DATABASE MANAGEMENT SYSTEM

CONCURRENCY CONTROL IN DBMS

Concurrency Control in DBMS

Overview

Concurrency control is a crucial concept in database management systems (DBMS) that ensures the correct execution of multiple transactions running concurrently, while maintaining the consistency, isolation, and durability of the database. It ensures that database transactions do not interfere with each other in a way that could lead to inconsistencies, corruption, or violation of integrity constraints.

Locking Mechanism in Database Systems

1. Overview of Locks:

- Definition: A lock is a mechanism to control concurrent access to a data item, ensuring data consistency and isolation between transactions.
- Purpose: Locks prevent conflicts by allowing only one transaction to modify a data item at a time or by restricting the types of access during concurrent execution.

2. Types of Locks:

1. **Exclusive (X) Lock:**
    - Usage: Allows both read and write on a data item.
    - Command: lock-X instruction.
    - Behavior:
        - Only one transaction can hold the exclusive lock on a data item.
        - Blocks other transactions from reading or writing to the item.
2. **Shared (S) Lock:**
    - Usage: Allows read-only access to a data item.
    - Command: lock-S instruction.

- Behavior:
  - Multiple transactions can hold a shared lock simultaneously.
  - Blocks other transactions from writing to the item but allows concurrent reads.

3. **Lock Request Process**:

- Concurrency-Control Manager: The manager grants or denies lock requests. A transaction can proceed only once the lock is granted.
- Granting Locks: A transaction must wait if another holds a conflicting lock.

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.

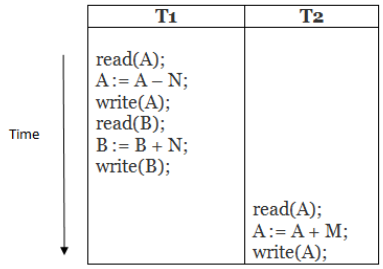
- But it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time.

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:

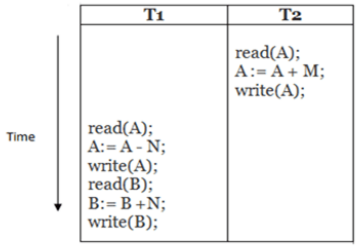
- Schedule A: Execute all operations of T1 first, then all operations of T2.
- Schedule B: Execute all operations of T2 first, then all operations of T1.

(a)



Schedule A

(b)

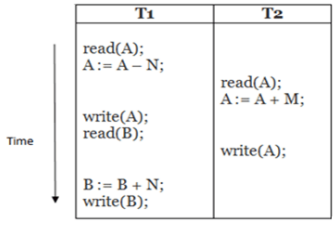


Schedule B

2. Non-Serial/Parallel Schedule

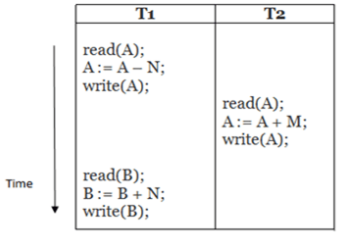
- To reduce the waiting time of transactions in the waiting queue and improve the system **efficiency**, we use non-serial schedules which allow **multiple transactions** to start before a transaction is completely executed.
- This may sometimes result in **inconsistency** and **errors** in database 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 a Read-Write Conflict?

A **Read-Write conflict** happens when:

- One transaction is **reading** a data item
- Another transaction is **writing/updating** that same data item **at the same time**

🛑 **Unrepeatable Read**

- **Definition:**
  When a transaction **reads the same data twice** but **gets different values** because another transaction **updated** the data between the two reads.
- **When it Happens:**
  ✅ Happens during **parallel/concurrent execution** of transactions.
  ❌ Does **not** happen in **serial execution** (because one transaction completes fully before the next starts).

What is Serializability?

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- **Serializability** checks if a **non-serial schedule** maintains database **consistency**.
- 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.

✅ **Goal:** Achieve consistency even when transactions run in parallel.

Deadlock and Starvation in DBMS

Deadlock:

- **Definition:**
  Deadlock occurs when **two or more transactions** are **waiting for each other to release locks**, and **none can proceed**.
- A **deadlock** situation occurs when a set of transactions are waiting on each other in a circular chain. This means each transaction is waiting for a resource that is held by another transaction, creating a cycle of dependencies where no transaction can proceed.

