# UNIT -2

## Transaction Management and Concurrency Control

# Transaction

- A transaction is a sequence of operations performed (using one or more SQL statements) on a database as a single logical unit of work.

# Transaction Property/ACID property

- A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.
- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability:**

# Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to database are not visible.

—**Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**

- If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

| **Before:** X : 500 | Y: 200 |
|---|---|
| Transaction T | |
| **T1** | **T2** |
| Read (X)<br>X: = X – 100<br>Write (X) | Read (Y)<br>Y: = Y + 100<br>Write (Y) |
| **After:** X : 400 | Y : 300 |

# Consistency

- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction.
- It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.
  Total **before T** occurs = **500 + 200 = 700**.
  Total **after T occurs** = **400 + 300 = 700**.
  Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

# Isolation

- This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state.
- Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.
- This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order. Let **X**= 500, **Y** = 500.

# Durability

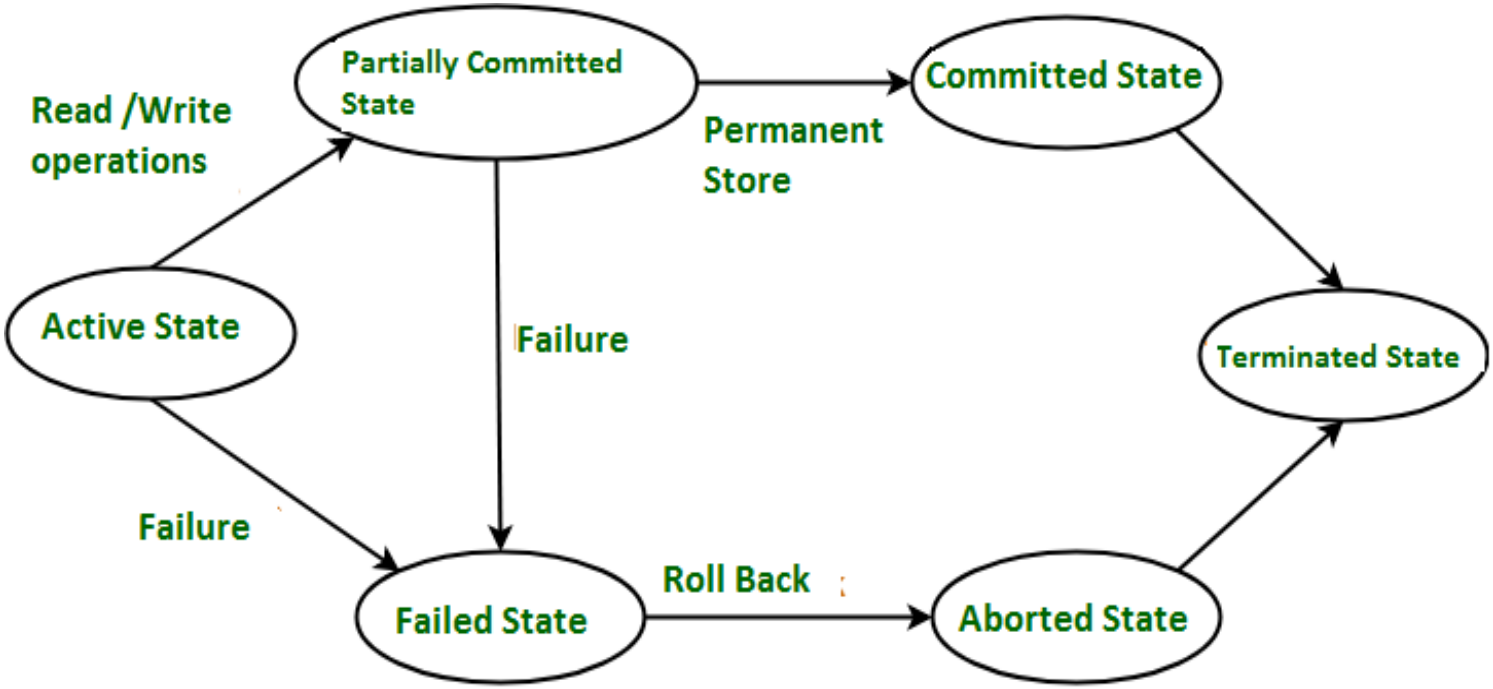
- This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.
- These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

# serializability

- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which **schedules** are serializable.
- A serializable schedule is the one that always leaves the database in consistent state.

