Unit -5

Lock-Based Protocol

A lock in DBMS controls concurrent access, allowing only one transaction to use a data item at a time. This ensures data integrity and prevents issues like lost updates or dirty reads during simultaneous transactions.

Lock Based Protocols in DBMS ensure that a transaction cannot read or write data until it gets the necessary lock. Here's how they work:

- These protocols prevent concurrency issues by allowing only one transaction to access a specific data item at a time.
- Locks help multiple transactions work together smoothly by managing access to the database items.
- Locking is a common method used to maintain the serializability of transactions.
- A transaction must acquire a read lock or write lock on a data item before performing any read or write operations on it.

Types of Lock

1. **Shared Lock (S):** Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
2. **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Rules of Locking

The basic rules for Locking are given below:

Read Lock (or) Shared Lock(S)

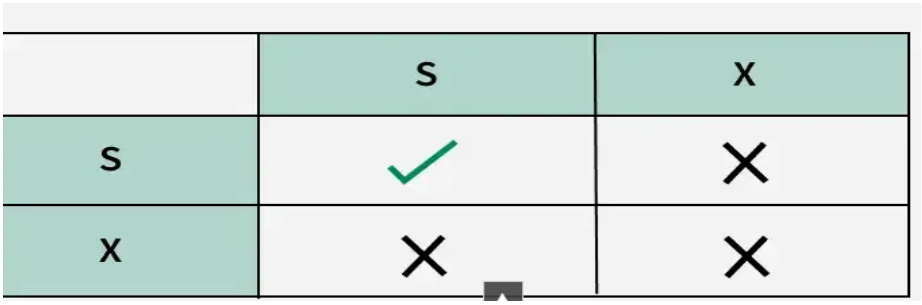
- If a Transaction has a Read lock on a data item, it can read the item but not update it.
- If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.
- So, the Read Lock is also called a Shared Lock.

Write Lock (or) Exclusive Lock (X)

- If a transaction has a write Lock on a data item, it can both read and update the data item.
- If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.
- So, the Write Lock is also known as Exclusive Lock.

Lock Compatibility Matrix

- A transaction can acquire a lock on a data item only if the requested lock is compatible with existing locks held by other transactions.
- **Shared Locks (S):** Multiple transactions can hold shared locks on the same data item simultaneously.
- **Exclusive Lock (X):** If a transaction holds an exclusive lock on a data item, no other transaction can hold any type of lock on that item.
- If a requested lock is not compatible, the requesting transaction must wait until all incompatible locks are released by other transactions.
- Once the incompatible locks are released, the requested lock is granted.



Compatibility Matrix

Concurrency Control Protocols

Concurrency Control Protocols are the methods used to manage multiple transactions happening at the same time. They ensure that transactions are executed safely without interfering with each other, maintaining the accuracy and consistency of the database.

These protocols prevent issues like data conflicts, lost updates or inconsistent data by controlling how transactions access and modify data.

Types of Lock-Based Protocols

1. Simplistic Lock Protocol

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable all transactions to obtain a lock on the data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

Example: Consider a database with a single data item X = 10.

Transactions:

- **T1**: Wants to read and update X.
- **T2**: Wants to read X.

Steps:

1. T1 requests an exclusive lock on X to update its value. The lock is granted.
2. T1 reads X = 10 and updates it to X = 20.
3. T2 requests a shared lock on X to read its value. Since T1 is holding an exclusive lock, T2 must wait.
4. T1 completes its operation and releases the lock.
5. T2 now gets the shared lock and reads the updated value X = 20.

This example shows how simplistic lock protocols handle concurrency but do not prevent problems like deadlocks or limits concurrency.

2. Pre-Claiming Lock Protocol

The Pre-Claiming Lock Protocol avoids deadlocks by requiring a transaction to request all needed locks before it starts. It runs only if all locks are granted; otherwise, it waits or rolls back.

Example: Consider two transactions T1 and T2 and two data items, X and Y:

Transaction T1 declares that it needs:

- A write lock on X.
- A read lock on Y.

Since both locks are available, the system grants them. T1 starts execution: It updates X. It reads the value of Y.

