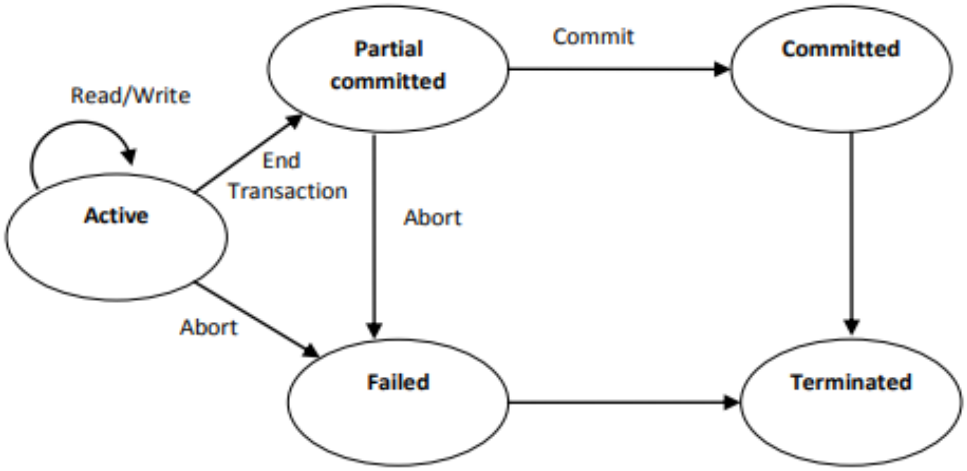
Transaction

- A transaction can either be a single database operation or series of database operation.
- It transforms a database from its one consistent state to another state. During the transaction, intermediate state may be consistent or not but final state must be a consistent state.
- A transaction is a **logical unit** of work that contains one or more SQL statements.
- A transaction is an **atomic** unit.
- A database transaction must be **atomic**, meaning that it must be either entirely **completed** or **aborted**.
- A transaction passes through various states during its execution.
- Following are the different states in transaction processing in database.



State transition Diagram

Active

- This is the initial state. The transaction stay in this state while it is executing.
- Various read and write operations are performed on database.

Partially Committed

- This is the state after the final statement of the transaction is executed.

Failed

- After the discovery that normal execution can no longer proceed.

Aborted (Failed)

- The state after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Aborted transaction can be restart later either automatically or manually.

Committed

- The state after successful completion of the transaction. It can be ensure that it will never be aborted

Terminated

- Transaction enters in this state either from committed or from aborted state.
- Information about the transaction is stored in the system tables.

Transaction Properties

- Transaction remains in consistent state for that it must have following four properties:
    1) **A**tomicity
    2) **C**onsistency
    3) **I**solation
    4) **D**urability
    - These properties are called **ACID** property of a transaction.

ACID property of transaction

Atomicity

- This property requires that all the operations of a transaction are **executed** or **not a single** operation is executed.
- Means, a transaction must be completely **successful** or **completely fail** without doing anything.
- **For** example consider below transaction to transfer Rs. 50 from account A to account B:
    1. **read**(*A*)
    2. *A* := *A* – 50
    3. **write**(*A*)
    4. **read**(*B*)
    5. *B* := *B* + 50
    6. **write**(*B*)
- In above transaction if Rs. 50 is deducted from account A then it must be added to account B.

Consistency

- A transaction must transform the database form **one consistent state** to **another consistent state**. Means our database must remain in **consistent state** after execution of any transaction.
- In above example total of **A** and **B** must remain **same** before and after the execution of transaction.

Isolation

- All transactions execute concurrently are **isolated** from one another. Means, all transaction must execute independently as if it is only transaction execute currently.
- Updates on data made by one transaction cannot be seen by other transactions until that **transaction commits**. Thus, transactions do not **interfere** with each other.
- In above **example** once your transaction start **from step one** its result should not be access by any other transaction **until last step** (step 6) is completed.

Durability

- Once a transaction is **committed**, changes made by that transaction **cannot be lost** even if the **system fails**.
- Once your transaction **completed up to step 6** its result must be stored **permanently**. It should not be removed if system fails.

Transaction Log (Journal)

- Transaction log is a **record** of all transactions and changes made to the database. Also referred as transaction **journal**.
- Transaction log is kept on disk. So, it is not affected by any type of failure except disk failure.
- This record is used to recover database from failures that affects transactions.
- DBMS automatically update the transaction log while executing transactions that modify the database. This results in the **processing overhead** of a DBMS and increase execution time.
- The **log** is written before any changes are made to the database. It is called **write-ahead strategy**. This helps to track changes made to the database in case of failure.

