Unit – III Transaction Management

Transactions in DBMS

DBMS

DBMS stands for Database Management System, which is a tool or software that is used for the creation, deletion, or manipulation of the current database.

Transaction

Any logical work or set of works that are done on the data of a database is known as a transaction. Logical work can be inserting a new value in the current database, deleting existing values, or updating the current values in the database.

For example, adding a new member to the database of a team is a transaction.

To complete a transaction, we have to follow some steps which make a transaction successful. For example, we withdraw the cash from ATM is an example of a transaction, and it can be done in the following steps:

- Initialization if transaction
- Inserting the ATM card into the machine
- Choosing the language
- Choosing the account type
- Entering the cash amount
- Entering the pin
- Collecting the cash
- Aborting the transaction

So, in the same way, we have three steps in the DBMS for a transaction which are the following:

- Read Data
- Write Data
- Commit

We can understand the above three states by an example. Let suppose we have two accounts, account1, and account2, with an initial amount as 1000Rs. each. If we want to transfer Rs.500 from account1 to account2, then we will commit the transaction.

- All the account details are in secondary memory so that they will be brought into primary memory for the transaction.
- Now we will read the data of account1 and deduct the Rs.500 from the account1. Now, account1 contains Rs.500.
- Now we will read the data of the account2 and add Rs.500 to it. Now, account2 will have Rs.1500.
- In the end, we will use the commit command, which indicates that the transaction has been successful, and we can store the changes in secondary memory.
- If, in any case, there is a failure before the commit command, then the system will be back into its previous state, and no changes will be there.

During the complete process of a transaction, there are a lot of states which are described below:

Active State:

When the transaction is going well without any error, then this is called an active state. If all the operations are good, then it goes to a partially committed state, and if it fails, then it enters into a failed state.

Partially Committed State:

All the changes in the database after the read and write operation needs to be reflected in permanent memory or database. So, a partially committed state system enters into a committed state for the permanent changes, and if there is any error, then it enters into a failed state.

Failed State:

If there is any error in hardware or software which makes the system fail, then it enters into the failed state. In the failed state, all the changes are discarded, and the system gets its previous state which was consistent.

Aborted State:

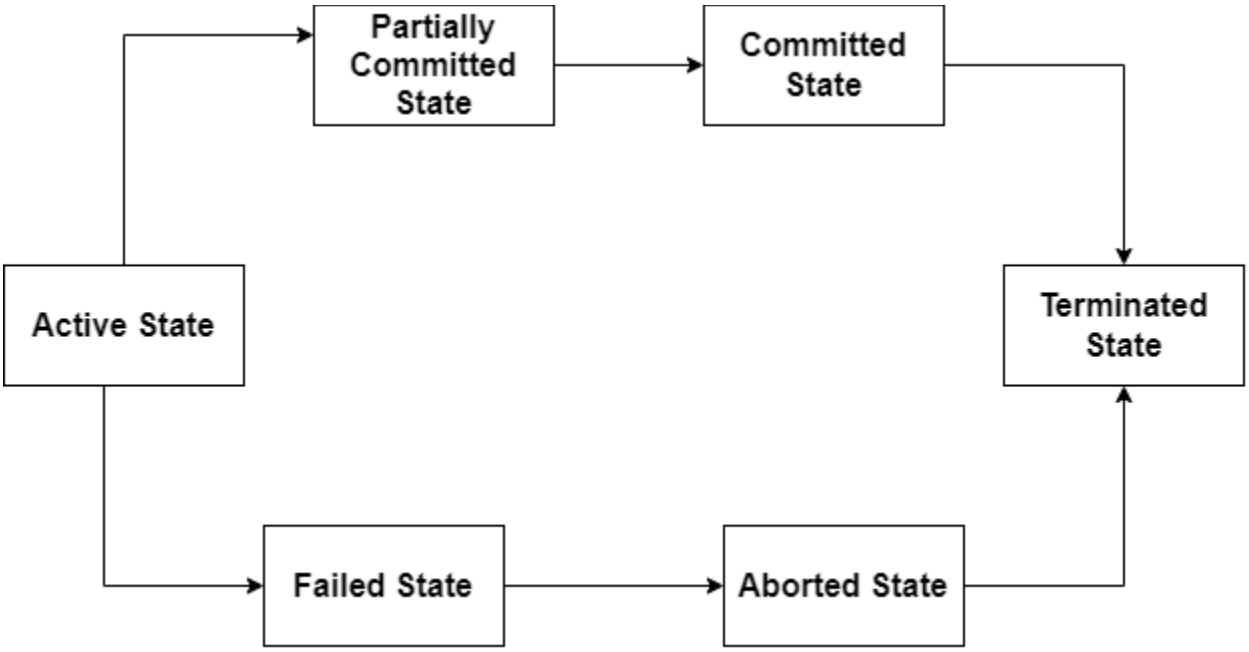
If there is any failure during the execution, then the system goes from failed to an aborted state. From an aborted state, the transaction will start its execution from a fresh start or from a newly active state.

