In Database Management Systems (DBMS), a transaction refers to a set of logically related operations that are performed on a database. These operations are the result of a user's request to access and modify the contents of the database.

A transaction 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 (typically SQL), or programming language (e.g., C++, or Java), with embedded database accesses in JDBC or ODBC.

Transaction Management

Data Access

- The blocks residing on the disk are **physical blocks**
- The blocks residing temporarily in main memory are **buffer blocks**
- The area of memory where blocks reside temporarily is called the **disk buffer**

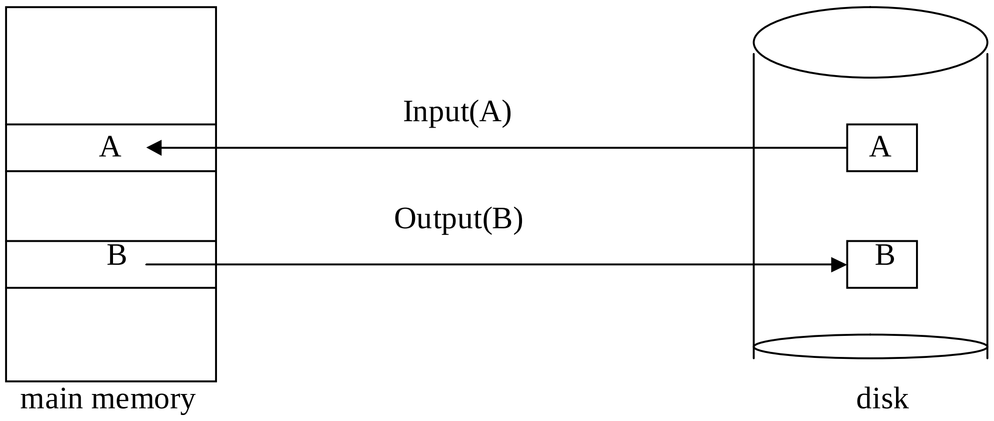


Figure : Block storage operations

Block movements between disk and main memory are initiated through the following two operations :

- Input(A) transfers the physical block B to main memory
- Output(B) transfers the buffer block B to the disk and replaces the physical block

Transaction

- Each transaction T_i has a private work area
- The system creates this work area when the transaction initiates
- Removes it when the transaction either commits or abort it

Read (X) : Reads a database item X into program variable. To simplify notation, we may assume that *the program variable is also named X.*

i. if a block B_x on which X resides is not in main memory, the system issues Input(B_x).
ii. The value of X from the buffer block is assigned to x_i

Write (X): Writes the value of program variable X into the database item named X.

Undo (T_i)

Restores the value of all data items updated by transaction T_i to the old values.

Redo (T_i)

Sets the value of all data items updated by transaction T_i to the new values. This operation is idem potent. It means executing it several times must be equivalent to executing it once.

Transition states:

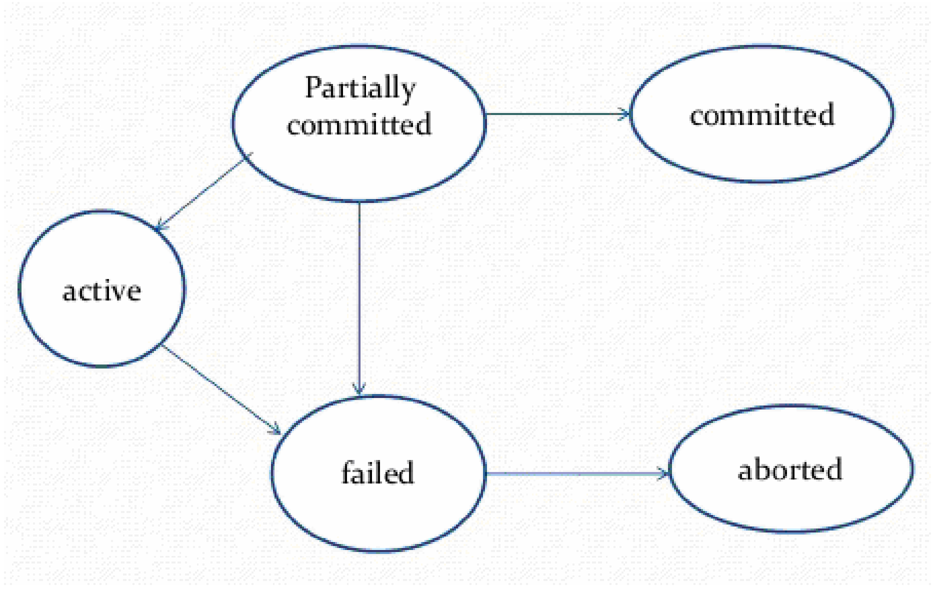


Fig: State transition diagram for transaction execution

- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.

- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.

- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further. Actual output may still be temporarily residing in main memory, and thus hardware failure may stop its successful completion.

- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −

    - Re-start the transaction
    - Kill the transaction

- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

ACID properties of a Transaction

- **A**tomic : Transaction either completes everything or does nothing. Partial execution is never possible.

- **C**onsistent : A database state is consistent if all associated semantic integrity constraints are satisfied. A transaction takes a database from one consistent state to another.
- **I**solated : A transaction runs in isolation. It does not see partial effects of Other transactions and does not allow other transactions to see its partial effects. It is basically a safeguard against mutual interferences between concurrent transactions.