Example:

- Transaction **T3** holds an **exclusive lock** on **B**, and **T4** wants a **shared lock** on **B** → T4 **waits** for T3.
- Transaction **T4** holds a **shared lock** on **A**, and **T3** wants an **exclusive lock** on **A** → T3 **waits** for T4.

Thus, **both are waiting for each other**, causing a **deadlock — no transaction can proceed**!

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

4 deadlock conditions in short:

1. **Circular Wait**: A set of transactions are waiting for each other in a cycle, preventing any from proceeding.
2. **Hold and Wait**: A transaction holds one resource and waits for another, leading to potential blocking.
3. **No Preemption**: Once a transaction holds a resource, it cannot be forcibly taken away, causing a deadlock if transactions are waiting for each other.
4. **Mutual Exclusion**: At least one resource is held in an exclusive mode, preventing other transactions from accessing it, which can cause waiting and deadlock.

Starvation

- **Definition:**
  Starvation happens when a transaction **waits indefinitely** to get a lock **because other transactions keep acquiring it first**.

Example:

- **T2** holds a **shared lock** on a data item.
- **T1** requests an **exclusive lock** → **T1 waits**.
- Meanwhile, **T3**, **T4**, **T5**… keep requesting **shared locks**, and each is **granted** before T1 gets a chance.
- So, **T1 keeps waiting forever** and **never gets its exclusive lock**.

✅ **T1 is starved** — it **cannot make progress**.

Locking Protocol:

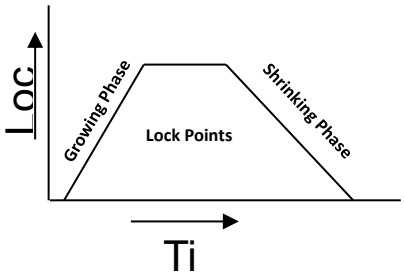
- A **locking protocol** is a set of **rules** or guidelines that all transactions follow while **requesting** and **releasing locks** on data items in a database.
- The primary goal of these protocols is to ensure **serializability** and **data consistency** during the concurrent execution of transactions.

🔒 Two-Phase Locking Protocol (2PL)

Purpose:

- Ensures **serializability** (correct, consistent transaction execution).

How it works:
A transaction **divides its lock requests into two phases**:



Phase	Meaning	Think of it like
Growing Phase	Transaction can acquire new locks but cannot release any locks.	"I can only collect more keys but I cannot return any key yet."
Shrinking Phase	Transaction can release locks but cannot acquire any new locks.	"Once I return a key, I can only return more keys — I cannot ask for new keys anymore."

🧩 **Key Points:**

- Transaction starts in **growing phase** — keeps acquiring locks.
- Once it **releases** a lock → **enters shrinking phase** → can **only release**, no more locking.

Two-phase locking *does not* ensure freedom from deadlocks

Extensions to Basic 2PL:

1. **Strict Two-Phase Locking:**
    - A transaction **must hold all its exclusive (write) locks until it commits or aborts**.
    - **Benefits:**
        - Ensures **recoverability**.
        - **Avoids cascading rollbacks** (since no other transaction can read uncommitted changes).
2. **Rigorous Two-Phase Locking:**
    - A transaction **must hold both shared (read) and exclusive (write) locks until it commits or aborts**.
    - **Benefits:**
        - Stronger than strict 2PL.
        - Transactions can be **serialized** based on their **commit order**.

- **Simplifies recovery and concurrency control**.

Practical Note:

- **Most database systems** use **rigorous 2PL** but often **refer to it simply as two-phase locking** in documentation.

