever some transaction T tries to issue R(X) or W(X) operation, it compares TS(T) with RTS(X) and WTS(X) to ensure that the TO of transaction execution is not violated.

Case 1: Whenever a transaction T issues W(X) operation

```
If RTS(X) > TS(T)  or if WTS(X) > TS(T) then
    Abort and rollback T and reject the operation
Otherwise
    Execute W(X) operation of T and
    set WTS(X) = TS(T)
```

Case 2: Whenever a transaction T issues R(X) operation

```
If  WTS(X) > TS(T) then
    Abort and rollback T and reject the operation
If  WTS(X) <= TS(T) then
    Execute R(X) operation of T and
    Set RTS(X) = largest(RTS(X), TS(T)
```

Advantages:

- Schedules are serializable (like 2PL protocols) : Whenever this algorithm detects two conflicting operations, that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. Hence, the schedules produced by basic TO are guaranteed to be conflict serializable.
- No waiting for transaction, thus, no deadlocks!

Disadvantages:

- Schedule may not be recoverable (read uncommitted data)
  - Solution: Utilize Strict TO Algorithm (see next)
- Starvation is possible (if the same transaction is continually aborted and restarted)
  - Solution: Assign new timestamp for aborted transaction

May 2018

1. Define transaction and explain desirable properties of transactions.
2. Compare wait/die with wound/wait scheme.

November 2015

1. Illustrate lost update problem with suitable example.
2. Draw transaction state diagram and describe each state that a transaction goes through during its execution.
3. Explain in detail about timestamp based concurrency control techniques.
4. Briefly discuss about different types of schedules.
5. Discuss about different types of failures.
6. What is 2-phase locking protocol? How does it guarantee serializability?
7. Describe Wait/Die & Wound/Wait protocols.
8. Why the concurrency control is needed? Explain it. [8M]
9. Write and explain optimistic concurrency control algorithm.
10. Write short notes on:
    i) Phantom Record ii) Repeatable Read iii) Incorrect Summary iv) Dirty Read
11. Describe Wait/Die and Wound/Wait deadlock protocols.
12. Explain WAL protocol.

May 2016

1. Explain about deadlocks.
2. Explain the time stamp based protocol for concurrency control in a DBMS. [8M]
3. Explain the ARIES recovery method. When does a system recover from a crash? In what order must a transaction be undone and redone? Why is this order important?

May 2017

1. Define the term ACID properties.
2. Explain read-only, write-only and read-before-write protocols in serializability. [8M]
3. How the use of 2PL would prevent interference between two transactions.

November 2016

1. State and explain two-phase locking protocol.
2. What is transaction? Mention the desirable properties of a transaction. [6M]
3. Discuss about transaction recovery techniques. [10M]
4. What is transaction log? Mention its content.
5. Why concurrency control is needed? Explain the problems that would arise when concurrency control is not provided by the database system.
6. What is serialization? Explain it.
7. Write about the transaction management with SQ L using commit, rollback, and savepoint.
8. Briefly discuss about various lock based mechanisms used in concurrency