- A serializable schedule always leaves the database in consistent state. A **serial schedule** is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.
- A **non-serial schedule** of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

# States of Transaction




Transaction States in DBMS

- **Active State –**
  When the instructions of the transaction are running then the transaction is in active state. If all the ‘read and write’ operations are performed without any error then it goes to the “partially committed state”; if any instruction fails, it goes to the “failed state”.

- **Partially Committed –**
  After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Data Base then the state will change to “committed state” and in case of failure it will go to the “failed state”.

- **Failed State –**
  When any instruction of the transaction fails, it goes to the “failed state” or if failure occurs in making a permanent change of data on Data Base.

- **Aborted State –**
  After having any type of failure the transaction goes from “failed state” to “aborted state” and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

- **Committed State –**
  It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the “terminated state”.

- **Terminated State –**
  If there isn’t any roll-back or the transaction comes from the “committed state”, then the system is consistent and ready for new transaction and the old transaction is terminated.

# The transaction log

- A DBMS uses a transaction log to keep track of all transactions that update the database.
- The information stored in this log is used by the DBMS for a recovery requirement triggered by a ROLLBACK statement, a program's abnormal termination, or a system failure such as a network discrepancy or a disk crash.
- Some RDBMSs use the transaction log to recover a database forward to a currently consistent state.
- After a server failure, for example, Oracle automatically rolls back uncommitted transactions and rolls forward transactions that were committed but not yet written to the physical database.
- While the DBMS executes transactions that modify the database, it also automatically updates the transaction log.

# The transaction log stores:

- A record for the beginning of the transaction.
- For each transaction component (SQL statement):
- The type of operation being performed (update, delete, insert).
- The names of the objects affected by the transaction (the name of the table).
- The “before” and “after” values for the fields being updated.
- Pointers to the previous and next transaction log entries for the same transaction.
  - The ending (COMMIT) of the transaction.

# DBMS schedulers :

- A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.
- Serial Schedule:
- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.
- Non-serial Schedule:
- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.

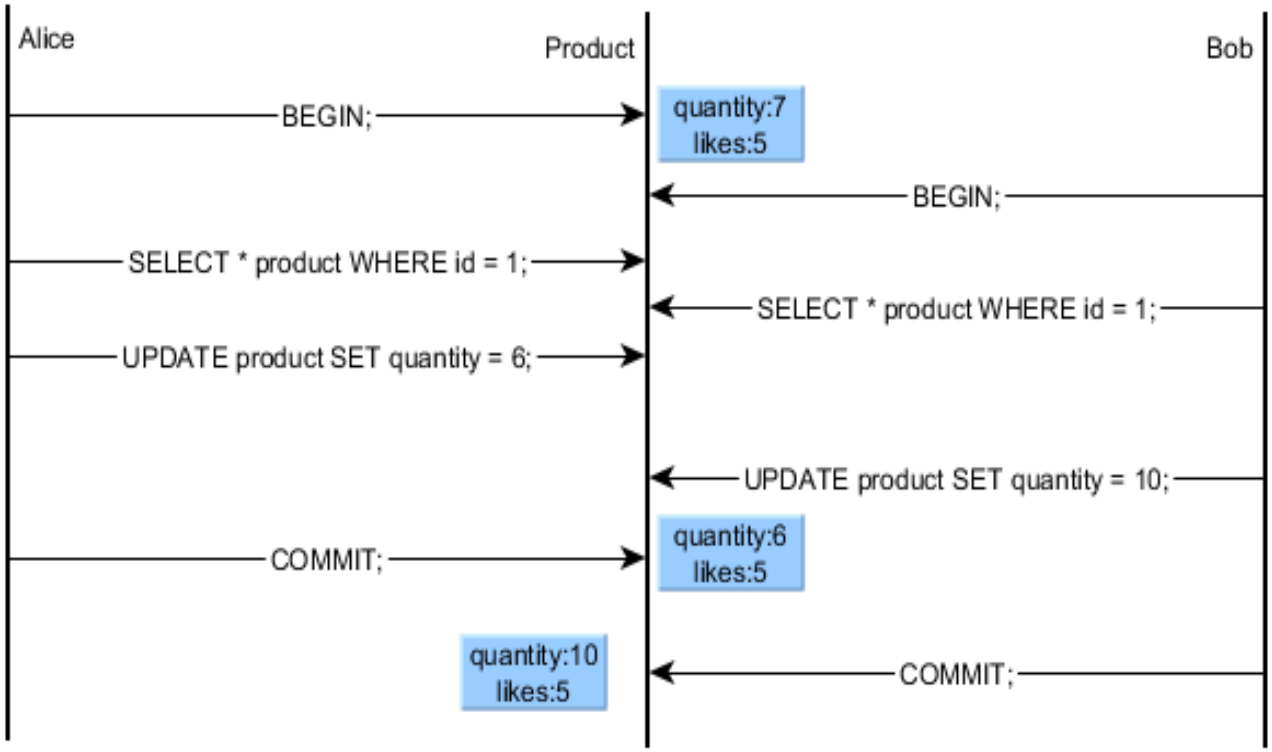
- Serializable schedule:
- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

