A **transaction** is a *unit* of program execution that accesses and possibly updates various data items. Two main issues to deal with: Failures of various kinds, such as hardware failures and system crashes ,Concurrent execution of multiple transactions .**ACID Properties** : A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure: **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures. Transaction State: **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 restored to its state prior to the start of the transaction. Two options after it has been aborted: restart the transaction can be done only if no internal logical error ,kill the transaction .**Committed** – after successful completion. **Concurrent Executions :** Multiple transactions are allowed to run concurrently in the system. Advantages are: **increased processor and disk utilization**, leading to better transaction *throughput* , **reduced average response time** for transactions: short transactions need not wait behind long ones. **Concurrency control schemes** – mechanisms to achieve isolation that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database **Serializability** : Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of: **conflict serializability** ,**view serializability.Conflict Serializability :** If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.**View Serializability :** Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q, If in schedule S, transaction T_i reads the

initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j . The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'. As can be seen, view equivalence is also based purely on **reads** and **writes** alone.**Lock Based Protocol:** A lock is a mechanism to control concurrent access to a data item.Data items can be locked in two modes : *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix : A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted. A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules. **The Two-Phase Locking Protocol** :This is a protocol which ensures conflict-serializable schedules. Phase 1: Growing Phase :transaction may obtain locks ,transaction may not release locks .Phase 2: Shrinking Phase :transaction may release locks ,transaction may not obtain locks . The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock). Two-phase locking *does not* ensure freedom from deadlocks . Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts. **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit. There can be conflict serializable schedules that cannot be obtained if two-phase locking is used. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense: Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Deadlock Handling: System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set. ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies : Require that each transaction locks all its data items before it begins execution (predeclaration). Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol). **More Deadlock Prevention Strategies** : Following schemes use transaction timestamps for the sake of deadlock prevention alone. **wait-die** scheme — non-preemptive : older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. a transaction may die several times before acquiring needed data item **wound-wait** scheme — preemptive : older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. may be fewer rollbacks than *wait-die* scheme. Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided. Timeout-Based Schemes: a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. thus deadlocks are not possible . simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.**Failure Classification:** **Transaction failure** : **Logical errors**: transaction cannot complete due to some internal error condition. **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock). **System crash**: a power failure or other hardware or software failure causes the system to crash. **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash . Database systems have numerous integrity checks to prevent corruption of disk data . **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage. Destruction is assumed to be detectable: disk drives use checksums to detect failures. **Recovery and Atomicity:** To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself. We study **log-based recovery mechanisms** in detail We first present key concepts And then present the actual recovery algorithm .Less used alternative: **shadow-paging.** **Log Based Recovery:** A **log** is kept on stable storage. The log is a sequence of **log records**, and maintains a record of update activities on the database. When transaction T_i starts, it registers itself by writing a $<T_i$ **start**>log record .*Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before

the write (the **old value**), and V_2 is the value to be written to X (the **new value**). When T_i finishes it last statement, the log record <T_i **commi**t> is written. Two approaches using logs :Deferred database modification ,Immediate database modification . The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits .Update log record must be written *before* database item is written We assume that the log record is output directly to stable storage . Output of updated blocks to stable storage can take place at any time before or after transaction commit . Order in which blocks are output can be different from the order in which they are written. The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit :Simplifies some aspects of recovery ,But has overhead of storing local copy