While T1 is executing, Transaction T2 declares that it needs: However, since T1 already holds a write lock on X, T2's request is denied. T2 must wait until T1 completes its operations and releases the locks. A read lock on X

Once T1 finishes, it releases the locks on X and Y. The system now grants the read lock on X to T2, allowing it to proceed.
This method is simple but may lead to inefficiency in systems with a high number of transactions.

3. Two-phase locking (2PL)

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases :

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

For more detail refer the article Two-phase locking (2PL).

4. Strict Two-Phase Locking Protocol

Strict Two-Phase Locking requires that in addition to the 2-PL all Exclusive(X) locks held by the transaction be released until after the Transaction Commits.

Problem With Simple Locking

Consider the Partial Schedule:

S.No	T1	T2
1	lock-X(B)	
2	read(B)	
3	B:=B-50	

S.No	T1	T2
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	
9	.....	.....

1. Deadlock

In the given execution scenario, T1 holds an exclusive lock on B, while T2 holds a shared lock on A. At Statement 7, T2 requests a lock on B, and at Statement 8, T1 requests a lock on A. This situation creates a deadlock, as both transactions are waiting for resources held by the other, preventing either from proceeding with their execution.

2. Starvation

Starvation is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

Timestamp based protocols

The **timestamp-based protocol** is a concurrency control method in a DBMS that assigns a unique timestamp to each transaction to order their execution and ensure serializability. Older transactions (smaller timestamps) have higher priority.

The protocol maintains two timestamps for every data item `X` in the database:

- **R_TS(X) (Read Timestamp):** The timestamp of the transaction that last successfully *read* the data item `X`.
- **W_TS(X) (Write Timestamp):** The timestamp of the transaction that last successfully *wrote* (updated) the data item `X`.

When a transaction `T` (with timestamp `TS(T)`) requests a read or write operation on data item `X`, the system checks the following rules:

Read Operation Rules

If transaction `T` wants to `read(X)`:

- **If `TS(T)` < `W_TS(X)`:** The operation is rejected, and `T` is aborted/rolled back. This is because `T` is trying to read a value that was written by a "younger" (newer) transaction, which violates the timestamp order.
- **Otherwise (`TS(T)` ≥ `W_TS(X)`):** The operation is executed, and `R_TS(X)` is updated to be the maximum of the current `R_TS(X)` and `TS(T)`

Write Operation Rules

If transaction `T` wants to `write(X)`:

- **If `TS(T)` < `R_TS(X)` or `TS(T)` < `W_TS(X)`:** The operation is rejected, and `T` is aborted/rolled back. This prevents an older transaction from overwriting a value that has already been read or written by a younger transaction.
- **Otherwise (`TS(T)` ≥ `R_TS(X)` and `TS(T)` ≥ `W_TS(X)`):** The operation is executed, and `W_TS(X)` is set to `TS(T)`.

Example

- Consider two transactions, `T1` and `T2`, where `T1` has a timestamp `TS(T1) = 10` and `T2` has a timestamp `TS(T2) = 20` (`T1` is older). Both want to access data item `A`, where initially `R_TS(A) = 0` and `W_TS(A) = 0`.

Step	Operation	Description	Outcome
1	`T1: read(A)`	`TS(T1)` (10) ≥ `W_TS(A)` (0)	**Allowed**. `R_TS(A)` becomes 10.
2	`T2: write(A)`	`TS(T2)` (20) ≥ `R_TS(A)` (10) and `TS(T2)` (20) ≥ `W_TS(A)` (0)	**Allowed**. `W_TS(A)` becomes 20.
3	`T1: write(A)`	`TS(T1)` (10) < `W_TS(A)` (20)	**Rejected**. `T1` is rolled back because it tries to wri an outdated value (a newer transaction, T2, has already written).

- This protocol ensures that all conflicting operations follow the order of the transaction timestamps, guaranteeing conflict serializability and preventing deadlocks. However, it can lead to cascading rollbacks and potential starvation for long-running transactions.

Validation-Based Protocols

The **validation-based protocol**, also known as **Optimistic Concurrency Control (OCC)**, is an approach in database management systems that aims to maximize concurrency by assuming that conflicts between transactions are rare. Instead of implementing rigid locking mechanisms during execution, transactions are allowed to run freely and are only checked for conflicts just before they attempt to commit their changes to the database.

