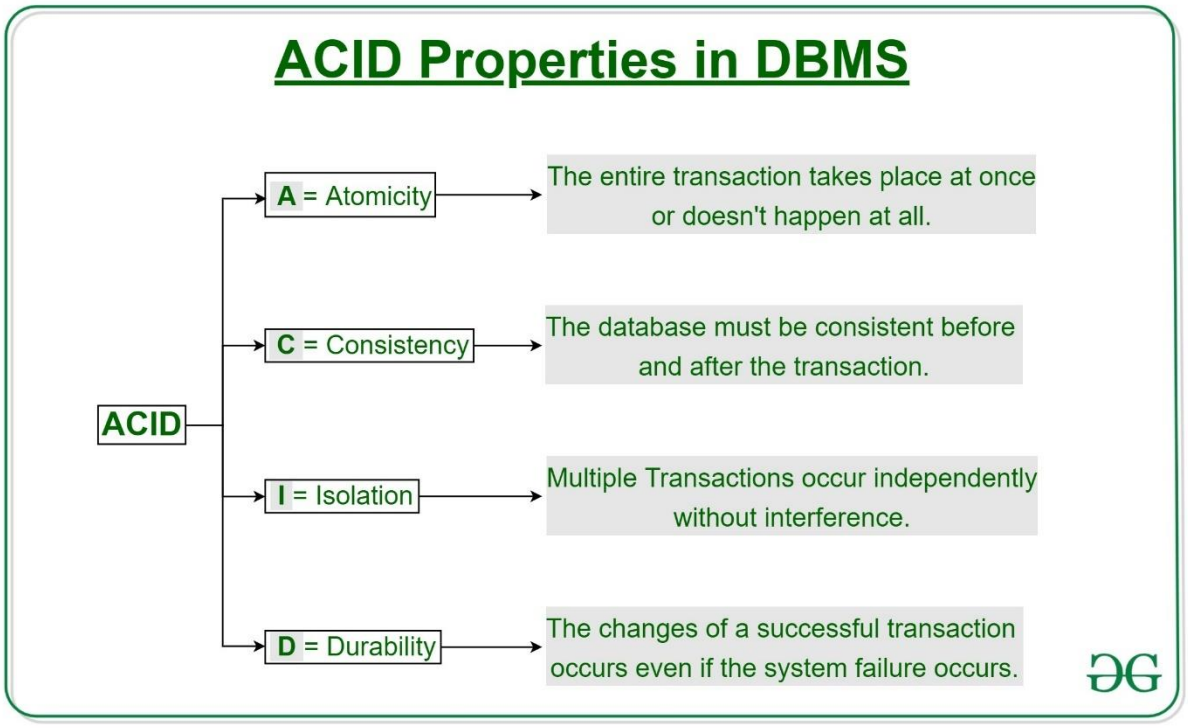
Lecture 6: Transactions, Concurrency & Recovery

ACID Properties in DBMS

In the world of **Database Management Systems** (DBMS), transactions are fundamental operations that allow us to modify and retrieve data. However, to ensure the integrity of a database, it is important that these transactions are executed in a way that maintains consistency, correctness, and reliability. This is where the **ACID properties** come into play.

ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These four key properties define how a transaction should be processed in a reliable and predictable manner, ensuring that the database remains consistent, even in cases of failures or concurrent accesses.



What Are Transactions in DBMS?

A **transaction** in DBMS refers to a sequence of operations performed as a single unit of work. These operations may involve reading or writing data to the database. To maintain data integrity, DBMS ensures that each transaction adheres to the **ACID properties**. Think of a transaction like an ATM withdrawal. When we withdraw money from our account, the transaction involves several steps:

- Checking your balance.
- Deducting the money from your account.
- Adding the money to the bank's record.

For the transaction to be successful, **all steps** must be completed. If any part of this process fails (e.g., if there's a system crash), the entire transaction should fail, and no data should be altered. This ensures the database remains in a **consistent** state.

The Four ACID Properties

1. Atomicity: "All or Nothing"

Atomicity ensures that a transaction is **atomic**, it means that either the entire transaction completes fully or doesn't execute at all. There is no in-between state i.e. transactions do not occur partially. If a transaction has multiple operations, and one of them fails, the whole transaction is rolled back, leaving the database unchanged. This avoids partial updates that can lead to inconsistency.

