# **Chapter 1: Transaction Processing Concepts**

# **Introduction**

This chapter discusses the various aspects of transaction processing. It also studies the low-level tasks included in a transaction, the transaction states and properties of a transaction. In the last portion, it will look over schedules and Serializability of schedules.

## **Transaction and System Concepts**

A **transaction** is an event which occurs on the database. Generally, a transaction **reads** a value from the database or **writes** a value to the database. Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A transaction involving only data retrieval without any data update is called read-only transaction. **A read operation** does not change the image of the database in any way. But **a write operation**, whether performed without the intention or intention of inserting, updating or deleting data from the database, changes the image of the database. That is, these transactions bring the database from old image to new image, called the **Before Image** or **BFIM** and **After Image** or **AFIM**.

Database transaction is a collection of SQL queries which forms a logical one task. For a transaction to be completed successfully all SQL queries have to run successfully. It is an atomic process that is either performed into completion entirely or is not performed at all. Database transaction executes either **all or none,** so for example if your database transaction contains 4 SQL queries and one of them fails then change made by other 3 queries should be rolled back. This way your database always remains consistent. The transaction is implemented in the database using **SQL keyword transaction, commit, and rollback**. **Commit** writes the changes made by transaction into database and **rollback** removes temporary changes logged in transaction log by database transaction.

On database transactions, each high-level operation (**read ()** or **write ()**) can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks −

* **read\_item()** − reads data item from storage to main memory. Which includes getting the disk block location too?
* **modify\_item()** − change value of item in the main memory. Manipulate the data and switch the old value with new value on buffer
* **write\_item()** − write the modified value from main memory to storage.

Database access is restricted to read\_item () and write\_item () operations. Likewise, for all transactions, read and write forms the basic database operations.

### **Why transaction is required in database**

Your database records need to exist in a consistent state. After an operation, the database records should move from **one consistent state** to **another consistent state.** That is why we need a transaction. The database is used to store data required by real life application e.g. Banking, Healthcare, Finance etc. All your money stored in banks is stored in the database, all your account is stored in the database and many applications constantly work on these data. In order to protect data and keep it consistent, any changes in this data need to be done in a transaction so that even in the case of failure data remain in the previous state before the start of a transaction. Consider a Classical example of ATM (Automated Tailor Machine); we all use to withdraw and transfer money by using ATM. If you break withdrawal operation into individual steps you will find:

1. Verify account details.
2. Accept withdrawal request
3. Check balance
4. Update balance
5. Dispense money

Suppose your account balance is 1000Birr and you make a withdrawal request of 900Birr. At fourth step, your balance is updated to 900Birr and ATM machine stops working due to power outage. **What will happen?**

Once power comes back and you again tried to withdraw money you surprised by seeing your balance just 100Birr instead of 1000Birr. This is not acceptable by any person in the world. So, we need a transaction to perform such task. If SQL statements would have been executed inside a transaction in database balance would be either 100Birr until money has been dispensed or 1000Birr if money has not been dispensed.

### **Transaction Operations**

The low-level operations performed in a transaction are −

* **Begin transaction** − A marker that specifies start of transaction execution.
* **read\_item or write\_item** − Database operations that may be interleaved with main memory operations as a part of transaction.
* **End transaction** − A marker that specifies end of transaction.
* **Commit** − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
* **Rollback** − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

### **Transaction States**

A transaction may go through a subset of five states:

* Active
* Partially committed
* Committed
* Failed and
* Aborted

**Active** − the initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations. This is the first state of transaction and here the transaction is being executed. For example, updating or inserting or deleting a record is done here. But it is still not saved to the database. Once the transaction starts executing from the first instruction begin\_transaction, the transaction will be considered in active state. During this state, it performs operations READ and WRITE on some data items.

**From active state, a transaction can go into one of two states, a partially committed state or a failed state.**

**Partially Committed** − after the execution of final statement.

The transaction enters this state after the last statement of the transaction has been executed.

* This is also an **execution phase where last step in the transaction is executed**. But data is still not saved to the database.  If you calculate total marks, on final display the total marks step is executed in this state. This is the state of a transaction that successfully executing its last instruction. That means, if an active transaction reaches and executes the COMMIT statement, then the transaction is said to be in partially committed state.

**From partially committed state, a transaction can go into one of two states, a committed state or a failed state.**

**Committed** − After successful completion of transaction.

At partially committed state the database recovery system will perform certain actions to ensure that a failure at this stage should not cause loss of any updates made by the executing transaction. If the current transaction passed this check, then the transaction reaches committed state.

