DATABASE MANAGEMENT SYSTEM

TRANSACTIONS IN DBMS

Transaction

A **transaction** is a **set of logically related operations** that are performed together to achieve a single task. It **accesses** and **modifies** the contents of the database.

What is Transaction Processing?

Transaction Processing is the way a database **handles and manages** the **execution** of transactions.

- It ensures that **multiple operations** inside a transaction are **executed safely, reliably, and correctly**.
- Even if **something goes wrong** (like system crash or network failure), **the database remains correct**.

🔵 **Example**:
Transferring ₹500 from Account A to Account B involves:

- Deducting ₹500 from Account A
- Adding ₹500 to Account B This whole process must happen **together** — either **both operations succeed** or **none**.

Transaction Operations

A transaction typically involves four basic operations:

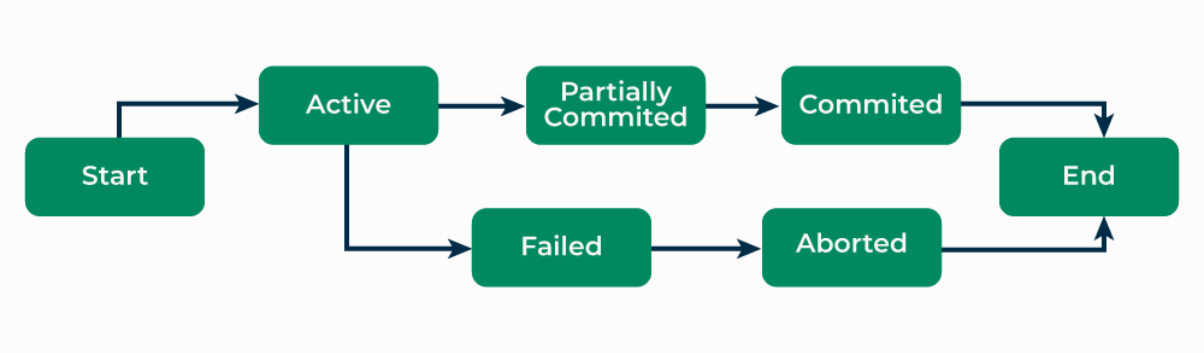
- **Read(X)** → Read a data item X.
- **Write(X)** → Write (update) a data item X.
- **Commit** → Save the changes permanently.
- **Rollback** → Undo the changes (in case of error).

Properties of Transaction – ACID Properties

Every transaction must satisfy **ACID** properties:

Property	Meaning
Atomicity	"All or nothing" — Either the entire transaction happens, or nothing happens.
Consistency	Transaction must take the database from one **valid state** to another.
Isolation	Transactions should not interfere with each other. They should appear to run independently.
Durability	Once a transaction is committed, changes must persist, even if a system failure occurs.

States of a Transaction:



State	Description
Active	Transaction starts and operations like read, insert, update, delete are performed. Data changes are in memory (buffer), not yet saved.
Partially Committed	Final operation is being executed. Ready to commit. If error → goes to **Failed**.
Committed	All operations including **commit** successful. Data saved permanently in the database.

State	Description
Failed	Error occurs before full commit. Database recovery system ensures consistency.
Aborted	Error not recoverable. Transaction rolled back to previous consistent state. May restart or terminate transaction.

Concurrent Executions

In a database management system (DBMS), **concurrent execution** allows multiple transactions to run at the same time. This is essential for improving performance and utilizing system resources efficiently.

Advantages of Concurrent Execution:

1. **Increased Processor and Disk Utilization**:
    - Multiple transactions can use the CPU and disk at the same time.
    - Example: While one transaction is using the CPU for computation, another transaction can access the disk for reading or writing data. This results in **better throughput** for the system.
2. **Reduced Average Response Time**:
    - Short transactions can execute and complete without waiting for long-running transactions to finish.
    - This **reduces the waiting time** for transactions, leading to faster response times and better system performance.

Transaction Schedules

- 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, when multiple transaction requests are made at the same time, we need to decide their order of execution.
- Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions.
- There are broadly two types of transaction schedules discussed as follows,

1. Serial Schedule

- In this kind of schedule, when multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed.
- This ensures consistency in the database as transactions do not execute simultaneously.
- But it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time.

