UNIT-V: TRANSACTION, INDEXING

1) What is Transaction? What are the properties of a Transaction? 5M

A **transaction** is a logical unit of work that transforms the database from one consistent state to another. To ensure correctness, every transaction in DBMS must satisfy the **ACID properties**.

Property	Meaning	Description / Example
A – Atomicity	All or Nothing	Ensures that either all operations of a transaction are completed or none are. ➡ Example: Money is deducted from one account and added to another – both must happen or neither.
C – Consistency	Preserves Validity	Ensures the database remains in a valid state before and after the transaction. ➡ Example: Total bank balance remins the same after transfer.
I – Isolation	Independent Execution	Each transaction executes as if it is the only one running, without interference. ➡ Example: Two users transferring money simultaneously don't affect each other.
D – Durability	Permanent Effect	Once a transaction is committed, its changes are permanent even if the system fails.

Summary:

The **ACID properties** — **Atomicity, Consistency, Isolation, and Durability** — ensure that transactions execute **reliably, safely, and correctly**, maintaining the integrity of the database.

1) Explain in detail about Serializability & Recoverability 5 M

Ans:

1. Serializability

- **Definition:**

Serializability is a **concept of concurrency control** in DBMS that ensures that the **concurrent execution of transactions** is **correct and consistent** — i.e., it gives the **same result** as if the transactions were executed **serially (one after another)**.

It is the **standard** for correctness in concurrent schedules.

- **Why Serializability is Needed:**

When multiple transactions execute at the same time, problems like:

- **Lost Update**

- **Dirty Read**
- **Inconsistent Read**

can occur.
Serializability ensures that even if transactions execute concurrently, the **final database state remains consistent**, just as it would be in some **serial order**.

🔹 **Example:**

Transaction T1	**Transaction T2**
READ(A) = 100	
A = A - 10	
	READ(A) = 100
	A = A + 20
WRITE(A) = 110	
	WRITE(A) = 120

If both execute together **without control**, final A = 120.
But if executed serially:

- **T1→T2** → Final A = 110 + 20 = 130
- **T2→T1** → Final A = 100 + 20 - 10 = 110

Since concurrent result (120) ≠ serial results (130 or 110), **schedule is NOT serializable**.

🔹 **Types of Serializability:**

1. **Conflict Serializability**
    - A schedule is conflict serializable if it can be transformed into a serial schedule by **swapping non-conflicting operations**.
    - Two operations conflict if:
        - They belong to different transactions,
        - They operate on the same data item,
        - At least one of them is a **write**.

✅ Example of **Conflict Serializable Schedule:**

T1: R(A) W(A)

T2: R(A)

This can be converted into serial order (T1→T2).

- **Tool:** We can check using **Precedence Graph (Serialization Graph)**
    - If graph has **no cycles → Conflict Serializable**.

---

2. **View Serializability**
    - Two schedules are view equivalent if:
        - They read initial values the same way,
        - They read the same data written by same transaction,
        - Final writes are done by the same transaction.
    - View serializability is **broader** than conflict serializability.

Summary:

Type	Definition	Check Method
Conflict Serializability	Can be transformed into serial schedule by swapping non-conflicting operations	Precedence Graph (no cycle)
View Serializability	Gives same view of reads/writes as some serial schedule	Check read/write relationships

2. Recoverability in DBMS

- **Definition:**

A schedule is **Recoverable** if, whenever a transaction **Tj** reads data written by another transaction **Ti**, then **Tj commits only after Ti commits**.

This ensures that if Ti fails, the effect of Ti does not persist in Tj.

- **Example:**

Time	Operation
T1: W(X)	
T2: R(X)	(reads value written by T1)
T1: Commit	
T2: Commit	

✅ This schedule is **Recoverable**, because **T2 commits after T1**.

- **Non-Recoverable Schedule Example:**

Time	Operation
T1: W(X)	
T2: R(X)	(reads value written by T1)
T2: Commit ❌	(committed before T1)
T1: Rollback	

If **T1 fails**, T2 has already used its dirty value → leads to **inconsistency**.
Hence, **Non-recoverable schedule**.

- **Types of Recoverability:**
  1. **Recoverable Schedule:**
     - Tj commits **after** Ti (if it has read from Ti).
  2. **Cascadeless Schedule:**
     - Transactions **read only committed data**.
     - Prevents **Cascading Rollbacks**.

✅ Example:

T1: W(X), Commit

T2: R(X)

→ Safe (Cascadeless).