# Concurrency control

- Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.
- **In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database**. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.
- There are three problems in concurrency control which are explained below:

# 1.Lost Update

- A lost update occurs when two different transactions are trying to update the same column on the same row within a database at the same time.
- Typically, one transaction updates a particular column in a particular row, while another that began very shortly afterward did not see this update before updating the same value itself.
- The result of the first transaction is then "lost," as it is simply overwritten by the second transaction.


Alice
Product
Bob
BEGIN;
quantity:7
likes:5
BEGIN;
SELECT * product WHERE id = 1;
SELECT * product WHERE id = 1;
UPDATE product SET quantity = 6;
UPDATE product SET quantity = 10;
quantity:6
likes:5
COMMIT;
quantity:10
likes:5
COMMIT;

# 2.Uncommitted data

- The phenomenon of uncommitted data occurs when two transactions, T1 and T2, are executed concurrently and the first transaction (T1) is rolled back after the second transaction (T2) has already accessed the uncommitted data—thus violating the isolation property of transactions.

# 3.Inconsistent retrievals or Unrepeatable Read Problem (W-R Conflict)

- *Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.*

- Consider two transactions, $T_X$ and $T_Y$, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

| Time | $T_x$ | $T_y$ |
|---|---|---|
| $t_1$ | READ (A) | — |
| $t_2$ | — | READ (A) |
| $t_3$ | — | A = A + 100 |
| $t_4$ | — | WRITE (A) |
| $t_5$ | READ (A) | — |

UNREPEATABLE READ PROBLEM

- At time t1, transaction $T_X$ reads the value from account A, i.e., $300.
- At time t2, transaction $T_Y$ reads the value from account A, i.e., $300.
- At time t3, transaction $T_Y$ updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction $T_Y$ writes the updated value, i.e., $400.
- After that, at time t5, transaction $T_X$ reads the available value of account A, and that will be read as $400.
- It means that within the same transaction $T_X$, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction $T_Y$, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.
- Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

# Concurrency control protocols

- In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions.

- We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.

- Concurrency control protocols can be broadly divided into two categories −

  **1)Lock based protocols**

  2) Time stamp based protocols

  3) Optimistic protocol

# Locking levels\ lock granularity

- Database level locking
- Table level locking
- Page level locking(disk –bock)
- Row (tuple) level locking
- Attributes(fields) level locking

# Lock-based Protocols

- Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

Binary Locks − A lock on a data item can be in two states; it is either locked or unlocked.

Shared/exclusive − This type of locking mechanism differentiates the locks based on their uses.

- If a lock is acquired on a data item to perform a write operation, it is an exclusive lock.
- Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state.
- Read locks are shared because no data value is being changed.

**There are four types of lock protocols available :**

- Simplistic Lock Protocol
- Pre-claiming Lock Protocol
- Two-Phase Locking 2PL
- Strict Two-Phase Locking