Concurrency control

- When more than one user is accessing same data at the same time then it is known as **concurrent access**.
- The technique used to **protect** data when **multiple users** are accessing it **concurrently** is called as **concurrency control**.

Need/problem of concurrency control

- If transactions are executed serially, i.e., sequentially with no overlap in time, no transaction concurrency exists.
- However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur.
- The main three problems of concurrency control are as given below:
    1) The Lost Update Problem
    2) The Dirty Read Problem
    3) The Inconsistent Retrieval Problem

1) The Lost Update Problem:

- This problem indicate that if two transactions **T1** and **T2** both read the same data and update it then effect of first update will be **overwritten** by the second update.

Example:

- In given below figure shows operations performed by two transaction, **Transaction- A and Transaction-B** with respect to time.

Transaction – A	Time	Transaction - B
--	t0	--
Read X	t1	--
--	t2	Read X
Update X	t3	--
--	t4	Update X
--	t5	--

- At time **t1**, Transaction-A reads value of X.
- At time **t2**, Transaction-B reads value of X.
- At time **t3**, Transaction-A updates value of X on the basis of the value seen at the time t1.
- At time **t4**, Transaction-B updates value of X on the basis of the value seen at the time t2.
- So, update of Transaction-A is lost at time t4 because Transaction-B overwrites it without looking at its current value.
- This type of problem is called **the Update lost problem** and this type of situation is called **Race Condition** because both transaction do race for update data by overwriting updates of one another.

2) The Dirty Read Problem (Uncommitted Dependency):

- The dirty read arises when one transaction updates some item and then fail due to some reason.
- This updated item is retrieved by another transaction before it is changed back to the original value.

Example:

- In given below figure shows operations performed by two transaction, **Transaction- A and Transaction-B** with respect to time.

Transaction – A	Time	Transaction - B
--	t0	--
--	t1	Update X
Read X	t2	--
--	t3	Rollback
--	t4	--

- At time **t1**, Transaction-B updates value of X.
- At time **t2**, Transaction-A reads value of X
- At time **t3**, Transaction-B rollback. So, it changes the value of X back to its state prior to time t1.
- Now, Transaction-A has value which is never become a part of the consistent database.
- This type of problem is called **the Dirty Read Problem**. Because one transaction reads a dirty value which is not committed.

3) The Inconsistent Retrieval Problem:

- The inconsistent retrieval problem arises when one transaction retrieves data to use in some operation.
- But before it can use this data another transaction update that data and commits.
- So, this change will be hidden from first transaction and it will continue to use previous retrieved data. This problem is also known as **Inconsistent Analysis Problem**.

Example:

- In given below figure shows two transaction operating on three accounts.
- **Transaction-A** is summing all balances and
- **Transaction-B** is transferring an amount 50 from **Account-3** to **Account-1**.

Account – 1	**Account – 2**	**Account – 3**
Balance = 200	**Balance = 300**	**Balance = 100**

Transaction – A	Time	Transaction - B
--	t0	--
Read Balance of Acc – 1 Sum <- 200	t1	--
Read Balance of Acc – 2 Sum <- Sum + 300	t2	--
--	t3	Read Balance of Acc – 3
--	t4	Update Balance of Acc – 3 100 -> 100 – 50 -> 50
--	t5	Read Balance of Acc - 1
--	t6	Update Balance of Acc – 1 200 -> 200 + 50 -> 250
--	t7	Commit
Read Balance of Acc – 3 Sum <- Sum + 50 = 550	t8	--

- Here, the result produced by **Transaction – A** is **550**, which is incorrect.
- So, if this result is written in database then database will be in inconsistent state because **actual** sum is **600**.

- In this case, **Transaction – B** commits all of its updates before **Transaction – A** reads Balance of **Account – 3**.

Degree of Consistency

There are **four level** of transaction consistency as given below:

1) Level 0 Consistency
2) Level 1 Consistency
3) Level 2 Consistency
4) Level 3 Consistency

1) Level 0 Consistency:

- Level 0 transactions are **unrecoverable**.
- The transaction **T** does not **overwrite** other transaction's **dirty** (Uncommitted) Data.

2) Level 1 Consistency:

- Level 1 transaction is **recoverable** from system failure.
- Transaction **T** must be in **level 0** consistency and it does not make any of its **update visible** before it **commits**.

3) Level 2 Consistency:

- Level 2 transactions consistency isolates from the updates of other transactions.
- Transaction **T** must be in **level 1** consistency and does not **read dirty** data of other transactions.

