Desirable properties of transactions.

- Transactions in a Database Management System (DBMS) should exhibit a set of desirable properties to ensure data consistency, integrity, and reliability.
- These ACID properties collectively ensure that database transactions maintain data integrity and consistency, even in the face of concurrent access, system failures, or errors.
- They are fundamental for building robust and reliable database applications, especially in scenarios where data accuracy and reliability are critical, such as financial systems, e-commerce platforms, and healthcare databases.
- These properties are often referred to as the ACID properties, which stands for Atomicity, Consistency, Isolation, and Durability. Let's explain each of these properties with examples:

Atomicity: Definition: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within a transaction are executed successfully, or none of them are executed at all. If any part of a transaction fails, the entire transaction is rolled back, leaving the database in its previous consistent state.

Example: Consider a banking application where a customer wants to transfer money from one account to another. The transaction involves two operations: deducting money from one account and crediting it to another. If the money deduction succeeds but the crediting operation fails due to a system crash, atomicity ensures that the deduction is also rolled back, preventing inconsistencies in the account balances.

Consistency: Definition: Consistency guarantees that a transaction brings the database from one consistent state to another consistent state. It ensures that a transaction must maintain all integrity constraints, business rules, and data validity during its execution.

Example: In an online shopping system, if a customer places an order for an item, the database should maintain the consistency of the inventory. The system should not allow an order to be placed if there is insufficient stock to fulfill the order, ensuring that the inventory remains consistent.

Isolation: Definition: Isolation ensures that the execution of one transaction is isolated from the execution of other concurrent transactions. Transactions should not interfere with each other, and their intermediate states should remain invisible to other transactions until they are committed.

Example: Suppose two customers simultaneously attempt to purchase the last available item in an online store. Both transactions need to check the inventory and reserve the item. Isolation ensures that the transactions do not see each other's reservations or inadvertently oversell the item.

Durability: Definition: Durability guarantees that once a transaction is successfully committed, its changes to the database are permanent and will survive system failures (e.g., crashes or

power outages). Even in the event of a system failure, the database should be able to recover and reflect the committed changes.

Example: After a successful online payment, the order information and payment details should be permanently stored in the database. Even if the database server crashes immediately after the payment confirmation, durability ensures that the payment is not lost, and the order can be retrieved when the system is back online.

Transaction Support in SQL

- Transaction support in SQL refers to the set of features and mechanisms that allow you to group one or more SQL statements into a single logical unit of work called a "transaction." Transactions ensure the consistency and integrity of a database by guaranteeing that a series of related operations either all succeed or all fail, leaving the database in a consistent state.
- An SQL transaction is a logical unit of work (i.e., a single SQL statement).

Simple example of an SQL transaction using the SQL syntax for a fictional banking application. This transaction transfers funds from one account to another while ensuring the atomicity, consistency, isolation, and durability (ACID properties) of the operation:

```
-- Start a new transaction
BEGIN TRANSACTION;

-- Step 1: Deduct $100 from Account A
UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountNumber = 'A';

-- Step 2: Add $100 to Account B
UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountNumber = 'B';

-- Check if the transaction should be committed or rolled back
IF (SELECT COUNT(*) FROM Accounts WHERE AccountNumber = 'A' AND Balance >= 0) = 1
  AND (SELECT COUNT(*) FROM Accounts WHERE AccountNumber = 'B' AND Balance >= 0) = 1
BEGIN
  -- Both updates were successful, commit the transaction
  COMMIT;
  PRINT 'Transaction committed successfully.';
END
ELSE
BEGIN
  -- At least one update failed, rollback the transaction
  ROLLBACK;
  PRINT 'Transaction rolled back due to insufficient funds.';
```

END;

In this example:

- We begin a new transaction using BEGIN TRANSACTION.
- We perform two UPDATE statements to deduct $100 from Account A and add $100 to Account B.
- We use conditional checks to ensure that both updates were successful (i.e., Account A and B have sufficient funds after the transaction). If both conditions are met, we commit the transaction using COMMIT. Otherwise, if there are insufficient funds in either account, we roll back the transaction using ROLLBACK.