For example: Suppose there are two transactions T1 and T2 which have some operations.

If no interleaving (mixing) of operations happens, then two possible serial executions are:

- Schedule A: Execute all operations of T1 first, then all operations of T2.
- Schedule B: Execute all operations of T2 first, then all operations of T1.

(a)

Time

T1	T2

read(A);
A := A – N;
write(A);
read(B);
B := B + N;
write(B);	

	read(A);
	A := A + M;
	write(A);

Schedule A

(b)

Time

T1	T2

	read(A);
	A := A + M;
	write(A);

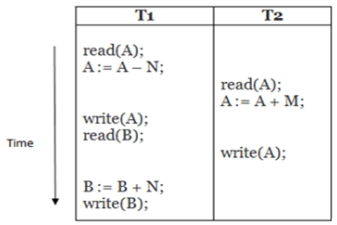
read(A);
A := A - N;
write(A);
read(B);
B := B + N;
write(B);	

Schedule B

2. Non-Serial/Parallel Schedule

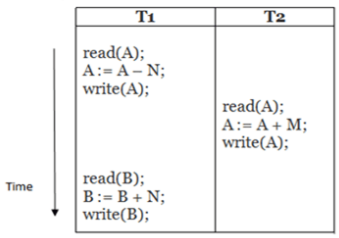
- To reduce the waiting time of transactions in the waiting queue and improve the system **efficiency**, we use non-serial schedules which allow **multiple transactions** to start before a transaction is completely executed.
- This may sometimes result in **inconsistency** and **errors** in database operation.
- So, these errors are handled with specific algorithms to maintain the consistency of the database and improve CPU throughput as well.

(c)



Schedule C

(d)



Schedule D

What is Serializability?

Serializability ensures that the final state of the database, after a set of transactions executes, is the same as if the transactions were executed one after the other (serial execution).

Basic Assumption:

- Each transaction preserves database consistency, meaning if transactions are executed serially, the database's consistency is maintained.
- A schedule (whether serial or concurrent) is **serializable** if its execution is equivalent to some serial schedule.

A schedule is **serializable** if **final result = result of some serial execution**.

Serial schedules (one transaction after another) are **always serializable** because transactions don’t overlap.

Types of Serializability:

1. **Conflict Serializability**:
    - **Definition**: A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
    - **Conflict**: Operations from different transactions conflict if they access the same data item, and at least one of them is a write operation.

Conflict Conditions:

- **Case 1**: li = read(Q), lj = read(Q) → No conflict (Both read the same item)
- **Case 2**: li = read(Q), lj = write(Q) → Conflict (One reads, the other writes)
- **Case 3**: li = write(Q), lj = read(Q) → Conflict (One writes, the other reads)
- **Case 4**: li = write(Q), lj = write(Q) → Conflict (Both write the same item)

Key Insight:

- Conflicting operations must occur in a **specific order**. If operations don't conflict, their order doesn't matter, and they can be swapped without changing the result of the transactions.

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 6

2. **View Serializability**:
    - **Definition**: A schedule is view serializable if it can be transformed into a serial schedule by swapping only non-conflicting instructions.
    - **Conflict Equivalence**: Two schedules are conflict equivalent if they can be transformed into one another by swapping non-conflicting operations.
    - **Conflict Serializable vs View Serializable**:
        - A schedule that is **conflict serializable** is also **view serializable**.
        - However, a schedule that is **view serializable** may not always be conflict serializable.

View Equivalence in Schedules:

Two schedules **S** and **S'** are **view equivalent** if they meet these conditions for each data item **Q**:

1. **Initial Value**: If in **S**, a transaction **Ti** reads the initial value of **Q**, then **Ti** must also read the initial value of **Q** in **S'**.
2. **Read from Write**: If in **S**, **Ti** reads **Q** after **Tj** writes it, **Ti** must read the same value written by **Tj** in **S'**.
3. **Final Write**: The transaction performing the **final write** on **Q** in **S** must also do so in **S'**.