If a conflict is detected during the validation phase, the conflicting transaction is aborted and restarted, ensuring serializability.

The Three Phases of a Validation Protocol

Every transaction following this protocol proceeds through three distinct phases:

1. **Read Phase:**

- The transaction reads data items from the actual database.
- All modification operations (calculations, updates) are performed strictly on local copies or temporary workspaces, keeping the main database consistent during execution.
- The system records the start and end times for this phase.

2. **Validation Phase:**

- This is the critical phase where concurrency control is applied. The system checks if the transaction's commitment will violate serializability.
- A common method is comparing the transaction's execution period with those of other committed transactions to ensure that the data it read wasn't modified by another concurrent transaction that committed while the current transaction was active.
- If the validation succeeds, the transaction is marked for commitment. If it fails, the transaction is immediately rolled back and restarted.

3. **Write Phase:**

- If and only if the validation phase is successful, the locally updated data values are permanently written to the main database.
- This phase is typically brief to minimize blocking other transactions.

Example: Concurrent Hotel Bookings

Consider a scenario where two travel agents, `T1` and `T2`, attempt to book the last room in a hotel simultaneously. The available room count `RoomsAvailable` is currently 1.

Time	Transaction `T1` (TS 100)	Transaction `T2` (TS 101)	Description
T0	`read(RoomsAvailable)`		`T1` reads the value (1) into its local workspace.
T1	`decrement local RoomsAvailable`		`T1` locally calculates `1 - 1 = 0`.

T2		read(RoomsAvailable)	T2 reads the value (1) into its local workspace.
T3		decrement local RoomsAvailable	T2 locally calculates 1 - 1 = 0.
T4	<Validation Phase>		T1 validates. No other transaction has committed a conflicting write yet. **Validation passes.**
T5	<Write Phase>		T1 commits its change. The database value of RoomsAvailable becomes **0**.
T6		<Validation Phase>	T2 validates. The system detects that the data item RoomsAvailable has been modified by T1 *after* T2 started reading it.
T7			T2 fails validation and is **aborted/rolled back**. T2 must restart its booking process.

This protocol prevents both agents from successfully booking the same last room. T1 succeeds and commits, while T2 is stopped before it can corrupt the database state, thus maintaining consistency through optimistic, rather than pessimistic (locking), control.

Deadlock Handling

Deadlock handling in a DBMS focuses on managing situations where two or more transactions block each other indefinitely while waiting for resources. The key concepts revolve around preventing, avoiding, or detecting and recovering from these deadlocks.

Necessary Conditions for Deadlock

A deadlock can only occur if all four of the following conditions are met simultaneously:

- **Mutual Exclusion:** At least one resource must be non-shareable, meaning only one transaction can use it at a time (e.g., an exclusive lock on a data item).
- **Hold and Wait:** A transaction holds at least one resource while waiting for another resource currently held by another transaction.
- **No Preemption:** Resources cannot be forcibly taken away from a transaction; they must be released voluntarily by the holding transaction upon completion.
- **Circular Wait:** A circular chain of transactions exists, where each transaction in the chain is waiting for a resource held by the next transaction in the chain.

Handling Strategies and Key Concepts

1. Deadlock Prevention

Prevention strategies ensure that the system never enters a deadlocked state by breaking one of the four necessary conditions before a request is granted.

Key Concept: Resource Ordering: Transactions must request locks in a predefined, global order. This breaks the circular wait condition.

Example: If the rule is to always lock `Table A` before `Table B`, two transactions T1 and T2 cannot simultaneously hold A and wait for B, and hold B and wait for A. T2 would simply have to wait for T1 to finish with A first.

- **Key Concept: Wait-Die Scheme:** Uses timestamps to decide whether a transaction should wait or abort when requesting a locked resource.
    - **Example:** An older transaction (smaller timestamp) is allowed to wait for a resource held by a younger one, but a younger transaction requesting a resource held by an older one is immediately aborted ("dies") and restarted later.

2. Deadlock Avoidance

Avoidance involves dynamically checking resource allocation requests to ensure that granting them will not lead to an "unsafe state" (a state from which deadlock might occur).

Key Concept: Safe State/Banker's Algorithm: The system simulates the allocation of resources before granting a request to determine if a safe sequence of transaction execution exists that allows all transactions to complete.