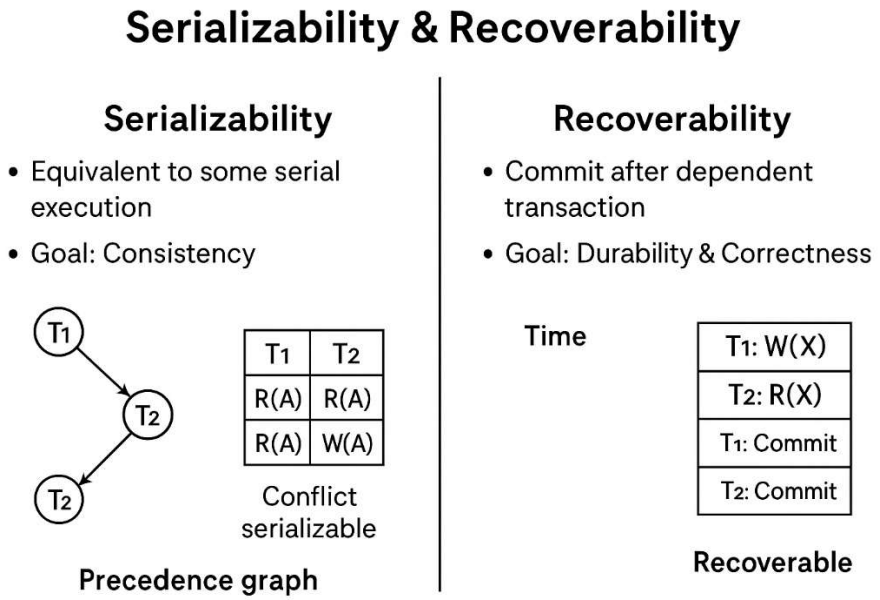
  3. **Strict Schedule:**
     - Transactions can **neither read nor write** a data item **until the last transaction that wrote it has committed**.
     - **Most restrictive but safest**.

- **Comparison Table:**

Type	Description	Problem Prevented
Recoverable	Commit after dependent transaction	Dirty Commit
Cascadeless	Read only committed data	Cascading Rollback
Strict	Neither read nor write until commit	Cascading Rollback + Dirty Read

Summary Notes:

Concept	Meaning	Goal
Serializability	Equivalent to some serial execution	**Consistency**
Recoverability	Ensures commits are safe after dependent transactions	**Durability & Correctness**



Summary:

Serializability ensures the correctness of concurrent transactions by making sure the result of concurrent execution is equivalent to some serial order of transactions.

It is mainly of two types: Conflict serializability (checked using precedence graph) and View serializability.

Recoverability ensures that a transaction commits only after all the transactions from which it has read have committed. Depending on the level of safety, schedules may be recoverable, cascadeless, or strict.

Recoverable schedules prevent data inconsistency and ensure reliable database recovery.

2) Explain ARIES Recovery Algorithm 5 M

Ans:

ARIES Recovery Algorithm

ARIES stands for: **Algorithm for Recovery and Isolation Exploiting Semantics**

The ARIES algorithm (Algorithm for Recovery and Isolation Exploiting Semantics) is a Write-Ahead Logging-based recovery method used in DBMS. It performs recovery in three phases: **Analysis**, **Redo**, and **Undo**.

The analysis phase identifies active transactions and dirty pages;

the redo phase reapplies all updates to repeat history;

and the undo phase rolls back all uncommitted transactions using compensation log records (CLRs).

ARIES ensures that all committed transactions are preserved and database consistency is restored after a crash.

Main Idea

ARIES ensures that **no committed transaction is lost**, and **no uncommitted transaction is applied** after recovery.

It follows the **Write-Ahead Logging (WAL)** rule:

Changes are written to the **log file before** they are written to the database.

🔹 Design Principles of ARIES

1. **Write-Ahead Logging (WAL):**
    - Log records are written to stable storage **before** the actual data pages are written to disk.
2. **Repeating History During Redo:**
    - During recovery, ARIES **reapplies all actions** (including uncommitted ones) from the log to bring the database to the exact state before the crash.
3. **Logging Changes During Undo:**
    - When undoing uncommitted transactions, ARIES logs **compensation log records (CLRs)** so that undo actions are also recoverable.

🔹 Phases of ARIES

ARIES recovery happens in **three major phases**:

🧩 1. Analysis Phase

Goal: Identify which transactions were **active** at the time of crash, and which pages **need redo**.