4) Level 3 Consistency:

- Transaction **T** must be in **level 2** consistency and other transactions do not **dirty** any data read by transaction **T** before **T** completes.

Permutable Actions

- A pair of action is permutable if every execution of A_i followed by A_j has the same result as the execution of A_j followed by A_i on the same record or page.
- For the action Read and Write, there are following possibilities:
  - **Read -Read:** Permutable
  - **Read-Write**: Not permutable. Because result is different depending on whether read is first or write is first.
  - **Write-Write:** Not Permutable. Because second write is always overwrites the first write.

Schedule

- A **chronological** execution sequence of transaction is called **schedule**.

  Example:

  - Here, **Transaction-A** reads & writes **X** and **Transaction-B** reads & writes **Y** and then again **Transaction-A** reads & writes **Z**.

Transaction-A	Time	Transaction-B
Read X	t0	--
Write X	t1	--
--	t2	Read Y
--	t3	Write Y
Read Z	t4	--
Write Z	t5	--

Simple Schedule

❖ **Transaction Conflicts:**

- If the transaction **T1** and **T2** access **unrelated** data then there is **no conflict**. Also order of operation cannot be affect the final result.
- If the transaction **T1** and **T2** access **related** data then conflict is possible among operations. Also order of operation can affect the final result.

❖ **Scheduler :**

- A scheduler is an in-built module of DBMS software which determines the **correct order** of execution of operations of multiple transactions.
- It ensures that the CPU is utilized in efficient way.

❖ **Types of Schedules :**

- Schedules can be classified as given below:
  1) Complete Schedule
  2) Non-Complete Schedule
  3) Serial Schedule
  4) Non-Serial Schedule

1) Complete Schedule:

- If a schedule contain either **COMMIT** or **ROLLBACK** actions for each transaction. It is known as **Complete Schedule**.

2) Non-Complete Schedule:

- If a schedule does not contain either COMMIT or **ROLLBACK** actions for each transaction. It is known as **Non-Complete Schedule**.

3) Serial Schedule:

- If actions of concurrent transactions are **not interleaved** the schedule is called non-serial schedule.

OR

- Transactions are executed one by one without any interleaved operations from other transactions.

Advantage: Correctness of results is preserved.

Disadvantage: Low CPU utilization results in inefficient processing.

4) **Non-Serial Schedule:**
   - If actions of concurrent transactions are **interleaved** the schedule is called non- serial schedule.

     Advantage: Better CPU utilization results in efficient processing.
     Disadvantage: A schedule may generate undesirable result if enough care is not taken.

Serializable Schedules

- A schedule **S'** is serializable schedule if there exists some **serial** schedule **S** such that **S'** is equivalent to **S**.
- The result produced by a **serializable schedule** is same as if the transactions are executed serially.
- The **purpose** of serializable schedule to improve the performance of the system.
- It is important **in multi-user** where several transactions are likely to be executed **concurrently**.

Rules of Serializability

- A transaction mainly involves two operations: **Read** and **Write.**
- The rules are given below:
  - If two transaction $\mathbf{T_1}$ and $\mathbf{T_2}$ only read a data item, they do not conflict and the order is **not important**.
  - If two transaction $\mathbf{T_{1\ and}}$ $\mathbf{T_2}$ either read or write completely separate data items. They do not conflict and the order is **not important**.
  - If one transaction $\mathbf{T_1}$ writes a data item and another transaction $\mathbf{T_2}$ either read or write the same data item then the order of **execution is important**.

Locking Methods for Concurrency control

- A **lock** is a variable associated with the data item which controls the access of that data item.
- Lock prevents access of the data item to second transaction until first transaction has completed the use of that data item.

Lock Granularity

- The size of data item, chosen as the unit of protection by a concurrency control technique is called granularity.
- A lock granularity indicates **level of lock** to use.
- Different level of lock are given below:
  1) Database Level
  2) Table Level
  3) Page Level
  4) Row Level
  5) Attribute Level

1) **Database Level Locking:**
   - Locks the entire database.
   - **Advantage:** Suitable for batch processing.
   - **Disadvantage:** Not suitable for multi user DBMSs.
2) **Table Level Locking:**
   - Lock the entire table.
   - **Advantage:** Less restrictive than database level lock.
   - **Disadvantage:** It can cause traffic jam when many transactions are waiting to access same table. Not suitable for multi-user DBMSs.
3) **Table Level Locking:**
   - Lock the entire disk-page.
   - A page has a fix size such as 4K, 8K, 16K and so on.
   - **Advantage:** Most suitable for multi-user DBMSs.