Example: A DBMS uses the Banker's algorithm to analyze if granting a transaction the last available resource would leave the system in a state where other processes could still eventually finish. If not, the request is denied or delayed.

3. Deadlock Detection and Recovery

This approach allows deadlocks to occur, but the system periodically runs an algorithm to detect them.

Key Concept: Wait-For Graph (WFG): A directed graph is maintained where nodes represent transactions and an edge from T1 to T2 means T1 is waiting for a resource held by T2. A cycle in this graph indicates a deadlock.

Example: If T1 waits for T2, and T2 waits for T1, a cycle is formed in the WFG, signaling a deadlock.

Key Concept: Victim Selection and Rollback: Once a deadlock is detected (via a cycle), the system chooses one or more "victim" transactions to abort (roll back) to break the cycle and free up resources. The victim is often chosen based on factors like transaction priority or the least work performed to minimize overhead.

Example: In a deadlock cycle involving T1, T2, and T3, the system might choose T3 as the victim, terminate it, roll back its changes, and then restart it later, allowing T1 and T2 to proceed.

Failure Classification

Failure classification in Database Management Systems (DBMS) categorizes different types of events that can disrupt normal database operations and potentially lead to data loss or inconsistency. Understanding these classifications is crucial for implementing effective recovery mechanisms.

1.Transaction Failures

Transaction failures occur when an individual transaction cannot complete its execution successfully. These can be further divided into:

Logical Errors:

These arise from issues within the transaction's logic or data constraints.

Example: A transaction attempts to divide by zero, or it tries to insert a duplicate primary key value, violating a unique constraint.

System Errors:

These are caused by internal system issues that prevent a transaction from proceeding.
Example: A transaction encounters a deadlock with another transaction, or the system runs out of memory or other critical resources needed for the transaction's completion.

2. System Crashes

System crashes involve a failure of the entire DBMS or the underlying operating system, leading to the loss of volatile memory contents.

- **Example:** A sudden power outage causes the main memory (RAM) to lose all its data, including any unsaved changes from active transactions. Another example is a bug in the DBMS software or operating system that causes the system to halt unexpectedly.

3. Media Failures (Disk Failures)

Media failures occur when the non-volatile storage media (e.g., hard disk) holding the database becomes damaged or corrupted.

- **Example:** A physical disk crash renders the stored database files unreadable. This could be due to a head crash, bad sectors developing on the disk, or a complete failure of the storage device.

4. Application Failures

These failures originate from errors within the application programs interacting with the database.

- **Example:** An application contains a bug that causes it to send malformed SQL queries to the database, leading to unexpected errors or data corruption, or it misuses a database API, resulting in incorrect data manipulation.

5. Concurrency Control Failures

These failures arise when multiple transactions attempt to access and modify shared data concurrently, leading to inconsistencies if not properly managed.

- **Example:** A "lost update" scenario where two transactions simultaneously update the same data item, and one of the updates is overwritten and effectively lost due to improper locking mechanisms. Another example is a "dirty read" where a transaction reads uncommitted data from another transaction, which is later rolled back.

Storage structure

In Database Management Systems (DBMS), storage structures define how data is physically organized and stored on secondary storage devices. Key concepts include:

1. Memory Hierarchy:

A tiered system of storage devices, ranging from fast, expensive, and volatile (like CPU registers and cache) to slower, cheaper, and non-volatile (like hard drives and magnetic tapes). Data moves between these levels as needed.

Example:
When a query is executed, relevant data blocks are moved from a hard disk (secondary storage) into RAM (main memory) for processing by the CPU (using registers and cache).

2. Data Blocks/Pages:

The fundamental unit of data transfer between disk and main memory. Records are grouped into blocks, and the entire block is read or written at once.

Example:
A database table might have records for students. Instead of reading each student record individually, the DBMS reads a block containing multiple student records into memory.

3. Files of Records:

A collection of records, forming the basic abstraction for storing data in a DBMS.

Example:
A "Students" file would contain all the records for individual students, each record holding attributes like student ID, name, and address.

4. File Organizations:

Heap File:

Concept: Records are stored in an unordered fashion, typically appended to the end of the file.

Example: Imagine a log file where new entries are simply added to the end without any specific sorting. Searching for a specific record in a large heap file can be slow as it might require scanning the entire file.