Steps:

- Scan the log **forward** from the last checkpoint.
- Determine:
    - **Active Transactions Table (ATT)** — transactions that were not committed or aborted.
    - **Dirty Page Table (DPT)** — pages that were modified but not yet written to disk.

Result:
We know which transactions were incomplete and which pages may have unflushed updates.

🧩 2. Redo Phase

Goal: Repeat history — reapply all updates to bring the database to the state at the time of crash.

Steps:

- Start from the **earliest log record** affecting dirty pages.

- Redo every action (even for uncommitted transactions) to ensure database reflects all changes before crash.

Condition for Redo:
A log record with LSN (Log Sequence Number) is redone **only if** the page on disk has an older pageLSN.

🧩 3. Undo Phase

Goal: Undo all uncommitted transactions.

Steps:

- Use the **ATT** (from analysis phase).
- Work backward through the log, undoing updates of uncommitted transactions.
- For each undo, generate a **Compensation Log Record (CLR)** to record the undo action.

CLR helps if the system crashes again during recovery — we can restart recovery safely.

🔹 Example Flow

Phase	Action
1. Analysis	Identify active transactions (say T1, T2) and dirty pages
2. Redo	Reapply all updates up to the crash point
3. Undo	Rollback all uncommitted transactions (e.g., T2)

🔹 Log Records in ARIES

Each log record has:

Field	Description
LSN	Log Sequence Number
PrevLSN	Pointer to previous log entry for same transaction
TransID	Transaction ID
PageID	Affected page
Type	Update / Commit / Abort / CLR etc.

🔹 Advantages of ARIES

✅ Supports **fine-grained (page-level) recovery**
✅ Handles **system crashes** efficiently
✅ Uses **repeating history** for fast restart
✅ Ensures **no data loss for committed transactions**
✅ Allows **partial rollbacks**

🔹 **Disadvantages**

❌ Implementation complexity is high

❌ Requires detailed log management and storage overhead

Summary

Phase	Purpose	Direction
Analysis	Find active transactions & dirty pages	Forward
Redo	Reapply all changes	Forward
Undo	Rollback uncommitted changes	Backward

3) Explain the need of commit, rollback and save point operations in transaction management. 5 M

Ans:

In transaction management, COMMIT, ROLLBACK, and SAVEPOINT are control statements used to maintain the consistency and atomicity of the database.

COMMIT makes all changes permanent,

ROLLBACK cancels all uncommitted changes, and

SAVEPOINT allows partial rollbacks within a transaction.

These operations ensure that the database remains in a consistent state even if errors or failures occur.

1. COMMIT

Definition:
Used to make all changes of a transaction permanent in the database.

Need / Purpose:

- Indicates successful completion of a transaction.
- Once committed, the changes cannot be undone.
- Ensures durability in the ACID properties.

Example:

```
UPDATE Account SET balance = balance - 500 WHERE acc_no = 101;
UPDATE Account SET balance = balance + 500 WHERE acc_no = 202;
COMMIT;
```

✅ Changes are permanently saved.

2. ROLLBACK

Definition:
Used to undo all the changes made by the current transaction.

Need / Purpose:

- Used when an error or failure occurs.
- Restores the database to the last consistent state before the transaction began.
- Ensures atomicity (either all or none of the transaction executes).

Example:

```
UPDATE Account SET balance = balance - 500 WHERE acc_no = 101;
ROLLBACK;
```

❌ Cancels the update — no changes saved.

3. SAVEPOINT

Definition:
A checkpoint set within a transaction to which a rollback can occur partially.

Need / Purpose:

- Allows partial rollback instead of undoing the entire transaction.
- Increases flexibility in transaction control.
- Useful in large transactions involving multiple steps.

Example:

```
SAVEPOINT A;
UPDATE Account SET balance = balance - 500 WHERE acc_no = 101;
SAVEPOINT B;
UPDATE Account SET balance = balance + 500 WHERE acc_no = 202;
ROLLBACK TO B;
COMMIT;
```

✅ Only part of the transaction is undone up to SAVEPOINT B.

Summary

Operation	Meaning	Purpose
COMMIT	Save changes permanently	Confirms successful transaction
ROLLBACK	Undo all changes	Restore consistency after failure
SAVEPOINT	Mark intermediate points	Allows partial rollback

4) Write about Operations on B+ tree with examples 5 M

Ans:

Operations on B+ Tree

A B+ tree is a balanced indexing structure that supports efficient searching, insertion, deletion, and range queries.