4) **Row level Locking:**
   - Lock the particular row.
   - **Advantage:** Improve availability of data.
   - **Disadvantage:** Management of row level locks requires high overhead cost.
5) **Attribute Level Locking:**
   - Lock the particular attribute.
   - Least restrictive than other locks.
   - **Advantage:** Provide the most flexible multi-user data access.
   - **Disadvantage:** Management of attribute level locks requires high overhead cost.

Lock Types

- DBMS mainly uses following type of locking techniques:
  1) Binary Locking
  2) Shared/Exclusive (or Read/Write) Locking
  3) Two-Phase Locking (2PL)

1) **Binary Locking:**
   - Binary lock can have two states:
     1. **Locked (or '1')**
     2. **Unlocked (or '0')**
   - If a database object is locked then no other transaction can use that object.
   - This technique use two operations as given below:

     Lock (X): To lock the data item **X**.

     Unlock (X): To unlock(release) the data item **X**.
   - If any transaction need to access data item **X** then it calls **Lock (X)**. Means **Lock(X) = 1** & another transaction need to access same data item **X**, it needs to wait.
   - When transaction completes its operation in **X**, it calls **Unlock (X)**. It sets **Lock(X) = 0.**
   - **Advantage:** Easy to implement.
   - **Disadvantage:** Two simultaneous Read can be performed on same data but this technique doesn't allow such type of concurrent access.

2) Shared/Exclusive Locking:

- This technique uses two different type of locks:
  I. Shared Locks
  II. Exclusive Locks

I. Shared Locks:

- These locks are also referred as **Read** locks and denoted by '**S'**.
- If any transaction has obtained shared lock on data item X then **it can read X** but **can't write X**.
- Multiple **shared lock** can be placed simultaneously on a data item.

II. Exclusive Locks:

- These lock are referred as **Write** locks and denoted by '**X**'.
- If any transaction has obtained exclusive lock on data item X then **it can read X** as well as **write X**.
- Only **one exclusive lock** can be placed on a data item at a time.

- **Advantage:** Provide optimal concurrency.
- **Disadvantage:** More complex to implement compared to Binary Locking.

3) Two-Phase Locking

- The Two Phase Locking Protocol defines the rules of how to obtain the locks on a data item and how to release the locks.
- Two phase locking (2PL) is a concurrency control method that guarantees **serializability**.
- The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.
  I. Growing Phase
  II. Shrinking Phase

I. Growing Phase

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

II. Shrinking Phase

- In this phase the transaction **may release** locks, but **cannot obtain any new lock**.
- After Lock Point has been reached, the transaction enters the shrinking phase.
- The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point.
- Initially the transaction is in growing phase, that is the transaction acquires locks as needed. Once the transaction releases lock, it enters the shrinking phase and no more lock request may be issued.
- **Upgrading** of lock is **not possible** in **shrinking phase**, but it is **possible** in **growing phase**.
- Below given table shows **Two Phase Locking Technique:**

Time	Transaction	Remarks
t0	Lock –X (A)	# Obtain Exclusive lock on A.
t1	Read A	# Read value of A.
t2	A = A -100	# Subtract 100 from A.
t3	Write A	# Write a new value of A.
t4	Lock-X (B)	#Obtain Exclusive lock on B.
t5	Read B	# Read value of B.
t6	B = B +100	# ADD 100 to B.
t7	Write B	# Write a new value of B.
t8	Unlock (A)	# Release lock on A.
t9	Unlock (B)	# Release lock on B.

- There are two different versions of the Two Phase Locking
    1. Strict Two Phase Locking
    2. Rigorous Two Phase Locking

1) Strict Two Phase Locking :

- In this protocol, a transaction may release **all the shared locks** after the Lock Point has been reached, but it **cannot release any of the exclusive locks** until the transaction **commits**.
- This solves the dirty read problem.

2) Rigorous Two Phase Locking :

- In Rigorous Two Phase Locking Protocol, a transaction is **not allowed to release any lock** (either shared or exclusive) until it **commits**.
- Here transaction can be serialized in the order in which they commit.

Deadlocks

- A set of transaction is **deadlocked**, if each transaction in the set is waiting for a lock held by some other transaction in the set.
- All the transaction will continue to **wait forever**.

Example:

- **Transaction-A** has obtain lock on **X** and is waiting to obtain lock on **Y**. While, **Transaction-B** has obtain lock on **Y** and is waiting to obtain lock on **X**. But none of them can execute further.