Testing for Serializability Using Precedence Graph

In order to test if a given schedule of transactions is serializable, we use a **precedence graph** (also called a **serializability graph**). The steps to construct and analyze the graph are outlined below:

Steps to Construct and Analyze the Precedence Graph:

1. **Vertices**: Each transaction is a vertex (e.g., T1, T2, ..., Tn).
2. **Edges**: Draw an edge from **Ti** to **Tj** if:

- Ti and Tj conflict (access the same data item).
- Ti accessed the item first.

3. **Edge Label**: Label edges with the conflicting data item (e.g., A, B).

Conflict Rules:

- Conflict occurs if:
    1. One transaction reads, the other writes the same item.
    2. Two transactions write the same item.

Example: For schedule:

- **T1**: R(A), W(B), W(A)
- **T2**: W(A), R(B)

Conflicts:

1. **T1 → T2** on A (T1 reads A before T2 writes A).
2. **T1 → T2** on B (T1 writes B before T2 reads B).

Precedence Graph:

- T1 → T2 (Conflict on A)
- T1 → T2 (Conflict on B)

Checking for Cycles:

- If there is a cycle in the graph, the schedule is **not serializable**.
- If no cycle, the schedule is **serializable**.

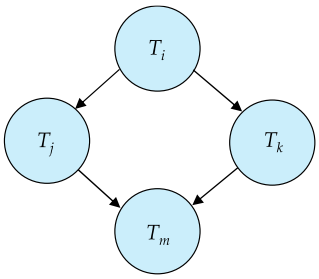
In this example, there are no cycles, so the schedule is **serializable**.

Test for Conflict Serializability

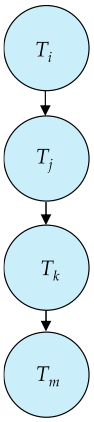
A schedule is **conflict serializable** if and only if its **precedence graph** is **acyclic**.

Steps to Test Conflict Serializability:

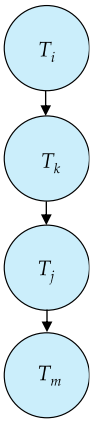
1. **Construct the Precedence Graph**: Build the graph where vertices represent transactions and edges represent conflicts.
2. **Check for Cycles**:
    - Use cycle-detection algorithms (e.g., $O(n^2)$ or $O(n + e)$ time complexity, where n = number of vertices, e = number of edges).
    - If there is **no cycle**, the schedule is **conflict serializable**.
    - If a **cycle is present**, the schedule is **not conflict serializable**.
3. **Obtain Serializability Order**: If the graph is acyclic, perform a **topological sort** of the precedence graph to find a **serializability order**. This gives a linear order of transactions that preserves consistency.



(a)



(b)



(c)

Example:

- Given a schedule with a valid acyclic precedence graph, a serializability order might be:
    - T5 → T1 → T3 → T2 → T4

Are there others?

- Yes, multiple serializability orders may exist if the precedence graph has multiple valid topological sorts.

Recoverable Schedules

A **recoverable schedule** ensures that if a transaction **Tj** reads a data item previously written by a transaction **Ti**, then **Ti's commit** must occur before **Tj's commit**. This prevents a scenario where **Tj** could read an inconsistent state if **Ti** aborts.

Key Points:

- A schedule is **recoverable** if, for each transaction **Tj** that reads a data item written by **Ti**, **Ti** must commit before **Tj** commits.
- **Non-recoverable schedule example**:
    - **T8** writes a data item, and **T9** reads it.
    - If **T8** aborts after **T9** reads, **T9** could have read an inconsistent state, leading to database issues.
    - This kind of schedule is **not recoverable** because **T9** would be exposed to an inconsistent state if **T8** aborts.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Conclusion: To maintain consistency, a database system must ensure that only **recoverable schedules** are executed, meaning transactions must commit in the correct order to avoid cascading failures.

Cascading Rollbacks

A **cascading rollback** occurs when a single transaction failure causes a series of rollbacks of other dependent transactions. This can result in significant work being undone.