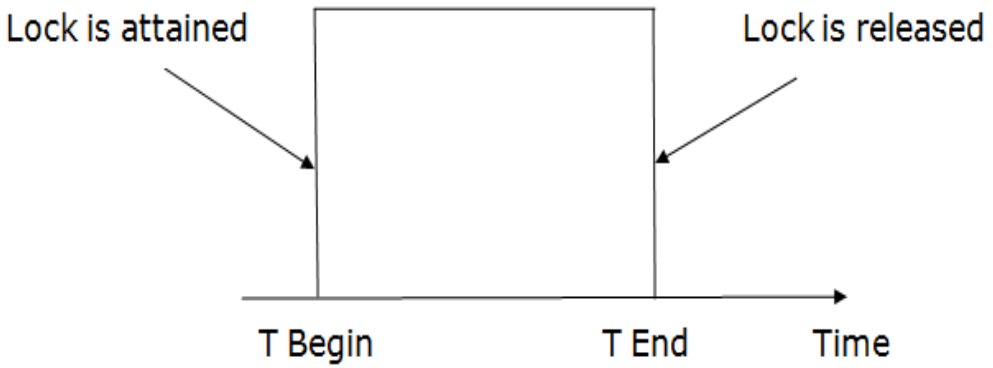
# 1.Simplistic Lock Protocol

- Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

- It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

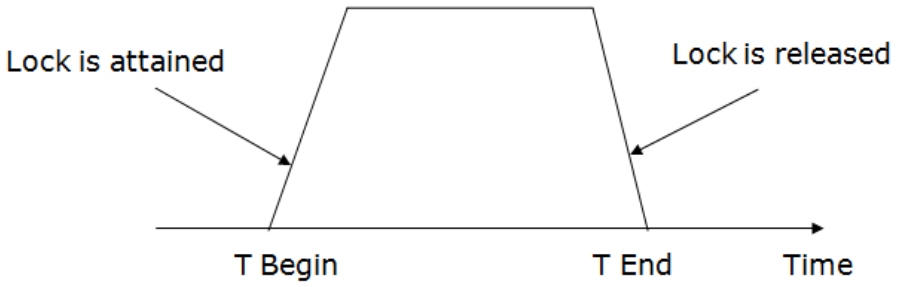
# 2.Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



# 3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction **acquires** all the locks. The third phase is started as soon as the transaction releases its first lock(**growing phase**).
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks(**shrinking phase**).



# Irrecoverable schedule

The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

- The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails.
- Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

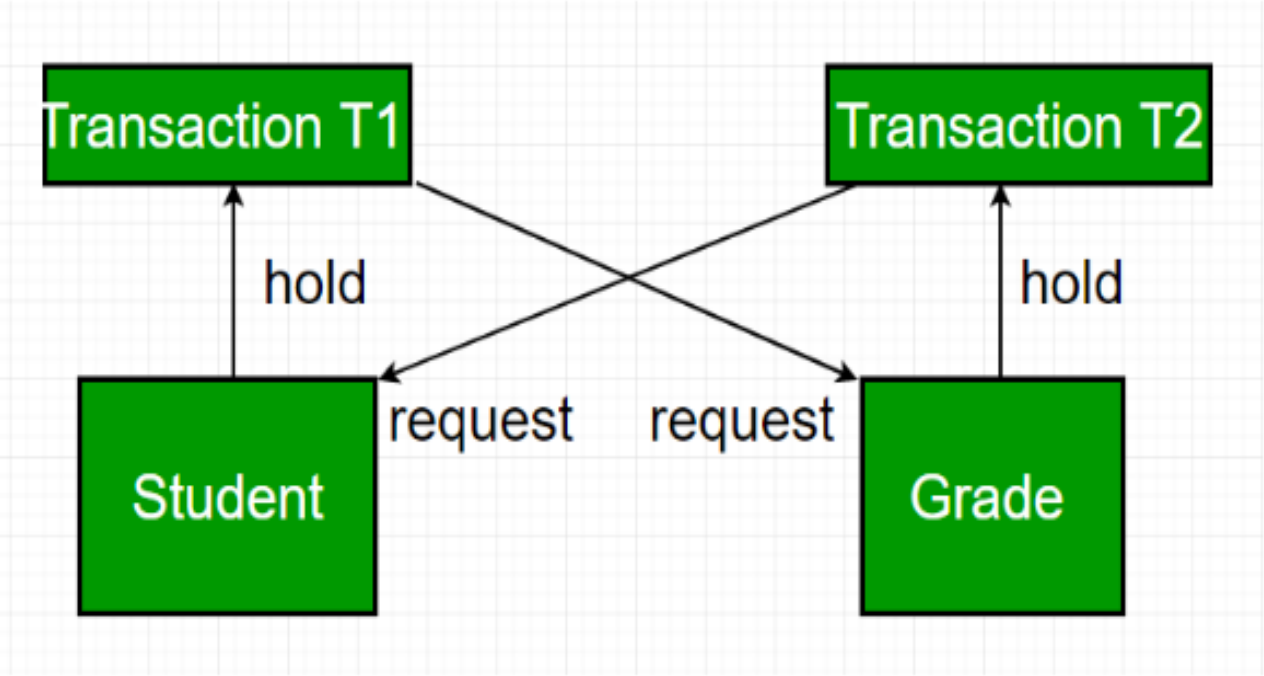
| T1 | T1's buffer space | T2 | T2's buffer space | Database |
|---|---|---|---|---|
| | | | | A = 6500 |
| Read(A); | A = 6500 | | | A = 6500 |
| A = A - 500; | A = 6000 | | | A = 6500 |
| Write(A); | A = 6000 | | | A = 6000 |
| | | Read(A); | A = 6000 | A = 6000 |
| | | A =A + 1000; | A = 7000 | A = 6000 |
| | | Write(A); | A = 7000 | A = 7000 |
| Failure Point | | | | |
| Commit; | | | | |
| | | Commit; | | |