Committed State:

If the execution of a transaction is successful, the changes are made into the main memory and stored in the database permanently, which is called the committed state.

Terminated State:

If the transaction is in the aborted state(failure) or committed state(success), then the execution stops, and it is called the terminated state.



Properties of Transaction

There are four properties of a transaction that should be maintained during the transaction.

- **Atomicity:**

It means either a transaction will take place, or it will fail. There will not be any middle state like partial completion.

- **Consistency:**

The database should be consistent before and after the transaction. Correctness and integrity constraints should be maintained during the transaction.

- **Isolation:**

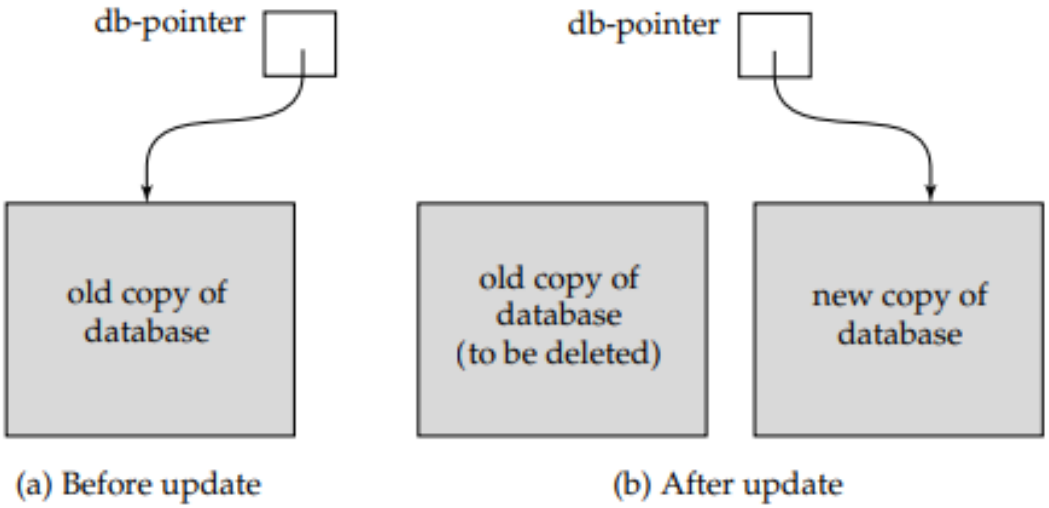
This property means multiple transactions can occur at the same time without affecting each other. If one transaction is occurring, then it should not bring any changes in the data for the other transaction, which is occurring concurrently.

- **Durability:**

It means if there is a successful transaction, then all changes should be permanent, so if there is any system failure, we will be able to retrieve the updated data.

Implementation of Atomicity and Durability

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the shadow copy scheme. This scheme, which is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database. In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.



The transaction is said to have been committed at the point where the updated dbpointer is written to disk. We now consider how the technique handles transaction and system failures. First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected. We can abort the transaction by just deleting the new copy of the database. Once the transaction has been committed, all the updates that it performed are in the database pointed to by dbpointer. Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure. Now consider the issue of system failure. Suppose that the system fails at any time before the updated db-pointer is written

to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database. Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk. Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

• **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

Consider again the simplified banking system of Section 15.1, which has several accounts, and a set of transactions that access and update those accounts. Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as

T1: read(A);

A := A − 50;

write(A);

read(B);

B := B + 50;

write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as T2: read(A);

temp := A * 0.1;

A := A − temp;

write(A); read(B);

B := B + temp;

write(B).