Example:

- If **T10** fails and has written data that **T11** and **T12** have read (but haven't committed yet), then both **T11** and **T12** must be rolled back as well.
- Even though the schedule is **recoverable** (since no transactions have committed yet), the failure of **T10** causes **T11** and **T12** to also roll back, resulting in a cascading effect.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Impact:

- This leads to the undoing of potentially large amounts of work, reducing system efficiency and causing unnecessary delays.
- To avoid cascading rollbacks, **strict schedules** or other mechanisms can be used, ensuring that a transaction's failure does not trigger a series of rollbacks.

Cascadeless Schedules

A **cascadeless schedule** ensures that **cascading rollbacks** cannot occur. This is achieved by guaranteeing that, for any pair of transactions **Ti** and **Tj**, where **Tj** reads a data item written by **Ti**, the **commit operation of Ti** occurs before **Tj** performs the read operation.

Key Points:

- **Cascadeless Schedule**: Prevents cascading rollbacks by ensuring the commit of a transaction happens before another transaction reads data it has written.
- **Recoverability**: Every cascadeless schedule is also recoverable, ensuring that if a transaction fails, the system can still maintain consistency.
- **Desirability**: It is generally desirable to restrict schedules to cascadeless ones because they prevent the need for multiple rollbacks, enhancing efficiency and consistency in a system.

Concurrency Control in Databases

- **Goal**: A database system must ensure that all schedules are:
    - **Conflict or View Serializable**.
    - **Recoverable**.
    - **Preferably Cascadeless** for consistency and reliability.
- **Serial Schedules**:
    - **Serial Execution** (only one transaction at a time) generates schedules that are **serializable** but **low concurrency**.

    - **Serial schedules are inherently recoverable and cascadeless**, as no interleaving occurs.

- **Testing Serializability After Execution**: It's inefficient to check if a schedule is serializable after it has been executed. The focus is on developing concurrency control mechanisms that ensure serializability **during execution**.
- **Concurrency Control Protocols**:
    - **Trade-offs**: Concurrency control protocols need to balance:
        - **Concurrency**: How many transactions can run concurrently.
        - **Overhead**: The additional complexity in maintaining consistency.
    - Some protocols generate only **conflict-serializable** schedules, while others may generate **view-serializable** schedules, which might not be conflict-serializable but still maintain consistency.
- **Key Considerations**:
    - Ensuring schedules are **serializable**, **recoverable**, and ideally **cascadeless**.
    - The system needs to manage concurrency while avoiding issues like cascading rollbacks or inconsistent states.

Concurrency Control vs. Serializability Tests:

Aspect	Concurrency Control	Serializability Tests
Purpose	Ensure concurrent schedules are **serializable**, **recoverable**, and **cascadeless**.	Validate if a schedule is **serializable** (conflict or view serializable).
How They Work	Imposes protocols/rules to prevent non-serializable schedules during execution.	Examines **precedence graph** to check serializability.
Trade-offs	Balances **concurrency** (more transactions running at once) with **overhead** (performance cost).	Helps validate correctness but does not directly impact performance.
Focus	Prevents **conflict** and **cascading rollbacks** in concurrent executions.	Tests if a schedule can be reordered to a **serial schedule**.
Goal	Allow concurrent execution while ensuring consistency and isolation.	Determine if a schedule can be **serializable** and identify violations.
Test Type	Does not directly analyze precedence graphs but relies on protocol rules.	Directly examines **precedence graphs** or equivalent to check serializability.
Protocol	Ensures serializability by design (e.g., 2PL, timestamp ordering).	Used to validate if a concurrency control protocol works correctly.

Transaction Definition in SQL:

- **Implicit Transaction Start**: In SQL, a transaction begins automatically when the first SQL statement is executed.
- **Ending a Transaction**:
    - **Commit**: COMMIT WORK ends the current transaction and starts a new one.

    - **Rollback**: ROLLBACK WORK aborts the current transaction.
- **Implicit Commit**:
    - By default, most databases automatically commit after each successful SQL statement.
    - Implicit commit behavior can be turned off using a database directive.
        - Example (JDBC): connection.setAutoCommit(false);
- **Isolation Levels**:
    - Isolation levels control the visibility of uncommitted changes to other transactions.
    - Can be set at the database level.
    - Can be changed at the start of a transaction.
        - Example (SQL): SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
        - Example (JDBC): connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

These settings help control transaction behavior, such as isolation, and ensure database consistency during concurrent operations.