- **Commit**: If the transaction is successful, the changes are permanently applied.
- **Abort/Rollback**: If the transaction fails, any changes made during the transaction are discarded.

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

Before: X : 500	Y : 200
Transaction T	
T1	T2
Read (X) X: = X - 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

Example

If the transaction fails after completion of T1 but before completion of T2 , the database would be left in an inconsistent state. With Atomicity, if any part of the transaction fails, the entire process is rolled back to its original state, and no partial changes are made.

2. Consistency: Maintaining Valid Data States

Consistency ensures that a database remains in a valid state before and after a transaction. It guarantees that any transaction will take the database from one consistent state to another, maintaining the rules and constraints defined for the data. In simple terms, a transaction should only take the database from one **valid** state to another. If a transaction violates any database rules or constraints, it should be rejected, ensuring that only consistent data exists after the transaction.

Example: Suppose the sum of all balances in a bank system should always be constant. Before a transfer, the total balance is **$700**. After the transaction, the total balance should remain $700. If the transaction fails in the middle (like updating one account but not the other), the system should maintain its consistency by rolling back the transaction

Total before T occurs = 500 + 200 = 700 .

Total after T occurs = 400 + 300 = 700 .

X = 500 Rs	Y = 500 Rs
T	T''
Read (X) X :=X*100 Write (X) Read (Y) Y:= Y - 50 Write (Y)	Read (X) Read (Y) Z:= X + Y Write (Z)

3. Isolation: Ensuring Concurrent Transactions Don't Interfere

This property ensures that **multiple transactions** can occur concurrently without leading to the **inconsistency** of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

This property ensures that when multiple transactions run at the same time, the result will be the same as if they were run one after another in a specific order. This property prevents issues such as **dirty reads** (reading uncommitted data), **non-repeatable reads** (data changing between two reads in a transaction), and **phantom reads** (new rows appearing in a result set after the transaction starts).

Example: Let's consider two transactions:Consider two transactions **T** and **T''.**

- **X = 500**, **Y = 500**

X = 500 Rs	Y = 500 Rs
T	T''
Read (X) X :=X*100 Write (X) Read (Y) Y:= Y - 50 Write (Y)	Read (X) Read (Y) Z:= X + Y Write (Z)

Transaction T:

- **T** wants to transfer $50 from **X** to **Y**.
- **T** reads **Y** (value: 500), deducts $50 from **X** (new **X = 450**), and adds $50 to **Y** (new **Y = 550**).

Transaction T'':

- **T''** starts and reads **X** (value: 500) and **Y** (value: 500), then calculates the sum: **500 + 500 = 1000**.

But, by the time **T** finishes, **X** and **Y** have changed to **450** and **550** respectively, so the correct sum should be **450 + 550 = 1000**. **Isolation** ensures that **T''** should not see the old values of **X** and **Y** while **T** is still in progress. Both transactions should be independent, and **T''** should only see the final state after **T** commits. This prevents inconsistent data like the incorrect sum calculated by **T''**

4. Durability: Persisting Changes

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in **non-volatile memory**. In the event of a failure, the DBMS can recover the database to the state it was in after the last committed transaction, ensuring that no data is lost.

Example: After successfully transferring money from Account A to Account B, the changes are stored on disk. Even if there is a crash immediately after the commit, the transfer details will still be intact when the system recovers, ensuring durability.

How ACID Properties Impact DBMS Design and Operation

The ACID properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

1. Data Integrity and Consistency

ACID properties safeguard the **data integrity** of a DBMS by ensuring that transactions either complete successfully or leave no trace if interrupted. They prevent **partial updates** from corrupting the data and ensure that the database transitions only between valid states.

2. Concurrency Control

ACID properties provide a solid framework for **managing concurrent transactions**. Isolation ensures that transactions do not interfere with each other, preventing data anomalies such as lost updates, temporary inconsistency, and uncommitted data.

3. Recovery and Fault Tolerance

Durability ensures that even if a system crashes, the database can recover to a consistent state. Thanks to the **Atomicity** and **Durability** properties, if a transaction fails midway, the database remains in a consistent state.