Why Concurrency Control Is Needed

Concurrency control is a critical component of a Database Management System (DBMS) for several reasons:

Data Integrity: In a multi-user DBMS environment, multiple users or applications can access and manipulate the same data simultaneously. Without concurrency control, there is a risk of data becoming inconsistent or corrupted. For example, two users might try to update the same record simultaneously, leading to a situation where one user's changes overwrite the other's.

Isolation of Transactions: Concurrency control ensures that transactions (sets of database operations) execute in isolation from one another. This means that one transaction's operations should not interfere with another's until they are both complete. This isolation is essential to maintain the integrity of the database and to ensure that each transaction's outcome is consistent with the database's constraints and rules.

Resource Management: DBMSs manage system resources such as CPU, memory, and disk I/O. Without concurrency control, multiple transactions competing for these resources can lead to inefficiencies, contention, and resource conflicts. Concurrency control mechanisms help allocate and manage these resources efficiently, ensuring fair access and minimizing contention.

Lack of concurrency control leads to following problems:

1. **The Lost Update Problem.**

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

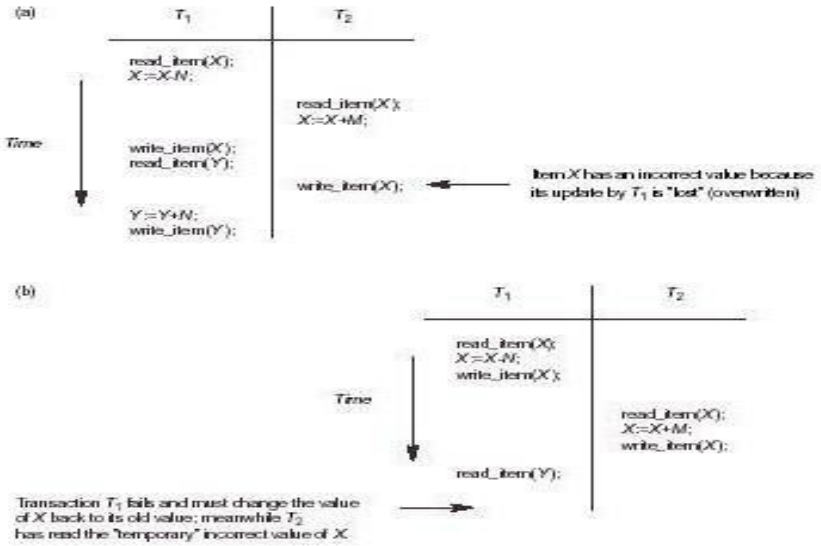
Example: transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved then the final value of item X is incorrect, because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

2. **The Temporary Update (or Dirty Read) Problem.**

This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

Example: Figure 19.03(b) shows an where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the "temporary" value of X, which will not be recorded permanently in the database because of the failure of T1. The value of item X that is read by T2 is called dirty data, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.



d

3. **The Incorrect Summary Problem.**

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure 19.03(c) occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

Phase Locking System

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items.

- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Generally, there is one lock for each data item in the database.

- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.
- Several types of locks are used in concurrency control.
- Binary locks,
- *shared/exclusive* locks—also known as *read/write* locks—

Binary Locks. A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item *X*. If the value of the lock on *X* is 1, item *X cannot be accessed* by a database operation that requests the item. If the value of the lock on *X* is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item *X* as **lock(*X*)**.

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item *X* by first issuing a **lock_item(*X*)** operation. If LOCK(*X*) = 1, the transaction is forced to wait. If LOCK(*X*) = 0, it is set to 1 (the
transaction **locks** the item) and the transaction is allowed to access item *X*. When the transaction is through using the item, it issues an **unlock_item(*X*)** operation, which sets LOCK(*X*) back to 0

Lock and unlock operations for binary locks.

```
lock_item(X):
B: if LOCK(X) = 0 (*item is unlocked*)
then LOCK(X) ←1 (*lock the item*)
else
begin
wait (until LOCK(X) = 0
and the lock manager wakes up the transaction);
go to B
end;
unlock_item(X):
LOCK(X) ← 0; (* unlock the item *)
if any transactions are waiting
then wakeup one of the waiting transactions;
```