What is the term serializability in DBMS?

A **schedule** is serialized if it is equivalent to a serial schedule. A concurrent schedule must ensure it is the same as if executed serially means one after another. It refers to the sequence of actions such as read, write, abort, commit are performed in a serial manner.

Example

Let's take two **transactions** T1 and T2,

If both transactions are performed without interfering each other then it is called as serial schedule, it can be represented as follows −

T1	**T2**
READ1(A)	
WRITE1(A)	
READ1(B)	

T1	T2
C1	
	READ2(B)
	WRITE2(B)
	READ2(B)
	C2

Non serial schedule − When a transaction is overlapped between the transaction T1 and T2.

Example

Consider the following example −

T1	T2
READ1(A)	
WRITE1(A)	
	READ2(B)
	WRITE2(B)
READ1(B)	
WRITE1(B)	
READ1(B)	

Types of serializability

There are two types of serializability −

View serializability

A schedule is view-serializability if it is viewed equivalent to a serial schedule.

The rules it follows are as follows −

- T1 is reading the initial value of A, then T2 also reads the initial value of A.
- T1 is the reading value written by T2, then T2 also reads the value written by T1.
- T1 is writing the final value, and then T2 also has the write operation as the final value.

Conflict serializability

It orders any conflicting operations in the same way as some serial execution. A pair of operations is said to conflict if they operate on the same data item and one of them is a write operation.

That means

- Readi(x) readj(x) - non conflict read-read operation
- Readi(x) writej(x) - conflict read-write operation.
- Writei(x) readj(x) - conflict write-read operation.
- Writei(x) writej(x) - conflict write-write operation.

Conflict serializability orders any conflicting operations in the same way as some serial execution. A pair of operations is said to conflict if they operate on the same data item and one of them is a write operation.

That means

- Readi(x) readj(x) - non conflict read-read operation
- Readi(x) writej(x) - conflict read-write operation.
- Writei(x) readj(x) - conflic t write-read operation.
- Writei(x) writej(x) - conflict write-write operation.

Precedence graph

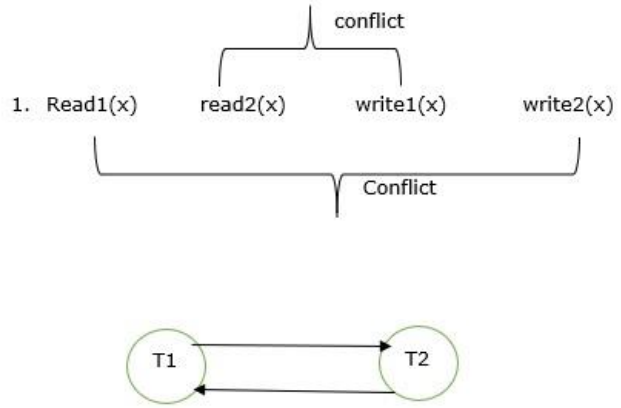
It is used to check conflict serializability.

The steps to check conflict serializability are as follows −

- For each transaction T, put a node or vertex in the graph.
- For each conflicting pair, put an edge from Ti to T_j.
- If there is a cycle in the graph then schedule is not conflict serializable else schedule is conflict serializable.

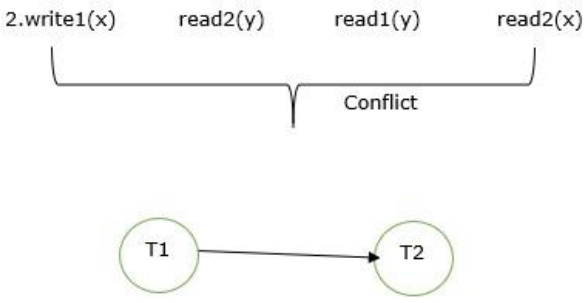
Example 1

The cycle is present so it is not conflict serializable.



Example 2

The cycle is not present, so it is conflict serializable.



Example 3

The cycle is not present, so it is conflict serializable.