Property	Responsibility for maintaining properties
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery

Advantages of ACID Properties in DBMS

1. **Data Consistency:** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity:** It maintains the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control:** ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery:** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS

1. **Performance Overhead**: ACID properties can introduce performance costs, especially when enforcing isolation between transactions or ensuring atomicity.
2. **Complexity**: Maintaining ACID properties in distributed systems (like microservices or cloud environments) can be complex and may require sophisticated solutions like distributed locking or transaction coordination.
3. **Scalability Issues**: ACID properties can pose scalability challenges, particularly in systems with high transaction volumes, where traditional relational databases may struggle under load.

ACID in the Real World: Where Is It Used?

In modern applications, ensuring the **reliability and consistency** of data is crucial. ACID properties are fundamental in sectors like:

- **Banking**: Transactions involving money transfers, deposits, or withdrawals must maintain strict consistency and durability to prevent errors and fraud.
- **E-commerce**: Ensuring that inventory counts, orders, and customer details are handled correctly and consistently, even during high traffic, requires ACID compliance.
- **Healthcare**: Patient records, test results, and prescriptions must adhere to strict consistency, integrity, and security standards.

Concurrency problems in DBMS Transactions

Concurrency control is an essential aspect of database management systems (DBMS) that ensures transactions can execute concurrently without interfering with each other. However, concurrency control can be challenging to implement, and without it, several problems can arise, affecting the consistency of the database. In this article, we will discuss some of the concurrency problems that can occur in DBMS transactions and explore solutions to prevent them.

When **multiple transactions** execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in a database environment.

The five concurrency problems that can occur in the database are:

- Temporary Update Problem
- Incorrect Summary Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem

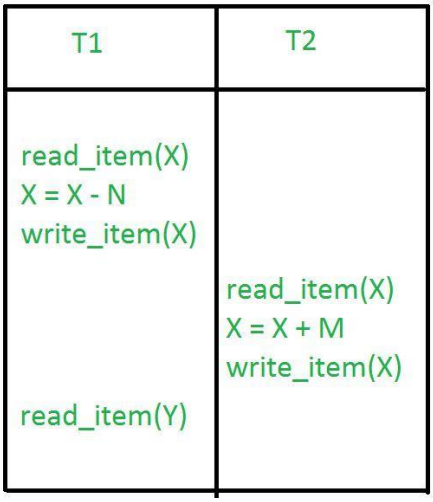
Concurrency control ensures the consistency and integrity of data in databases when multiple transactions are executed simultaneously. Understanding issues like lost updates, dirty reads, and non-repeatable reads is crucial when studying DBMS.

These are explained as following below.

Temporary Update Problem:

Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Example:



In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

Incorrect Summary Problem:

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:

T1	T2
	sum = 0 read_item(A) sum = sum + A
read_item(X) X = X - N write_item(X)	
	read_item(X) sum = sum + X read_item(Y) sum = sum + Y
read_item(Y) Y = Y + N write_item(Y)	

In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

Lost Update Problem:

In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

T1	T2
read_item(X) X = X + N	
	X = X + 10 write_item(X)

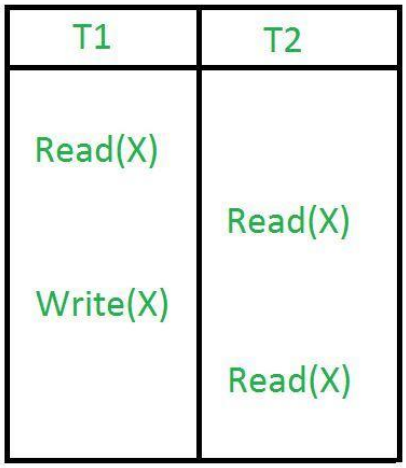
In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*. Therefore, the update done

by transaction 2 will be lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

Unrepeatable Read Problem:

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

Example:

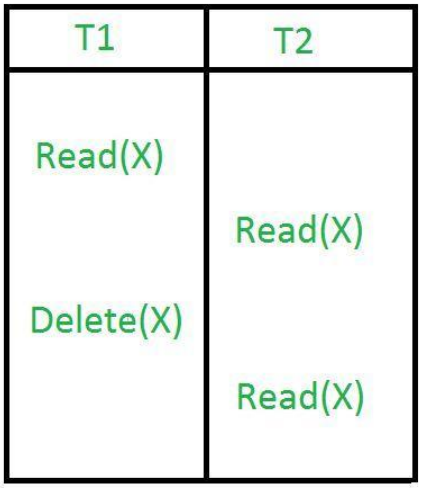


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

Phantom Read Problem:

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

Example:



In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Transactions & Concurrency Control

In database systems, **transactions** are sequences of operations that perform tasks like reading, writing, or updating data. To ensure data integrity and consistency, especially in multi-user environments, **concurrency control** mechanisms are employed.

Serializability

Serializability is the highest isolation level in transaction management. It ensures that the outcome of executing transactions concurrently is equivalent to some serial execution of those transactions.

There are two primary types of serializability:

- **Conflict Serializability**: Ensures that the schedule of transactions can be transformed into a serial schedule by swapping non-conflicting operations.
- **View Serializability**: Ensures that the transactions' reads and writes can be reordered to form a serial schedule without violating the transaction's view of the data.
- **Locking Techniques**

Locking mechanisms are pivotal in concurrency control to prevent conflicts and ensure data consistency.

Two-Phase Locking (2PL)

The **Two-Phase Locking Protocol** is a concurrency control method that guarantees conflict-serializability. It operates in two phases:

1. **Growing Phase**: A transaction may obtain locks but cannot release any.
2. **Shrinking Phase**: A transaction may release locks but cannot obtain any new ones

This protocol ensures that once a transaction releases a lock, it cannot acquire any new locks, preventing cycles and ensuring serializability.

Shared & Exclusive Locks

- **Shared Lock (S-Lock)**: Allows multiple transactions to read (but not write) a data item simultaneously.
- **Exclusive Lock (X-Lock)**: Allows only one transaction to write to a data item, preventing other transactions from accessing it.

Log-Based Recovery

Log-based recovery employs a transaction log to record all operations performed on the database. This log includes details such as the start of a transaction, the operations executed, and the commit or abort status.

How It Works:

1. **Logging Operations**: Before any changes are made to the database, the corresponding operations are written to the transaction log.
2. **Commit and Abort**: Upon transaction commit, the log is updated to reflect the successful completion. If a transaction is aborted, the log records the rollback actions.
3. **Recovery Process**:
    - **Undo**: For transactions that were active but did not commit before a crash, the system reverts their changes.

- **Redo**: For transactions that committed before the crash, their changes are reapplied to ensure durability.

This method ensures that the database can be restored to a consistent state by replaying or rolling back operations as necessary.

Shadow Paging

Shadow Paging is an alternative recovery technique that avoids the overhead of logging by maintaining two page tables: the **current** and the **shadow**.

How It Works:

1. **Initialization**: At the start of a transaction, the current page table is copied to create the shadow page table.
2. **Transaction Execution**: All updates during the transaction are made to the current page table, while the shadow table remains unchanged.
3. **Commit**: Upon commit, the shadow page table is updated to point to the new pages, effectively making the changes permanent.
4. **Rollback**: If a crash occurs, the shadow page table is used to restore the database to its state before the transaction.

This technique ensures atomicity and durability without the need for undo or redo operations.

Comparison

Feature	Log-Based Recovery	Shadow Paging
Logging	Extensive logging of all operations	Minimal, only page table changes
Recovery Process	Undo and redo operations	Switching page tables
Performance	Can be slower due to logging overhead	Faster, but may require more storage
Concurrency	Supports multiple concurrent transactions	More challenging to implement concurrency
Storage Overhead	Depends on log size	Requires additional storage for shadow pages

Choosing the Right Technique

- **Log-Based Recovery**: Suitable for systems requiring high concurrency and where the overhead of logging is manageable.
- **Shadow Paging**: Ideal for systems with lower write activity and where simplicity and performance are prioritized.

Both methods have their advantages and trade-offs, and the choice between them depends on the specific requirements and constraints of the database system.