Insertion may cause node splitting and key promotion, while deletion may cause merging or borrowing.

All leaf nodes are linked, enabling fast sequential access. These properties make B+ trees ideal for database indexing and file systems.

A **B+ Tree** is a **balanced multi-level index structure** used in DBMS for efficient searching, insertion, and deletion.
It maintains **sorted data** and allows **range queries** efficiently.

🔹 **Structure of B+ Tree**

- All **leaf nodes** are at the same level.
- **Internal nodes** store only **keys** (for navigation).
- **Leaf nodes** store **actual data or pointers to data**.
- Leaf nodes are **linked** sequentially for range access.

🔸 **1. Search Operation**

Goal: To find a key value efficiently.

Steps:

1. Start from the **root node**.
2. Compare key K with keys in the node.
3. Follow the appropriate child pointer based on range.
4. Continue until reaching the **leaf node**.
5. If found, return the record pointer.

Example:
Let's search for key **25** in the B+ tree below:

```
        [20 | 40]
       /    |    \
   [10]   [20,30]   [40,50]
```

→ Search(25):
20 < 25 < 40 → go to middle child → found in leaf [20,30].

✅ **Result:** Key 25 found in leaf level.

🔸 2. Insertion Operation

Goal: To add a new key while keeping the tree balanced.

Steps:

1. Find the **correct leaf node** using search.
2. Insert the key in **sorted order**.
3. If the node **overflows** (more keys than allowed):
    - **Split** the node into two halves.
    - **Promote** the middle key to the parent node.
4. If parent also overflows, **split recursively** up to root.

Example:
Insert key **60** into B+ tree:

Before:

```
      [30]
     /   \
 [10,20]  [40,50]
```

→ Insert 60 into right node → [40,50,60] → Overflow → Split:

After Insertion:

```
      [30,50]
     /   |   \
[10,20] [40] [50,60]
```

✅ Tree remains balanced.

🔸 3. Deletion Operation

Goal: Remove a key and maintain balance.

Steps:

1. Locate the **leaf node** containing the key.
2. Delete the key.

3. If node **underflows** (less than minimum keys):
    - **Borrow** a key from a sibling, or
    - **Merge** with a sibling.
4. Update parent nodes if required.

Example:
Delete key **20** from:

[30,50]

/ | \

[10,20] [40] [50,60]

After deletion of 20 → [10] has too few keys.
Merge [10] and [40] → Update parent.

[50]

/ \

[10,30,40] [50,60]

✅ Tree remains balanced after deletion.

🔸 **4. Traversal (Range Queries)**

Because **leaf nodes are linked**, we can easily perform **range queries**.

Example:
To find all keys between 30 and 60 → traverse leaf nodes sequentially:
→ [30,40,50,60] ✅

Summary

Operation	Description	Action Taken on Overflow/Underflow
Search	Finds a key from root to leaf	Simple traversal
Insert	Adds a new key	Split node and promote middle key
Delete	Removes a key	Borrow or merge nodes
Traversal	Sequentially visit leaves	Used for range queries

5) Explain Concurrent Execution in detail 5 M

Ans:

🔹 **Definition:**

Concurrent execution means that **two or more transactions execute simultaneously** (overlapping in time) in a multi-user database system.

DBMS allows concurrent execution to improve **performance**, **resource utilization**, and **system throughput**, while ensuring **data consistency** through concurrency control.

However, it may cause problems such as lost updates, dirty reads, or inconsistent data.

To prevent these issues, concurrency control techniques like locking, timestamp ordering, and validation-based protocols are used.

The goal is to achieve serializability and maintain database consistency while enabling efficient transaction processing.

- **Need for Concurrent Execution**

In real-world databases, many users access data at the same time.
Without concurrency, transactions would execute **one after another (serially)**, leading to:

- Slow response times
- Poor CPU & disk utilization
- Low system throughput

Hence, DBMS allows **overlapping transactions**, while maintaining correctness.

- **Example:**

Consider two transactions:

- **T1:** Transfer ₹100 from A to B
- **T2:** Add 10% interest to A

Without concurrency:
T1 completes fully before T2 starts.
With concurrency:
T1 and T2 execute **interleaved**:

T1: Read(A)

T2: Read(A)

T1: A = A - 100

T2: A = A * 1.1

T1: Write(A)

T2: Write(A)

If not controlled, this may cause **data inconsistency** (lost updates).

