**Transaction:**

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 with embedded database accesses in JDBC or ODBC.

A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**.

The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

A transaction is action, or series of actions, carried out by user or application, which accesses or updates contents of database.

It Transforms database from one consistent state to another, although consistency may be violated during transaction.

The concept of transaction provides a mechanism for describing logical units of database processing.

Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions.

Examples of such systems include systems for reservations, banking, stock markets, super markets and other similar systems.

They require high availability and fast response time for hundreds of concurrent users.

Example:

Let’s consider an online banking application:

**Transaction:** When a user performs a money transfer, several operations occur, such as:

* Reading the account balance of the sender.
* Writing the deducted amount from the sender’s account.
* Writing the added amount to the recipient’s account.

In a transaction, all these steps should either complete successfully or, if any error occurs, the database should rollback to its previous state, ensuring no partial data is written to the system.

**Single User Vs. Multi User Systems:**

* A DBMS is a single user if at most one user at a time can use the system.
* A DBMS is a multi user if many users can use the system and hence access the database concurrently.
* Multiple users can access databases and use the computer systems simultaneously because of the concept of Multiprogramming.
* Multiprogramming allows the computer to execute multiple programs or processes at the same time. If only a single central processing unit(CPU) exists, it can actually executes at most one process at a time.
* However multiprogramming operating systems executes some actions from one process then suspend that process and execute some actions of the next process,and so on.
* A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again.
* Hence concurrent execution of process is actually interleaved as illustrated in the following figure, which shows two processes A and B executing concurrently in an interleaved fashion.

Interleaving also prevents the long process from delaying other processes.

A transaction can have one of two outcomes:

**Success -** transaction *commits* and database reaches a new consistent state.

**Failure -** transaction *aborts*, and database must be restored to consistent state before it started. Such a transaction is *rolled back* or *undone*.

Committed transaction cannot be aborted.

Aborted transaction that is rolled back can be restarted later.

**Transactions, Read and Write Operations, and DBMS Buffers**

A **transaction** is an executing program that forms a logical unit of database processing.

A **transaction** includes one or more database access operations such as insertion, deletion, modification, or retrieval operations.

The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

A single application program may contain more than one transaction if it contains several transactions boundaries.

**read-only transaction:**

If the database operations in a transaction do not update the database but

Only retrieve data, the transaction is called a **read-only transaction.**

A database is basically represented as a collection of named data items.

**Granularity:**

The size of a data item is called its granularity, and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block,

**Basic database access operations:**

**1) Read(X)**

A read operation is used to read the value of a particular database element X and stores it in a temporary buffer in the main memory for further actions such as displaying that value.

Example:

For a banking system, when a user checks their balance, a Read operation is performed on their account balance:

SELECT balance FROM accounts WHERE account\_id = 'A123';

This updates the balance of the user's account after withdrawal

**2) Write(X)**

A write operation is used to write the value to the database from the buffer in the main memory. For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it, e.g. some set of arithmetic operations are performed on it according to the user's request, then to store the modified value back in the database, a write operation is performed.

Example:

For the banking system, if a user withdraws money, a Write operation is performed after the balance is updated:

UPDATE accounts SET balance = balance - 100 WHERE account\_id = 'A123';

This updates the balance of the user’s account after withdrawal.

**3) Commit**

This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction.

Example:

After a successful money transfer in a banking system, a Commit operation finalizes the transaction:

COMMIT;

Once the transaction is committed, the changes to the database are permanent, and the transaction is considered successful.

**4) Rollback**

If an error occurs, the Rollback operation undoes all the changes made by the transaction, reverting the database to its last consistent state. In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

Example:

Suppose during the money transfer process, the system encounters an issue, like insufficient funds in the sender’s account. In that case, the transaction is rolled back:

ROLLBACK;

This will undo all the operations performed so far and ensure that the database remains consistent.

A transaction is an atomic unit of work that is either completed entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Hence, the recovery manager keeps track of the following operations:

**BEGIN\_TRANSACTION:** This marks the beginning of transaction execution.

**READ DR WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.

**END\_TRANSACTION:** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by

the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

**COMMIT\_TRANSACTION:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

**ROLLBACK (OR ABORT):** This signals that the transaction has ended *unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

Figure 17.4 shows a state transition diagram that describes how a transaction moves through its execution states.

**Active state**: A transaction goes into an active state immediately after it

, where it can issue READ and WRITE operations.

**Partially committed state:** When the transaction ends, it moves to the **partially** committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently

**Committed state:**

Once check in partially committed state is successful, the transaction is said to have reached its commit point and enters the committed state.

Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

**Failed state:**

A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state.

The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

**Terminated state:**

The terminated state corresponds to the transaction leaving the system.

The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.

**ACID Properties or DESIRABLE PROPERTIES OF TRANSACTIONS**

In DBMS **ACID** (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically

**Atomicity:**

Atomicity refers to the ability of the DBMS to guarantee that either all of the operations of a transaction are performed or none of them are. Database modifications must follow an all or nothing rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.

If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

**Consistency:**

The consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not).

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database.

A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction,

**Isolation:**

The isolation portion of the ACID Properties is needed when there are concurrent transactions. Concurrent transactions are transactions that occur at the same time, such as shared multiple users accessing shared objects.

Although multiple transactions may execute concurrently, each transaction must be independent of other concurrently executing transactions. A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

**Durability:**

Maintaining updates of committed transactions is critical. These updates must never be lost. The ACID property of durability addresses this need. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails. Features to consider for durability:

* recovery to the most recent successful commit after a database software failure
* recovery to the most recent successful commit after an application software failure
* recovery to the most recent successful commit after a CPU failure
* recovery to the most recent successful backup after a disk failure
* recovery to the most recent successful commit after a data disk failure

**The System Log**

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items.

This information may be needed to permit recovery from failures.

The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.

In addition,the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

We now list the types of entries-called log records-that are written to the log and the action each performs.

In these entries, T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:

**Transaction Schedules**

When multiple transaction requests are made at the same time, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions.

There are broadly two types of transaction schedules discussed as follows:

**i) Serial Schedule**

In this kind of schedule, when multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed. This ensures consistency in the database as transactions do not execute simultaneously.

But, it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time. To improve the throughput of the system, another kind of schedule are used which has some more strict rules which help the database to remain consistent even when transactions execute simultaneously.

Example:

In a serial schedule, the first transaction completes before the second transaction starts:

* Transaction 1: Read balance → Update balance → Commit
* Transaction 2: Read balance → Update balance → Commit

**ii) Non-Serial Schedule**

To reduce the waiting time of transactions in the waiting queue and improve the system efficiency, we use non-serial schedules which allow multiple transactions to start before a transaction is completely executed. This may sometimes result in inconsistency and errors in database operation.

So, these errors are handled with specific algorithms to maintain the consistency of the database and improve CPU throughput as well. Non-serial schedules are also sometimes referred to as parallel schedules, as transactions execute in parallel in these kinds of schedules.

Example:

Transaction 1 and Transaction 2 can be executed in parallel:

* Transaction 1: Read balance → Update balance
* Transaction 2: Read balance → Update balance

Without proper isolation mechanisms, this may cause inconsistencies.

**Serializable**

Serializability in DBMS is the property of a non-serial schedule that determines whether it would maintain the database consistency or not. The non-serial schedule which ensures that the database would be consistent after the transactions are executed in the order determined by that schedule is said to be Serializable Schedules.

The serial schedules always maintain database consistency as a transaction starts only when the execution of the other transaction has been completed under it. Thus, serial schedules are always serializable. A transaction is a series of operations, so various states occur in its completion journey.