**From committed state, a transaction can go into terminated state.**

**Failed** − If any failure occurs.

The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail. If a transaction cannot proceed to the execution state because of the failure of the system or database, then the transaction is said to be in failed state. In the total mark calculation example, if the database is not able fire a query to fetch the marks, i.e.; very first step of transaction, then the transaction will fail to execute. While a transaction is in the active state or in the partially committed state, the issues like transaction failure, user aborting the transaction, concurrency control issues, or any other failure, would happen. If any of these issues are raised, then the execution of the transaction can no longer proceed. At this stage a transaction will go into a failed state.

**From failed state, a transaction can go into only aborted state.**

**Aborted** − After rolled back to the old consistent state.

This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began. If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. It brings the database to consistent state by aborting or rolling back the transaction. If the transaction fails in the middle of the transaction, all the executed transactions are rolled back to it consistent state before executing the transaction. Once the transaction is aborted it is either restarted to execute again or fully killed by the DBMS. After the failed state, all the changes made by the transaction has to be rolled back and the database has to be restored to its state prior to the start of the transaction. If these actions are completed by the DBMS then the transaction considered to be in aborted state.

**From aborted state, a transaction can go into terminated state.**

A transaction is an atomic operation from the users’ perspective. But it has a collection of operations and it can have a number of states during its execution.

**Therefore, A transaction can end/terminate in three possible ways.**

1. **Successful Termination:** when a transaction completes the execution of all operations in it and reaches the COMMIT command.

**Suicidal(**ራስን ማጥፋት**) Termination:** when the transaction detects an error during its processing and decide to abrupt(በድንገት) itself before the end of the transaction and perform a ROLL BACK

1. **Murderous (**ገዳይ **)Termination:** When the DBMS or the system force the execution to abort for any reason.

Transaction Initiated

Ok to Commit

**Consistent State**

System Initiated

Database Modified

Database unmodified

System Detects Error

Error Detected by Transaction

**Consistent State**

No Error

### **Desirable Properties of Transactions**

Every transaction, for whatever purpose it is being used, has the following four properties: Atomicity, Consistency, Isolation, and Durability. Taking the initial letters of these four properties collectively it is called the **ACID Properties**. Any transaction must maintain the ACID properties.

* **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its own completeness or not performed at all.
* **Consistency protection**: A correct execution of the transaction must take the database from one consistent state to another.
* **Isolation:** A transaction should **not** make its updates visible to other transactions until it is committed.
* **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

**Atomicity** − This property states that a transaction is an atomic unit of processing, that is, either it is performed in **its entirety or not performed at all**. No partial update should exist. This property states that each transaction must be considered as a single unit. No transaction in the database is left **half completed**. Database should be in a state either before the transaction execution or after the transaction execution. It should not be in a state ‘executing’.

In our example above, the transaction should not be left at any one of the step above. All the 5 steps have to be either completed or none of the step has to be completed. If a transaction is failed to execute any step, then it has to rollback all the previous steps and come to the state before the transaction or it should try to complete the failed step and further steps to complete whole transaction.

Say for example, we have two accounts A and B, each containing Birr 1000. We now start a trans 7KI8U7YT54Zaction to deposit Birr 100 from account A to Account B.

Read A;  
A = A – 100;  
Write A;  
Read B;  
B = B + 100;  
Write B;

The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Birr 900 in A and Birr 1100 in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Birr 900 in A, but the same Birr 1000 in B. It would be said that Birr 100 evaporated in the air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that, there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Birr 1000, as if the failed transaction had never occurred. The Atomicity property ensures that.

**Consistency** − A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database. Any transaction should not inject any incorrect or unwanted data into the database. it should maintain the consistency of the database.

In above example, while calculating the balance, it should not perform any other action like inserting or updating or delete. It should also not pick balance of other customers. It should be picking the amount for the A and B customers and adjust their balance. Hence it maintains the consistency of the database.

**Isolation** − A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running. If there are multiple transactions executing simultaneously, then all the transaction should be processed as if they are single transaction. But individual transaction in it should not alter or affect the other transaction. That means each transaction should be executed as if they are independent.

There are several ways to achieve this and the most popular one is using some kind of **locking mechanism**. Locking states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

For example, account A is having a balance of 400Birr and it is transferring 100Birr to account B & C both. So, we have two transactions here. Let’s say these transactions run concurrently and both the transactions read 400Birr balance; in that case the final balance of A would be 300Birr instead of 200Birr. This is wrong. If the transaction were to run in isolation, then the second transaction would have read the correct balance 300Birr (before debiting 100Birr) once the first transaction went successful.