🔹 **Problems in Concurrent Execution**

1. **Lost Update Problem:**
    - Two transactions overwrite each other's updates.
    - ❌ Example: Both T1 and T2 modify the same record, last write wins.
2. **Temporary (Dirty) Read:**
    - A transaction reads data written by another **uncommitted** transaction.
    - If the other transaction rolls back, the first one has **invalid data**.
3. **Inconsistent Retrievals:**
    - A transaction reads a set of records that are being modified by another transaction.
4. **Unrepeatable Read:**
    - A transaction reads the same record twice and gets **different values** because another transaction updated it in between.

🔹 **Concurrency Control Techniques**

DBMS uses **Concurrency Control** to ensure correctness of concurrent execution.

Main techniques:

1. **Lock-based Protocols** (e.g., Two-phase locking)
2. **Timestamp Ordering Protocols**
3. **Validation-based (Optimistic) Protocols**
4. **Multiversion Concurrency Control (MVCC)**

These ensure **Serializability**, i.e., the concurrent execution produces the same result as some **serial order**.

🔹 **Advantages of Concurrent Execution**

✅ Better CPU and I/O utilization
✅ Increased system throughput
✅ Reduced waiting time for users
✅ Faster transaction response

🔹 **Disadvantages**

❌ Risk of data inconsistency
❌ Requires complex control mechanisms
❌ Possible deadlocks or rollbacks

Summary

Aspect	Description
Meaning	Simultaneous execution of multiple transactions
Goal	Improve performance and resource utilization
Main Problem	Data inconsistency
Solution	Concurrency control (Locks, Timestamps, MVCC)
Outcome	Serializable and consistent results

6) Discuss briefly about primary and secondary indexes? Explain their role in indexing the tables with suitable example. 5 M

Ans:

Indexing is a technique used to speed up the retrieval of records from a database table. An index is like a book's index — it helps locate data quickly without scanning the entire table.

1. Primary Index:

- A **primary index** is created on a field that has **unique values** (usually the *primary key*).
- The data file is **ordered** (sorted) based on this key field.
- Each index entry contains the **key value** and a **pointer** to the corresponding data block.

Characteristics:

- Created automatically when a primary key is defined.
- It is **dense** if every record has an index entry.
- It is **sparse** if only one entry per data block is indexed (commonly used for large sorted files).

Example:

Consider a STUDENT table:

Roll_No	Name	Branch	Marks
101	Ravi	CSE	85
102	Priya	ECE	78
103	Arjun	CSE	90

- The table is sorted on **Roll_No** (Primary Key).
- The primary index stores:

- Index:
- Roll_No → Block_Address
- 101 → B1
- 102 → B2
- 103 → B3

This allows direct access to a student record using Roll_No without full table scan.

---

2. Secondary Index:

- A **secondary index** is created on fields that are **not primary keys** or **not sorted** in the data file.
- It provides a way to access records based on **non-ordering attributes**.
- Can have **duplicate values**.

Characteristics:

- Created manually for faster search on non-key columns.
- Always **dense** (every record has an entry).

Example:

If we create an index on the Branch column:

Index (Branch → List of Roll_Nos)

CSE → {101, 103}

ECE → {102}

Now, searching for all students in the **CSE branch** becomes faster using the secondary index.

Role in Indexing Tables:

- **Primary Index** → Improves access based on primary key; efficient for range and equality queries.
- **Secondary Index** → Improves performance of queries on non-key attributes.

Summary Table:

Feature	Primary Index	Secondary Index
Based on	Primary Key / Sorted Field	Non-key / Unsorted Field
Uniqueness	Unique	May have duplicates
File organization	Ordered	Unordered
Type of index	Usually Sparse	Always Dense

Feature	Primary Index	Secondary Index
Example Field	Roll_No	Branch

In short:
Primary index provides fast access using the primary key, while secondary index enhances retrieval using non-key attributes, thus improving query performance.

7) How to use hash-based indexing? Explain briefly and differentiate it from other indexing mechanisms.
7 M

Ans:

1. Concept:

Hash-based indexing is a method of locating records in a database using a **hash function**.
A **hash function** converts a **search key value** into a **hash value**, which directly points to the location (bucket/block) where the record is stored.

---

2. How It Works:

1. A **hash function (h)** is applied to the search key: $h(key) = bucket\_number$
2. The resulting **bucket number** determines where the record is stored or should be searched.
3. Each **bucket** may contain multiple records that hash to the same value (called **collisions**).