# Recoverable with cascading rollback:

- The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.
- The below Table shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

| T1 | T1's buffer space | T2 | T2's buffer space | Database |
|---|---|---|---|---|
| | | | | A = 6500 |
| Read(A); | A = 6500 | | | A = 6500 |
| A = A - 500; | A = 6000 | | | A = 6500 |
| Write(A); | A = 6000 | | | A = 6000 |
| Commit; | | Read(A); | A = 6000 | A = 6000 |
| | | A =A + 1000; | A = 7000 | A = 6000 |
| | | Write(A); | A = 7000 | A = 7000 |
| | | Commit; | | |

# Deadlock




*Deadlock in DBMS*

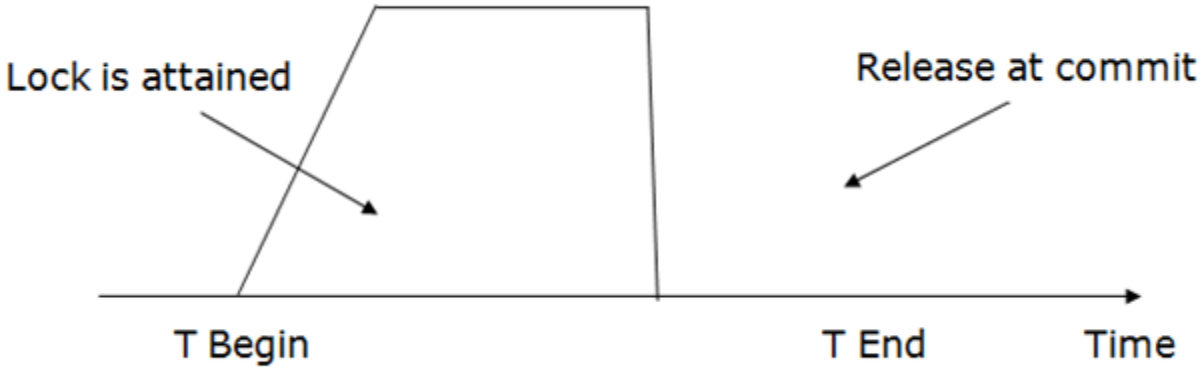
- In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.
- Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.
- **Example –**
- let us understand the concept of Deadlock with an example :
  Suppose, Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table.
- Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.
- Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

# Deadlock Avoidance

- When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database.
- Deadlock avoidance method is suitable for smaller databases whereas the deadlock prevention method is suitable for larger databases. One method of avoiding deadlock is using application-consistent logic.
- In the above given example, Transactions that access Students and Grades should always access the tables in the same order.
- In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins.
- When transaction T2 releases the lock, Transaction T1 can proceed freely. Another method for avoiding deadlock is to apply both row-level locking mechanism and READ COMMITTED isolation level.
- However, It does not guarantee to remove deadlocks completely.

# 4. Strict Two-phase locking (Strict-2PL)

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



# Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

# Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:
- If W_TS(X) >TS(Ti) then the operation is rejected.
- If W_TS(X) <= TS(Ti) then the operation is executed.
- Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:
- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

- **Where,**
  **TS(TI)** denotes the timestamp of the transaction Ti.
- **R_TS(X)** denotes the Read time-stamp of data-item X.
- **W_TS(X)** denotes the Write time-stamp of data-item X.

