DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 1
TRANSACTION CONCEPT AND STATE

THE CONCEPT OF A TRANSACTION:

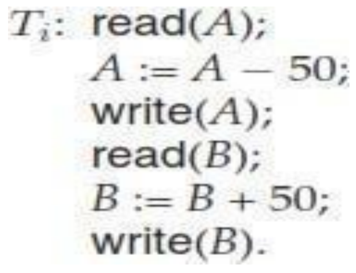
A transaction is a set of operations used to perform a logical unit of work. It is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data- manipulation language or programming language.

Transactions access data using two operations:

• ***read(X),*** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

• ***write(X)***, which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as:



```
Ti: read(A);
    A := A − 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

ACID Properties:

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. These ensure tomaintain data in the face of concurrent access and system failures:

Atomicity: Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(A)operation but before the write(B)operation. In this case, the values of accounts A and B reflected in the database are

$950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum A + B is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an inconsistent state. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account A is $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the transaction-management component.

Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

Ex: the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to

B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as they allow multiple transactions to execute concurrently. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

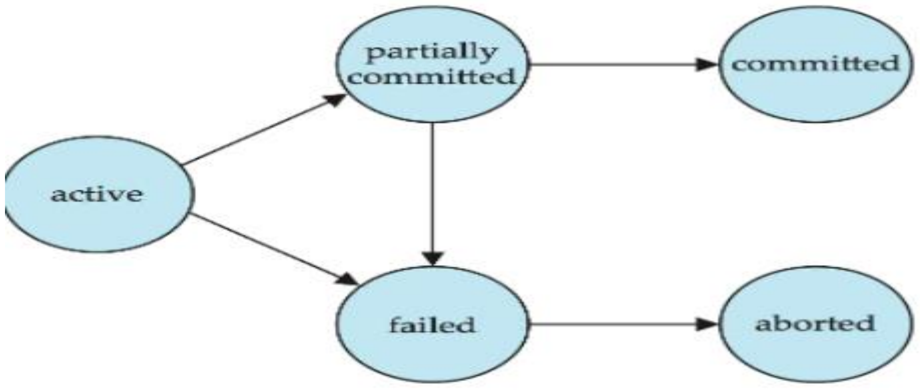
1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

Transaction State

Transaction State Diagram: A simple abstract transaction model is shown in fig below:



A transaction must be in one of the following states:

• *Active*, the initial state; the transaction stays in this state while it is executing

• *Partially committed*, after the final statement has been executed

• *Failed*, after the discovery that normal execution can no longer proceed

• *Aborted*, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

• *Committed*, after successful completion.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state. As mentioned earlier, we assume for now that failures do not result in loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

→It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

→It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.