Automatic Acquisition of Locks:

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

```
if Ti has a lock on D
 then
      read(D)
 else begin
       if necessary wait until no other
          transaction has a lock-X on D
       grant Ti a  lock-S on D;
       read(D)
      end
```

- The operation **write***(D)* is processed as:

```
if Ti has a  lock-X on D
 then
  write(D)
 else begin
    if necessary wait until no other trans. has any lock on D,
    if Ti has a lock-S on D
       then
```

upgrade lock on *D* to **lock-X**

else

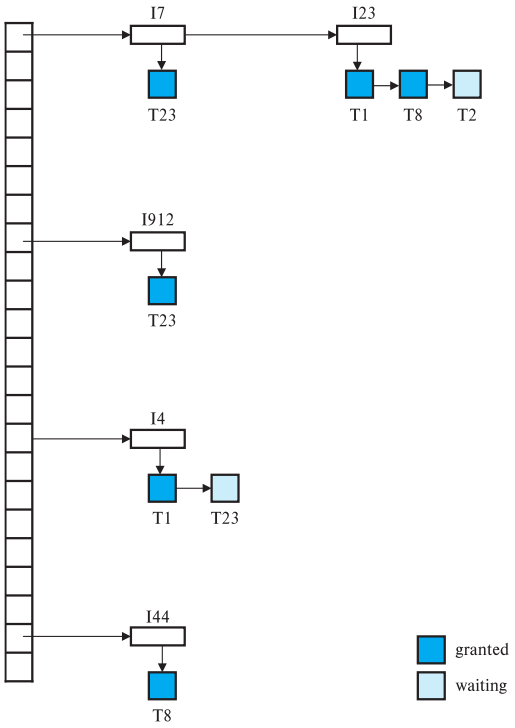
grant T_i a **lock-X** on *D*

write(*D*)

end;

- **All locks are released after commit or abort**

Implementation of Locking



- A **Lock Manager** controls locking in the database and can be **implemented as a separate process**.
- **Transactions** send **lock** and **unlock requests** to the lock manager as **messages**.
- The **Lock Manager** responds:
  - **Grants the lock** (lock grant message), or
  - **Asks for rollback** if a deadlock is detected.
- The **transaction waits** until the lock manager replies to its request.
- The **Lock Manager maintains a Lock Table**, an **in-memory data structure**, to:
  - **Record** currently **granted locks**,
  - **Track pending lock requests**.

Lock Table

- A **Lock Table** is an **in-memory structure** managed by the **Lock Manager**.

- **Dark rectangles** = **Granted locks**
  Light rectangles = **Waiting (pending) lock requests**.
- Each entry in the lock table records:
  - **Type of lock** (Shared or Exclusive),
  - **Transaction requesting/holding** the lock.

How Locking Works:

- **New lock requests** are added **at the end** of the queue for that data item.
- A lock is **granted immediately** if it is **compatible** with all previously granted locks.
- **Unlock operations**:
  - Remove the transaction's lock from the lock table,
  - Check if pending requests can now be granted.
- **If a transaction aborts**:
  - **All its locks** (granted or waiting) are **deleted**.
  - Lock manager may maintain a **list of locks held by each transaction** to handle this quickly.

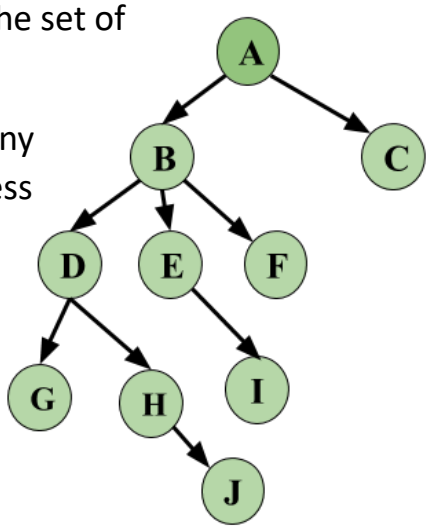
2. Graph-Based Protocols

Graph-Based Protocols are designed to ensure **serializability** and avoid conflicts between transactions. The general idea is that we have a **partial ordering** of data items and use that order to manage how transactions acquire locks.

Key Concepts of Graph-Based Protocols:

1. **Partial Ordering of Data Items**:

- A **partial ordering** (denoted as →) is imposed on the set of data items $D = \{d1, d2, \ldots, dh\}$.
- For any two data items **di** and **dj**, if **di → dj**, then any transaction that accesses both **di** and **dj** must access **di** before accessing **dj**.
- This ordering ensures that transactions follow a sequence that maintains database consistency. The partial ordering can be based on the **logical or physical organization** of the data, or it may be specifically designed for **concurrency control**.