# Following are the three basic variants of timestamp-based methods of concurrency control:

- Total timestamp ordering :

  - Total Timestamp Ordering The total timestamp ordering algorithm depends on maintaining access to granules in timestamp order by aborting one of the transactions involved in any conflicting access.
  - No distinction is made between Read and Write access, so only a single value is required for each granule timestamp.

- Partial timestamp ordering :

  - Partial Timestamp Ordering In a partial timestamp ordering, only non-permutable actions are ordered to improve upon the total timestamp ordering. In this case, both Read and Write granule timestamps are stored.
  - The algorithm allows the granule to be read by any transaction younger than the last transaction that updated the granule. A transaction is aborted if it tries to update a granule that has previously been accessed by a younger transaction.
  - The partial timestamp ordering algorithm aborts fewer transactions than the total timestamp ordering algorithm, at the cost of extra storage for granule timestamps.

- Multiversion timestamp ordering :

  - Multiversion Timestamp Ordering The multiversion timestamp ordering algorithm stores several versions of an updated granule, allowing transactions to see a consistent set of versions for all granules it accesses.
  - So, it reduces the conflicts that result in transaction restarts to those where there is a Write-Write conflict. Each update of a granule creates a new version, with an associated granule timestamp.
  - A transaction that requires read access to the granule sees the youngest version that is older than the transaction. That is, the version having a timestamp equal to or immediately below the transaction's timestamp.

Advantages and Disadvantages of TO protocol:

TO protocol ensures serializability since the precedence graph is as follows:


**Image:** Precedence Graph for TS ordering

TS protocol ensures freedom from deadlock that means no transaction ever waits.

But the schedule may not be recoverable and may not even be cascade- free.

# Granule timestamp

- Granule timestamp is a record of the timestamp of the last transaction to access it.
- Each granule accessed by an active transaction must have a granule timestamp.
- A separate record of last Read and Write accesses may be kept. Granule timestamp may cause. Additional Write operations for Read accesses if they are stored with the granules.
- The problem can be avoided by maintaining granule timestamps as an in-memory table. The table may be of limited size, since conflicts may only occur between current transactions.
- An entry in a granule timestamp table consists of the granule identifier and the transaction timestamp. The record containing the largest (latest) granule timestamp removed from the table is also maintained.
- A search for a granule timestamp, using the granule identifier, will either be successful or will use the largest removed timestamp.

# Conflict resolution in timestamp

- To deal with conflicts in timestamp algorithms, some transactions involved in conflicts are made to wait and to abort others.

  Following are the main strategies of conflict resolution in timestamps:

  **WAIT-DIE:** The older transaction waits for the younger if the younger has accessed the granule first.

  The younger transaction is aborted (dies) and restarted if it tries to access a granule after an older concurrent transaction.

  **WOUND-WAIT:** The older transaction pre-empts the younger by suspending (wounding) it if the younger transaction tries to access a granule after an older concurrent transaction.

  An older transaction will wait for a younger one to commit if the younger has accessed a granule that both want.
- The handling of aborted transactions is an important aspect of conflict resolution algorithm. In the case that the aborted transaction is the one requesting access, the transaction must be restarted with a new (younger) timestamp. It is possible that the transaction can be repeatedly aborted if there are conflicts with other transactions.
  An aborted transaction that had prior access to granule where conflict occurred can be restarted with the same timestamp. This will take priority by eliminating the possibility of transaction being continuously locked out.

# *Drawbacks of Time-stamp*

- Each value stored in the database requires two additional timestamp fields, one for the last time the field (attribute) was read and one for the last update.
- It increases the memory requirements and the processing overhead of database.