Sorted File:

Concept: Records are stored in a specific order based on one or more key attributes.

Example: A "Students" file sorted by StudentID allows for efficient retrieval of students within a specific ID range using binary search.

Indexed File (e.g., B+ Trees, ISAM):

Uses an index data structure to provide fast access to records based on key values. The index contains pointers to the actual data records.

Example: A B+ tree index on the StudentID column in a "Students" table allows for quick retrieval of a student's record by traversing the tree to find the page containing the desired record.

Hashed File:

Concept: Uses a hash function to directly map a key value to a physical storage location, enabling very fast retrieval for exact key matches.

Example: A hash file on EmployeeID could directly calculate the disk block where an employee's record is stored, providing near-instant access.

5. Buffering and Buffer Manager:

A portion of main memory (buffer pool) used to temporarily store copies of disk blocks. The buffer manager is responsible for allocating and managing this buffer space, optimizing disk I/O.

Example:

When a frequently accessed data block is requested, the buffer manager checks if it's already in the buffer pool. If so, it can be accessed directly from memory, avoiding a slower disk read.

These storage structures and concepts are crucial for optimizing database performance, especially for data retrieval and manipulation operations.

Recovery and Atomicity

Recovery in DBMS

Recovery in a DBMS refers to the process of restoring the database to a consistent state after a system failure (e.g., power outage, hardware malfunction, software error). It works in conjunction with atomicity and durability to ensure data integrity.

Techniques for Recovery:

Log-Based Recovery:

The DBMS maintains a log file on stable storage, recording all changes made by transactions (e.g., transaction ID, data item affected, old value, new value).

- **Undo:** If a transaction failed before committing, its changes are rolled back using the log's "old values."
- **Redo:** If a committed transaction's changes were not fully written to disk before a crash, they are reapplied using the log's "new values."

Shadow Paging:

This technique involves maintaining two copies of the page table: a current page table and a shadow page table. Changes are made to new pages, and the current page table is updated to point to these new pages. If the transaction commits, the current page table becomes the new shadow page table. If it aborts, the system reverts to the original shadow page table, effectively discarding the changes.

Example: Using the bank transfer example with log-based recovery:

- Before deducting from A, a log record <start T> is written.
- Before updating A, a log record <T, A, old_value_A, new_value_A> is written.
- Before updating B, a log record <T, B, old_value_B, new_value_B> is written.
- Upon successful completion, a log record <commit T> is written.

If a crash occurs after updating A but before updating B and committing, the recovery manager would find the <start T> record but no <commit T> record. It would then use the log to undo the changes made to A, restoring its original balance, thus upholding atomicity and consistency.

Atomicity in DBMS

Atomicity, one of the ACID properties (Atomicity, Consistency, Isolation, Durability), ensures that a transaction is treated as a single, indivisible unit of work. This means that either all operations within a transaction are completed successfully, or none of them are. There is no partial completion.

Example: Consider a bank transfer transaction where money is moved from account A to account B. This transaction involves two main steps: Deducting the amount from account A and Adding the amount to account B.

If the system crashes after step 1 but before step 2, atomicity ensures that the entire transaction is rolled back. Account A's balance is restored to its original state, and the deduction never appears to have happened. This prevents the database from entering an inconsistent state where money is deducted from one account but not credited to another.

Log-Based Recovery

Log-based recovery in a Database Management System (DBMS) is a technique used to ensure data consistency and durability in the event of a system failure. It relies on maintaining a detailed log of all database modifications, which can then be used to restore the database to a consistent state.

Working Procedure

Logging Changes:

Whenever a transaction modifies data in the database, the DBMS records this operation in a sequential log file. Each log record typically includes:

Transaction ID: Identifies the transaction performing the operation.

Data Item ID: Identifies the specific data item being modified.

Old Value: The value of the data item before the modification.

New Value: The value of the data item after the modification.

Operation Type: (e.g., insert, delete, update).

Transaction Status: Records the start, commit, or abort of a transaction.

Stable Storage:

The log file is stored on stable storage (e.g., disk) to ensure its persistence even if the main memory or database is lost due to a crash.