- **D**urable : Once a transaction is committed , its effects are never lost.

- Atomocity and Durability are guaranteed by the recovery manager

- Isolation is guaranteed by the concurrency control manager

- Consistency is ensured by

- DBMS (if it is told to ensure certain constraints
- Application Programmer

Storage Structure

To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.

Storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage.

Volatile storage: Example -- main memory and cache memory. Feature -- Fast

Nonvolatile storage: Example -- secondary storage devices (e.g. magnetic disk, flash storage), tertiary storage devices (e.g. optical media, magnetic tapes) used for archival storage.

Feature -- Slower than volatile storage, particularly for random access; susceptible to failure which may result in loss of information

Stable storage: Information residing in stable storage is *never* lost. To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

Why Concurrency Control ?

If there is no concurrency control, the following problems arise

- The Lost Update Problem
- The Dirty Read Problem
- The Phantom Problem

Lost Update Problem

Consider two transactions (e.g. withdrawals from a joint bank account) running with uncontrolled interleaving.

T1	T2
Read (A)	
	Read (A)
	A = A-1000
A = A –500	
Write (A)	
	Write (A)

Update of T1 is lost

Dirty Read Problem

T1 updates X

T2 reads X

T1 aborts and is rolled back

T2 has read a state that never committed. Hence it should be considered as if it has never existed.

Phantom Problem

Consider a bank with many tellers. Assume each teller maintains its balance and also the overall branch balance. Sum of teller balances should match the branch balance.

Consider an audit transaction T1 made of T11 and T12 that checks this condition.

T11 : sum of balances of tellers of SBI, BESUS
T12 : Balance of SBI, BESUS

T2 : Add a teller to the branch

T3 : Account deposit by this teller

If T2 and T3 are done after T11 but before T12, the audit transaction does not tally balances.

Concurrency Control

There are two good reasons for allowing concurrency:

Improved throughput and resource utilization: Throughput is the number of transactions executed in a given amount of time.

Reduced waiting time: Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time.

Let *T*1 and *T*2 be two transactions that transfer funds from one account to another. Transaction *T*1 transfers Rs.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).
```

Serializability

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

- **Schedule** − A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

- **Serial Schedule** − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

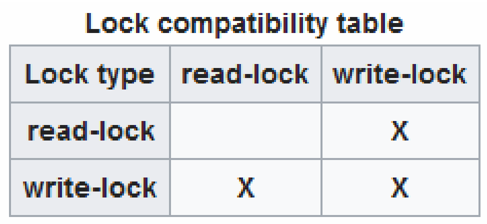
In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state.

```
T2: read(A);
temp := A * 0.1;
A := A − temp;
write(A);
read(B);
B := B + temp;

write(B).
```

To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

- **Write-lock** (**exclusive lock**) : the transaction locks the object or acquires lock for it before *writing* (inserting/modifying/deleting) this object.
- **Read-lock** (**shared lock**) is associated with a database object by a transaction before *reading* (retrieving the state of) this object.



Lock compatibility table

Lock type	read-lock	write-lock
read-lock		X
write-lock	X	X

There are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages. In practice, the most frequently used schemes are two-phase locking.

Lock-Based Protocols

As an illustration, consider again the banking example. Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A*. Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*.

```
T1: lock-X(B);
read(B);
B := B − 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A).
```

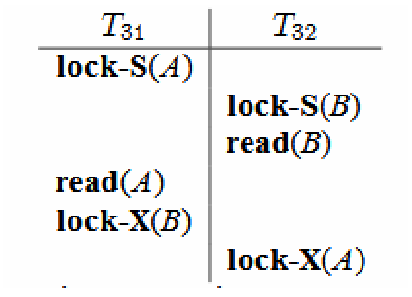
Figure Transaction *T*1.

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure Transaction *T*2.

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions {T_0, T_1, T_2, ...,T_n}. T_0 needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.



T_{31}	T_{32}
lock-S(A)	
	lock-S(B)
	read(B)
read(A)	
lock-X(B)	
	lock-X(A)

Deadlock Situation

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

Growing phase. A transaction may obtain locks, but may not release any lock.

Shrinking phase. A transaction may release locks, but may not obtain any new locks.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure Partial schedule under two-phase locking.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

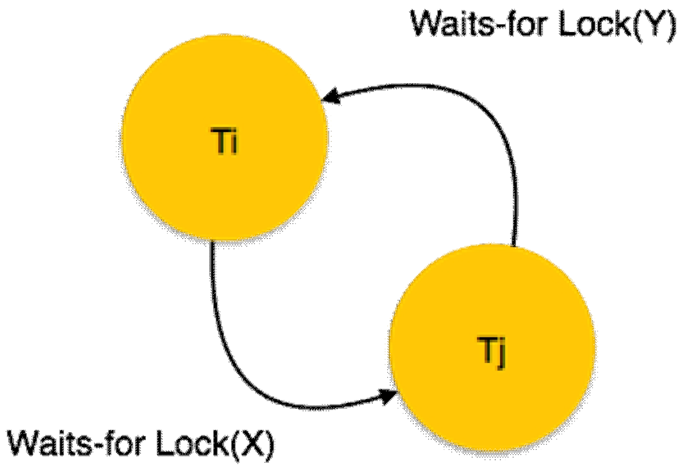
Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction T_i requests for a lock on an item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.

- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.