Transactions are concurrency control mechanisms, and they deliver consistency even when being interleaved. Isolation brings us the benefit of hiding uncommitted state changes from the outside world, as failing transactions shouldn’t ever corrupt the state of the system. Isolation is achieved through **concurrency control** using pessimistic or optimistic locking mechanisms.

**Durability** − If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure. The database should be strong enough to handle any system failure. It should not be working for single transaction alone. It should be able to handle multiple transactions too. If there is any set of insert /update, then it should be able to handle and commit to the database. If there is any failure, the database should be able to recover it to the consistent state.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that **Atomicity** and **Durability** is essentially the same thing, just as **Consistency** and **Isolation** is essentially the same thing.

A successful transaction must permanently change the state of a system, and before ending it, the state changes are recorded in a persisted **transaction log**. If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be replayed.

### **Database users Environment**

A DBMS can support many different types of databases. Databases can be classified according to:

* The number of users,
* The database location, and
* The expected type and extent of use.
* The number of users determines whether the database is classified as a
  + - single-user or
    - multiuser.

**SINGLE-USER DBMS**

A single-user can access the database at one point of time. If user A is using the database user B or C must wait until user A is through. These types of systems are optimized for a personal desktop experience, not for multiple users of the system at the same time.

* Properties :-
* All the resources are always available for the user to work.
* The architecture implemented is both One or Two tier(ደረጃ).
* Both the application and physical layer are operated by user.
* For Ex: Standalone Personal Computers, Microsoft Access, etc.

In a single-user environment, the workspace repository resides on the local machine, and can be accessed by the owner of the machine only. Limited facilities exist for sharing work with other users.

**MULTI USER DBMS**

Multi user DBMS are the systems that support two or more simultaneous users. These type of database are familiar in an enterprise database and workgroup database environment. All mainframes and minicomputers are multi-user systems, but most personal computers and workstations are not.

* A multiuser database may exist on a single machine, such as a mainframe or other powerful computer, or it may be distributed and exist on multiple computers.
* Multiuser databases are accessible from multiple computers simultaneously
* Multiuser databases are accessible from multiple computers simultaneously.
* Many people can be working together to update information at the same time.
* All employees have access to the most up-to-date information all of the time.
* Customers have instant access to their personal information held by companies.

In a multiuser environment, the workspace repository resides on a database server, and can be accessed by any user with appropriate database privileges.

COMPARISION BETWEEN SINGLE USER AND MULTIPLE USER DATABASE.

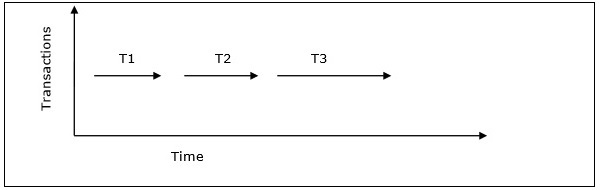
|  |  |
| --- | --- |
| **Single-User** | **Multiuser** |
| * Access Restricted to single user at a time * Database Structure relatively simple * Switching between projects is difficult as single schemas repository is used * Committing change in the database without causing deadlock changes * Infrastructure cost is minimum | * Access can share by Multiple user at a time * Complex Database Structure due to shared access Complexity Increases with the structure of database * Switching between projects is easy as different schemas repositories are used * Access sharing makes it difficult, sometimes causes deadlock * Infrastructure cost is higher such as Servers, Networks etc * Maintenance is also overhead expense Wastage of CPU and resource when Optimum usage/optimization of the user/application remain idle |

## **Schedules and Recoverability**

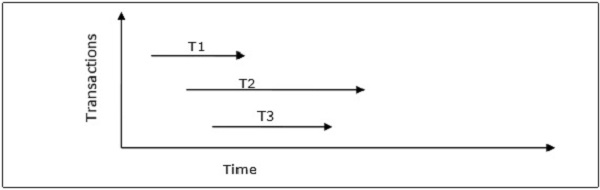
When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction. A schedule is a collection of many transactions which is implemented as a unit.

**Schedule** − A chronological (የጊዜ ቅደም ተከተል) 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. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

* **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 serial schedule, at any point of time, only one transaction is active, due to this, there is no overlapping of transactions. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. This is depicted in the following graph −



**Parallel Schedules(non-serial schedule)** − In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time. This parallel execution brings a **Concurrent transaction; t**he transactions are executed in a preemptive, time shared method.This is depicted in the following graph –



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.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a

a serial schedule\_\_\_\_:

* **longer waiting time** and response time,
* low amount of resource utilization.

In concurrent schedule,

* CPU time is shared among two or more transactions in order to run them concurrently.

However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let’s explain with the help of an example.