2. **Tree Protocol**:
   - The **tree protocol** is a lock-based concurrency control method that enforces a specific structure for locking and unlocking data items.
   - In the **tree protocol**, **only lock-X (exclusive lock)** is allowed, meaning a transaction must acquire exclusive locks on data items to modify them.

Deadlock Prevention Techniques

Deadlock prevention strategies aim to ensure that the system does not enter a deadlock state by eliminating one or more of the necessary conditions for a deadlock: mutual exclusion, hold and wait, no pre-emption, and circular wait. There are two primary approaches to deadlock prevention:

1. Lock Ordering (Preventing Cyclic Waits)

In this approach, transactions request locks on data items in a predefined, consistent order. By ensuring that transactions lock data items in the same sequence, cyclic waits (and hence deadlocks) are avoided.

Details:

- **Lock All Items Before Execution**: A transaction locks all the data items it requires before starting. Either all are locked in one step, or none are locked. This method can cause two issues:

- **Data Item Ordering**: Another method involves imposing a strict order on data items and requiring transactions to lock them in that order (e.g., D1→D2→D3).

2. Preemption and Transaction Rollbacks

This approach uses **timestamps** assigned to each transaction to manage whether a transaction should wait or be rolled back when it requests a lock held by another transaction. If a deadlock is detected, transactions can be preempted or rolled back to prevent deadlock situations.

Key Components:

- **Timestamps**: Each transaction gets a unique timestamp when it starts. This timestamp is used to determine whether a transaction should be allowed to wait or be rolled back.
- **Preemption**: If a transaction requests a lock held by another, the system can preempt the transaction holding the lock, rolling it back and granting the lock to the requesting transaction.

Two Schemes for Handling Deadlocks Using Timestamps:

1. Wait-Die Scheme (Non-preemptive):

Wait-for Graph

- **Vertices**: Represent **transactions**.
- **Edge Ti → Tj**: Means **Ti is waiting** for a lock **held by Tj** (in a conflicting mode).
- **Deadlock** occurs **if and only if** there is a **cycle** in the wait-for graph.

2. Wound-Wait Scheme (Preemptive):

Condition: In this scheme, if Ti requests a lock held by Tj, Ti can only wait if its timestamp is larger (i.e., Ti is younger than Tj). Otherwise, Tj is preempted and rolled back (i.e., **wounded** by Ti).

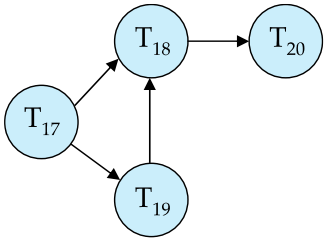
Deadlock Detection and Recovery

- **Periodically run** a **deadlock-detection algorithm** to check for cycles.
- **When a deadlock is found**:
  - Choose a **victim transaction** to **roll back** and **break the cycle**.
  - Prefer **victims with minimum rollback cost** (least impact).

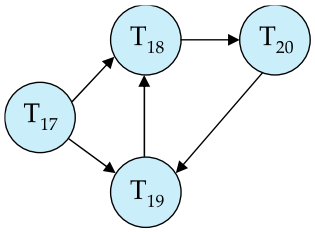
Rollback Types:

- **Total Rollback**:
  - **Abort the entire transaction** and **restart** it.
- **Partial Rollback**:
  - **Undo only enough** operations to **release the required locks**.

Condition: If transaction Ti requests a lock on a data item held by transaction Tj , Ti is allowed to wait if its timestamp is smaller (i.e., Ti is older than Tj). If Ti's timestamp is larger, it is rolled back (i.e., **dies**).