# Optimistic Methods

**Validation phase** is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

- **Read phase**.
- **Validation phase**
- **Write phase**

# Read phase :

- In a Read phase, the updates are prepared using private (or local) copies (or versions) of the granule.
- In this phase, the transaction reads values of committed data from the database, executes the needed computations, and makes the updates to a private copy of the database values.
- All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.
- It is conventional to allocate a timestamp to each transaction at the end of its Read to determine the set of transactions that must be examined by the validation procedure.
- These set of transactions are those who have finished their Read phases since the start of the transaction being verified

# Validation or certification phase :

- In a validation (or certification) phase, the transaction is validated to assure that the changes made will not affect the integrity and consistency of the database.
- If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted, and the changes are discarded.
- Thus, in this phase the list of granules is checked for conflicts. If conflicts are detected in this phase, the transaction is aborted and restarted.
- The validation algorithm must check that the transaction has
  1.Seen all modifications of transactions committed after it starts.
  2.Not read granules updated by a transaction committed after its start.

# Write phase :

- In a Write phase, the changes are permanently applied to the database and the updated granules are made public.
- Otherwise, the updates are discarded and the transaction is restarted.
- This phase is only for the Read-Write transactions and not for Read-only transactions.

# *Advantages of Optimistic Methods for Concurrency Control :*

- This technique is very efficient when conflicts are rare. The occasional conflicts result in the transaction roll back.
- The rollback involves only the local copy of data, the database is not involved and thus there will not be any cascading rollbacks.

# *Problems of Optimistic Methods for Concurrency Control :*

- Conflicts are expensive to deal with, since the conflicting transaction must be rolled back.
- Longer transactions are more likely to have conflicts and may be repeatedly rolled back because of conflicts with short transactions.

# *Applications of Optimistic Methods for Concurrency Control :*

- Only suitable for environments where there are few conflicts and no long transactions.
- Acceptable for mostly Read or Query database systems that require very few update transactions

# Database recovery management

# Transaction failure

- A transaction has to abort when it fails to execute or when it reaches a point from where it can’t go any further. This is called transaction failure where only a few transactions or processes are hurt.

# Transaction failure causes

- **Logical errors** – Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** – Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction

# Failure

- System Crash or failure:

  There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

  Examples may include operating system errors.
- Disk Failure:

  In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

  Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

# Log-based Recovery

- Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.
- Log-based recovery works as follows −

  The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.
- $<T_n$, Start> When the transaction modifies an item X, it write logs as follows −
- $<T_n$, X, $V_1$, $V_2$> It reads $T_n$ has changed the value of X, from $V_1$ to $V_2$.
- When the transaction finishes, it logs −
- $<T_n$, commit>

- The database can be modified using two approaches −
- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.
- Shadow paging: this is method where all the transactions are executed in the primary memory.

  sOnce all the transactions completely executed ,it will be updated to the databse.

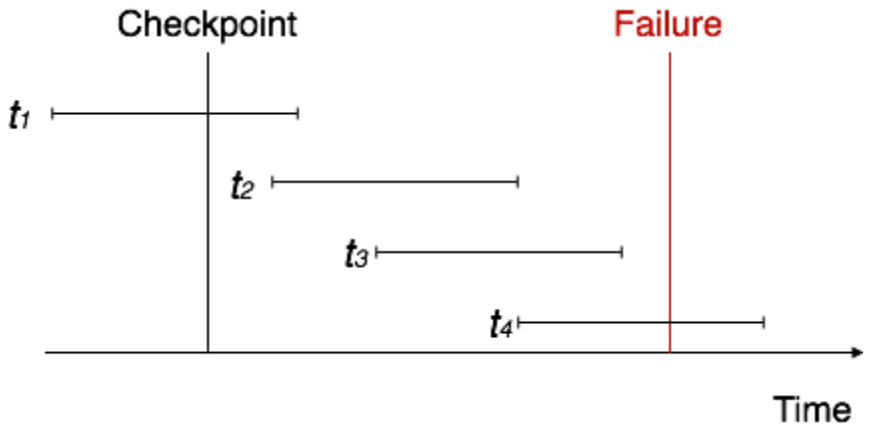
# Recovery with Concurrent Transactions

- When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering.
- To ease this situation, most modern DBMS use the concept of 'checkpoints'.

# Checkpoint

- Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system.
- As time passes, the log file may grow too big to be handled at all.
- Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.
- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

- When a system with concurrent transactions crashes and recovers, it behaves in the following manner –
-



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with $<T_n$, Start> and $<T_n$, Commit> or just $<T_n$, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with $<T_n$, Start> but no commit or abort log found, it puts the transaction in undo-list.
- All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

# Storage Structure

- We have already described the storage system. In brief, the storage structure can be divided into two categories −
- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.