Recovery Process:

In case of a system failure (e.g., power outage, software error), the DBMS uses the log file to recover the database:

Redo Operations: Transactions that committed before the crash but whose changes might not have been fully written to disk are "redone" using the "new value" entries in the log. This ensures durability.

Undo Operations: Transactions that were in progress and did not commit before the crash are "undone" or rolled back using the "old value" entries in the log. This ensures atomicity.

Example:

Consider a bank database with an account balance for customer A (Account A) and customer B (Account B).

Transaction T1 starts: Transfer $100 from Account A to Account B.

Log record: <T1, Start>
T1 updates Account A: Decreases balance by $100.

Log record: <T1, Account A, Old_Value_A, New_Value_A> (e.g., <T1, Account A, 500, 400>)
T1 updates Account B: Increases balance by $100.

Log record: <T1, Account B, Old_Value_B, New_Value_B> (e.g., <T1, Account B, 300, 400>)
T1 commits:
Log record: <T1, Commit>

Scenario: System Crash before T1 commits

If a system crash occurs after the updates to Account A and Account B but before the <T1, Commit> log record is written, the recovery process would:

- **Undo T1:** Since T1 did not commit, the DBMS would use the log records to restore Account A to Old_Value_A and Account B to Old_Value_B, effectively rolling back the incomplete transaction.

Scenario: System Crash after T1 commits

If a system crash occurs after the <T1, Commit> log record is written but before the changes are fully written to disk, the recovery process would:

- **Redo T1:** The DBMS would find the <T1, Commit> record in the log and then use the "new value" entries for Account A and Account B to ensure these changes are applied to the database, ensuring the transaction's durability.

Recovery with concurrent transactions

Recovery with concurrent transactions in DBMS ensures data integrity after crashes by using logs (write-ahead logging) to redo committed changes and undo aborted ones, maintaining ACID properties (Atomicity, Consistency, Isolation, Durability).

Key concepts include **Undo/Redo Lists**, **Checkpoints**, and **Restart Recovery**, which process logs to restore the database to a consistent state, making concurrent operations safe by hiding intermediate results from others.

Key Concepts:

1. **Write-Ahead Logging (WAL)**: Before any data change hits the disk, its log record (start, commit, update with old/new values) is written to stable storage.
2. **Undo/Redo Lists (Analysis Phase)**: During recovery, the log is scanned:

   Undo List: Transactions that started but didn't commit (active/failed) go here (need rollback).

   Redo List: Transactions that committed but whose changes might not be on disk (due to crash) go here (need reapplication).
3. **Checkpoints**: Periodically, a `<checkpoint>` record is written, flushing buffers to disk, making recovery faster by defining a point before which all committed transactions are already in the DB.
4. **Restart Recovery**:

   Analysis: Determine Undo/Redo Lists from logs since the last checkpoint.

   Redo: Reapply updates for committed transactions (Redo List) to ensure durability.

   Undo: Rollback uncommitted transactions (Undo List) to ensure atomicity.

Example Scenario (Bank Transfer):

Database: Account A ($100), Account B ($200). Total: $300.

Transaction T1 (Transfer $50 from A to B):

`T1_Start`, `A_Old=100`, `A_New=50` (Log)

`B_Old=200, B_New=250` (Log)

`T1_Commit` (Log)

- **Transaction T2 (Transfer $100 from C to A)**:

  `T2_Start`, `C_Old=..., C_New=...` (Log)

  `A_Old=50, A_New=150` (Log)

  `T2_Commit` (Log)

- **System Crash**: Crash occurs after T1 commits but before T2 commits, and maybe after T2 writes to C but before it updates A (or even before T2 writes anything).

  Recovery Steps (After Crash):
  Analyze: Scan logs from the last checkpoint.
  Find `T1_Start`, `T1_Commit`, `T2_Start`, `T2_Commit` (or just `T2_Start` if it failed mid-way).

  Undo List: {T2} (if incomplete).

  Redo List: {T1} (if T1's changes not fully on disk).

  Redo: Reapply T1's changes to ensure data is consistent (e.g., A=50, B=250).**Undo**: Rollback T2 (undo its changes to C, etc.) to maintain consistency (A=50, B=250).

  The system ends up consistent (like T1 completed, T2 didn't) even with concurrent operations and a crash, ensuring atomicity and durability.