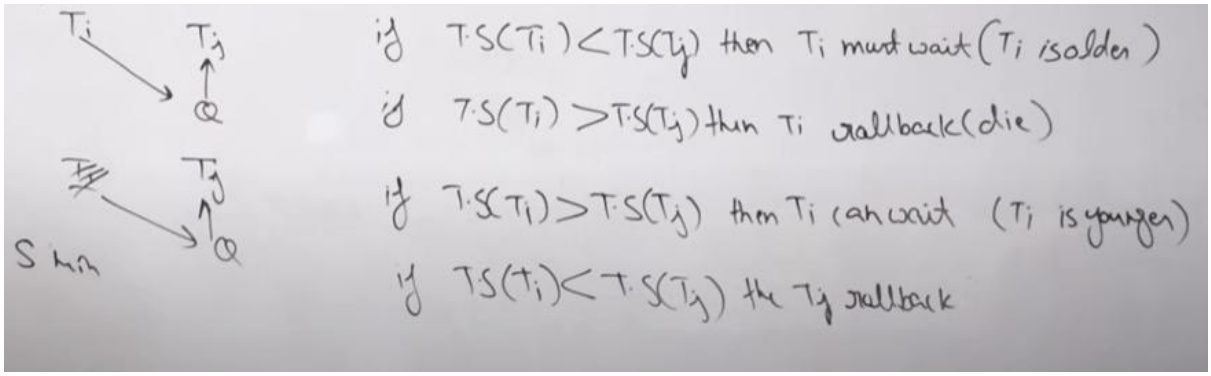
Wait-for graph without a cycle



Wait-for graph with a cycle

- **Wound-Wait Scheme** (Preemptive):

Condition: In this scheme, if Ti requests a lock held by Tj, Ti can only wait if its timestamp is larger (i.e., Ti is younger than Tj). Otherwise, Tj is preempted and rolled back (i.e., **wounded** by Ti).

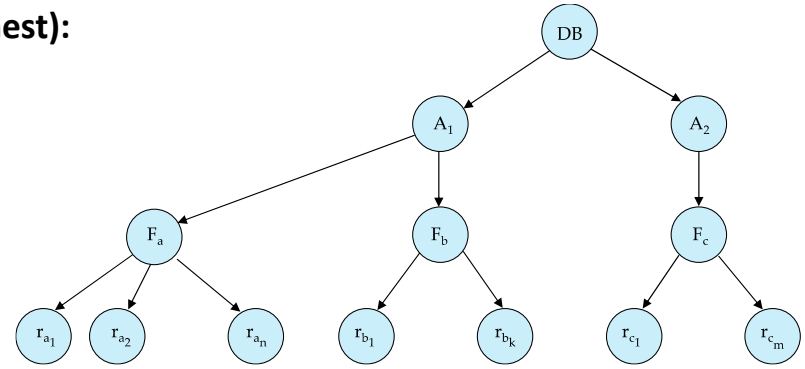


Multiple Granularity

- Multiple Granularity means locking data at different levels of size — like whole database, area, file, or a single record.
- Data is organized in a tree structure, where big data items are at the top and smaller data items are at the bottom.
- Locking a higher node (like a file) automatically locks all its lower nodes (like records inside that file).

Granularity Levels (Coarsest → Finest):

Database → Area → File →Record



Granularity Trade-off:

Granularity	Concurrency	Locking Overhead
Fine (lower in tree)	High	High
Coarse (higher in tree)	Low	Low

Intention Lock Modes (for Multiple Granularity)

Besides **S** (Shared) and **X** (Exclusive) locks, there are **three intention locks**:

Lock Mode	Meaning
IS (Intention-Shared)	Will lock lower-level nodes **in Shared (S) mode**.
IX (Intention-Exclusive)	Will lock lower-level nodes **in S or X mode**.
SIX (Shared + Intention-Exclusive)	Node is **Shared-locked**; lower levels may have **Exclusive locks**.

Phantom Phenomenon

- Happens when **new tuples** (rows) are **inserted/updated during a transaction's predicate scan**.
- Even if **no common tuple** is accessed, **conflicts** can still happen.

Example:

1. Transaction T1 reads:

   select count(*) from instructor where dept_name = 'Physics';

2. Transaction T2 inserts:

   insert into instructor values ('11111', 'Feynman', 'Physics', 94000);

3. Problem:

- T1 misses the new record by T2, even though they don’t lock the same old tuples.
- Non-serializable schedule is possible.

Timestamp-Based Protocols

Each transaction Ti is assigned a **unique timestamp TS(Ti)** before execution starts.

Timestamp: A timestamp is a unique identifier assigned to each transaction when it enters the system. Determines the transaction's priority and order of execution-older transactions (with smaller timestamps) are given higher priority over newer ones.