Let’s consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

**T1**  
Read A;  
A = A – 100;  
Write A;  
Read B;  
B = B + 100;  
Write B;

**T2**  
Read A;  
Temp = A \* 0.1;  
Read C;  
C = C + Temp;  
Write C;

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

|  |  |
| --- | --- |
| **T1** | **T2** |
| Read A; A = A – 100; Write A; |  |
|  | Read A; Temp = A \* 0.1; Read C; C = C + Temp; Write C; |
| Read B; B = B + 100; Write B; |  |

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Birr 100 from A and writes the new value of Birr 900 into A. T2 reads the value of A, calculates the value of Temp to be Birr 90 and adds the value to C. The remaining part of T1 is executed and Birr 100 is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

|  |  |
| --- | --- |
| **T1** | **T2** |
| Read A; A = A – 100; |  |
|  | Read A; Temp = A \* 0.1; Read C; C = C + Temp; Write C; |
| **Write A;**  Read B; B = B + 100; Write B; |  |

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Birr 1000 each, then the result of this schedule should have left Birr 900 in A, Birr 1100 in B and add Birr 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Birr 900/- has been updated in A. T2 reads the old value of A, which is still Birr 1000, and deposits Birr 100 in C. C makes an unjust gain of Birr 10 out of nowhere.

In the above example, we detected the error simple by examining the schedule and applying common sense. But there must be some well-formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules.

### **Problems Associated with Concurrent Transaction Processing**

Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result which needs control over access. Having a concurrent transaction processing, one can enhance the ***throughput*** of the system. As reading and writing is performed from and on secondary storage, the system will not be idle during these operations, if there is a concurrent processing.

Every transaction should be correct by themselves, but this would not guarantee that the interleaving of these transactions will produce a correct result. The three potential problems caused by concurrency are:

* Lost Update Problem…… **ww conflict**
* Uncommitted Dependency Problem(dirty ready)………… **Rw conflict**
* Inconsistent Analysis Problem……….. **wR conflict**

**Lost Update Problem**

Successfully completed update on a data set by one transaction is overridden by another transaction/user.

E.g. Account with balance A=100.

* + - * T1 reads the account A
      * T1 withdraws 10 from A
      * T1 makes the update in the Database
      * T2 reads the account A
      * T2 adds 100 on A
      * T2 makes the update in the Database
* In the above case, if done one after the other (serially) then we have no problem.
  + If the execution is T1 followed by T2 then A=190
  + If the execution is T2 followed by T1 then A=190
* But if they start at the same time in the following sequence:
  + - * T1 reads the account A=100
      * T1 withdraws 10 making the balance A=90
      * T2 reads the account A=100
      * T2 adds 100 making A=200
      * T1 makes the update in the Database A=90
      * T2 makes the update in the Database A=200
* After the successful completion of the operation in this schedule, the final value of A will be 200 which override the update made by the first transaction that changed the value from 100 to 90.

**Uncommitted Dependency Problem – Temporary update Problem(RW conflict) or dirty read**

* Occurs when one transaction can see intermediate results of another transaction before it is committed.

E.g.

* + T2 increases 100 making it 200 but then aborts the transaction before it is committed. T1 gets 200, subtracts 10 and make it 190. But the actual balance should be 90

**Inconsistent Analysis Problem – incorrect summery**

Occurs when transaction reads several values but second transaction updates some of them during execution and before the completion of the first.

E.g.

* + T2 would like to add the values of A=10, B=20 and C=30. after the values are read by T2 and before its completion, T1 updates the value of B to be 50. at the end of the execution of the two transactions T2 will come up with the sum of 60 while it should be 90 since B is updated to 50.

These concurrent transactions should be in such a way to avoid any interference between them. This demands a new principle in transaction processing, which is Serializability of the schedule of execution of multiple transactions.

### **Schedules and Conflicts**

In a system with a number of simultaneous transactions, a **schedule** is the total order of execution of operations. Given a schedule S comprising of n transactions, say T1, T2, T3………..Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

### **Conflicts in Schedules**

In a schedule comprising of multiple transactions, a **conflict** occurs when two active transactions perform non-compatible operations.

Two operations are said to be in conflict, when all of the following three conditions exists simultaneously −

* The two operations are parts of different transactions.
* Both the operations access the same data item.
* At least one of the operations is a write\_item () operation, i.e. it tries to modify the data item.

## **Serializability**

A **serializable schedule** of ‘n’ transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same ‘n’ transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule. The definition of *serializable schedule* is as follows: A schedule *S* of *n* transactions is **serializable** if it is *equivalent to some serial schedule* of the same *n* transactions.s