Transaction-A	Time	Transaction-B
--	t0	--
Lock (X)	t1	--
--	t2	Lock (Y)
Lock (Y)	t3	--
Wait	t4	Lock (X)
Wait	t5	Wait
Wait	t6	Wait

Deadlock Situation

Deadlock Detection and Deadlock Prevention:

1. **Deadlock Detection:**
   - This technique allows deadlock to occur but then it detects it and to solve it.
   - If a deadlock is detected, one of the transactions involved in deadlock cycle is aborted. Other transactions continue their execution.
   - An aborted transaction is rollback and restarted.
2. **Deadlock Prevention:**
   - This technique prevents a **deadlock to occur**.
   - It requires that all transactions locks, all data items they need in advance.
   - But if any of the locks cannot be obtained, a transaction will be aborted. All the locks obtained are released and a transaction will be rescheduled.
   - This technique ensures that a transaction **never needs to wait** for any lock during its execution time period.

Time-stamp Method for Concurrency Control

- A **time-stamp** is a unique identifier used to identify the relative starting time of a transaction. This method uses either **system time** or **logical counter** to be used as a **time-stamp**.
- A **time-stamp** for transaction **T** is denoted by **TS(T)**.
- **To implement** this time stamping, following two time-stamp values are associated with each data item.
  1. **W-Timestamp (X):**
     - **Write** time-stamp of data-item **X** is denoted by **W-timestamp(X)**.
     - It specifies the **largest timestamp** of any transaction that execute **write (X)** successfully.
  2. **R-Timestamp (X):**
     - **Read** time-stamp of data-item **X** is denoted by **R-timestamp(X)**.
     - It specifies the **largest timestamp** of any transaction that execute **Read (X)** successfully.
- The **timestamp ordering protocol** ensures that any conflicting **Read** and **Write** operations are **executed** in **timestamp order**. This method operates as follow:

  1. **If a transaction Ti issues Read (X) operation:**
     a. **If TS(Ti) < W-timestamp (X)**
        - ✓ Then, **Ti** needs to **read** a value of **X** that was already **overwritten**.
        - ✓ Hence, the **read** operation is **rejected**, and **Ti** is rolled back.

     b. **If TS(Ti) ≥ W-timestamp (X)**
        - ✓ Then, the **read** operation is executed, and **R-timestamp(X)** is set to the **maximum** of **R-timestamp(X) and TS(Ti)**.
  2. **If a transaction Ti issues Write (X) operation:**
     a. **If TS(Ti) < R-timestamp(X)**
        - ✓ Then, the value of **X** that **Ti** is producing was needed previously, and the system assumed that that value would never be produced.
        - ✓ Hence, the **write** operation is **rejected**, and **Ti** is rolled back

     b. **If TS(Ti) < W-timestamp(X)**

- ✓ Then, **Ti** is attempting to **write** an obsolete (out-dated) value of X.
- ✓ Hence, this **write** operation is **rejected**, and **Ti** is rolled back.

c. Otherwise, the **write** operation is **executed**, and **W-timestamp(X)** is set to **TS(Ti)**.

Advantages:

- Ensures serializability among multiple transactions running concurrently.
- Provide freedom from deadlock as no any locks are used in this method.

Disadvantages:

- **Starvation** for long transaction is possible, if a sequence of conflicting short transactions causes repeated restarting of the long transaction.
- Increase **memory requirement** and **process overhead.** Because two additional fields required in database to store **R-Timestamp** and **W-Timestamp**.

Optimistic Method for Concurrency Control

- In this method assume that **conflicts are very rare**. It allows transaction to run completely. And it checks for conflicts **before** they **commit**.
- It is also known as **validation** or **certification** method.

Execution of transaction **Ti** is done in **three** phases.

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

1) Read Phase:

- The transaction reads input values from the database, performs computation and records the updates in **a temporary local variable** that is **not accessed** by other transactions.

2) Validation Phase:

- Transaction **Ti** performs a ``**validation test**'' to determine if local variables can be written without violating serializability.
- If test is **positive**, then transaction **goes to the write** phase **otherwise** changes are **discarded** and transaction **Ti** is **restarted**.

3) Write Phase:

- In this phase **changes**, recorded in local variables are **permanently applied to the database**.

Advantages:

- It is very efficient when conflicts are rare.
- If rollback is required then database not involved. So, it is **less expensive**.

Disadvantage:

- Conflict requires entire transaction to rollback. So, it becomes **more expensive**.
- They may suffer from starvation.