- Timestamp order defines the **serializability**:
  If **TS(Ti) < TS(Tj)**, the system must schedule **Ti before Tj**.

How Timestamps Are Assigned:

- **System Clock**: TS(Ti) = current system time at transaction start.
- **Logical Counter**: TS(Ti) = incremented counter value.

Purpose:

- Ensures **conflict-serializable schedules** based on timestamps.
- Older transactions (smaller timestamp) get higher priority.

Notations Used:

- **TS(Ti)**: Timestamp assigned to transaction Ti.
- **W_TS(Q)**: Largest TS of any transaction that **wrote** Q successfully.
- **R_TS(Q)**: Largest TS of any transaction that **read** Q successfully.

Thus, **timestamps control the execution order** and **avoid conflicts** automatically without deadlocks.

Timestamp-Ordering Protocol

- **Purpose**: Ensures that conflicting operations (read/write) occur **according to timestamp order** to guarantee serializability.

Rules:

1. **When Ti issues read(Q):**
    - If **TS(Ti) < W_TS(Q)** →
        - Ti is trying to read an outdated value →
        - **Reject read and rollback Ti**.
    - Else →
        - **Allow read**, and update **R_TS(Q) = max(R_TS(Q), TS(Ti))**.
2. **When Ti issues write(Q):**
    - If **TS(Ti) < R_TS(Q)** →
        - New value is **too late** (needed earlier) →
        - **Reject write and rollback Ti**.
    - Else if **TS(Ti) < W_TS(Q)** →
        - Ti is writing an **obsolete value** →
        - **Reject write and rollback Ti**.
    - Else →
        - **Allow write**, and set **W_TS(Q) = TS(Ti)**.

✅ **Key Points**:

- Prevents inconsistent schedules.
- May cause more rollbacks compared to Thomas' Write Rule (which is an improvement).
- **Timestamps** guide both reads and writes.

Thomas' Write Rule

- **Problem**: In basic Timestamp Ordering (TO), even unnecessary rollbacks happen.
  Example:
  - **T27** starts before **T28** (so, TS(T27) < TS(T28)).
  - **T28** writes **Q** before T27 writes **Q**.
  - T27's write(Q) is now **obsolete** but **rollback happens unnecessarily** under basic TO.
- **Idea**:
  If a transaction tries to write an **outdated value** (already overwritten by a newer transaction), **just ignore** its write — **no rollback needed**.

Thomas' Write Rule:

When Ti issues **write(Q)**:

1. **If TS(Ti) < R_TS(Q)** →
   - Ti is **too old**, needed value already read →
   - **Rollback Ti**.
2. **If TS(Ti) < W_TS(Q)** →
   - Ti's write is **obsolete** →
   - **Ignore** the write (no rollback).
3. **Otherwise** →
   - Allow the **write**, and update **W_TS(Q) = TS(Ti)**.

Summary:

- **Less rollback**, **better concurrency**, **more efficient** than pure timestamp-ordering.
- **Key**: Obsolete writes are ignored instead of causing rollback.

Validation-Based Protocol (Optimistic Concurrency Control)

Idea:
The commit time is used as the serialization order. Transactions execute fully in the hope that they won't conflict, and their final correctness is checked at commit time.

- The protocol operates on the assumption that most transactions won't conflict, so they are allowed to execute freely and only validated later.

Phases of Transaction Execution:

1. Read and Execution Phase:
    - The transaction reads data and performs computations.
    - Writes are made only to temporary local variables, ensuring that no changes are made to the database yet.
2. Validation Phase:
    - Once the transaction has completed its computation, it performs a validation test to check if the operations can be committed without violating serializability.
    - The validation checks whether the data read or written by the transaction does not conflict with other transactions that have already committed.
3. Write Phase:
    - If the transaction passes the validation test, the updates are applied to the database.
    - If the validation fails, the transaction is rolled back, meaning none of the changes are applied.

Execution Flow:

- Interleaved Transactions: Multiple transactions can execute concurrently in these phases, but each transaction must go through the three phases in the same order: Read → Validation → Write.
- Atomic Validation/Write:
    - For simplicity, we assume that the validation and write phases are executed atomically for each transaction (i.e., only one transaction can perform validation and writing at a time).