Example:

Consider a table STUDENT(Roll_No, Name, Branch) and we create a hash index on Roll_No.

Let the hash function be:

$$h(Roll\_No) = Roll\_No \mod 5$$

Roll_No	h(Roll_No)	Bucket
101	1	B1
102	2	B2
106	1	B1
110	0	B0

When we search for **Roll_No = 106**,
→ Apply hash function: h(106) = 1
→ Go directly to **Bucket B1** to find the record.

Thus, data retrieval is **very fast (O(1))** for equality searches.

3. Collision Handling:

When multiple records hash to the same bucket:

- **Chaining:** Store all colliding records in a linked list at that bucket.
- **Open addressing:** Find another empty bucket using a probe sequence.

---

4. Comparison with Other Indexing Mechanisms:

Feature	Hash-Based Indexing	B+ Tree Indexing (Ordered Index)
Data Organization	Based on hash function	Based on sorted order of keys
Search Type	Best for *equality* searches (=)	Supports both *equality* and *range* searches (<, >)
Access Time	Constant time O(1) on average	Logarithmic time O(log n)
Order Preservation	Not preserved	Preserved
Range Queries	Not possible	Efficiently supported
Structure	Buckets with hash function	Hierarchical (tree) structure

5. Advantages:

- Extremely fast for **exact match queries**.
- Simple and efficient implementation.

6. Limitations:

- Not suitable for **range queries** (e.g., Roll_No > 100).
- Performance drops if many **collisions** occur.
- Rehashing needed when table grows large.

Summary:
Hash-based indexing uses a hash function to map search keys to data locations, making exact-match lookups very fast. Unlike **B+ tree indexing**, it cannot handle range or sorted queries efficiently.

8) Discuss Transaction States with a diagram 3 M

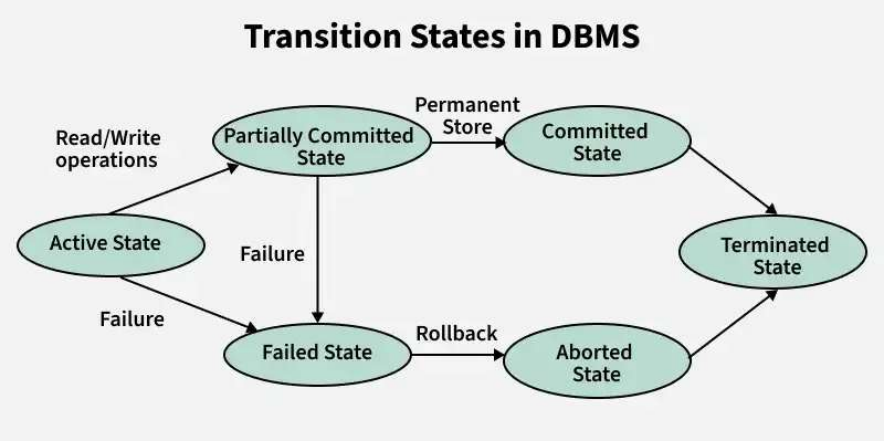
Ans:

A **transaction** in DBMS is a sequence of database operations that performs a single logical unit of work. During its execution, a transaction passes through several **states** that represent its progress.

---

1. Transaction States:

State	Description
1. Active	The transaction is currently being executed. Operations like read/write are performed in this state.
2. Partially Committed	All operations are done, and the transaction is ready to be committed, but the changes are not yet made permanent in the database.
3. Committed	The transaction has successfully completed, and all changes are permanently saved in the database.
4. Failed	If any error, system crash, or constraint violation occurs, the transaction moves to the failed state.
5. Aborted	All the changes made by the transaction are rolled back (undone), and the database is restored to its previous consistent state. The transaction may be restarted or terminated.



9) Explain the concept of deadlock and dead lock prevention policies with an example 5 M

Ans:

1. Concept of Deadlock:

A **deadlock** occurs in a database system when **two or more transactions** are **waiting for each other indefinitely** to release locks on resources.

In other words, **each transaction holds a resource** that the **other transaction needs**, causing both to wait forever.

2. Example of Deadlock:

Consider two transactions **T1** and **T2**:

Time	Transaction T1	Transaction T2
1	Locks **Resource A**	
2		Locks **Resource B**

Time	Transaction T1	Transaction T2
3	Wants **Resource B** (held by T2)	Wants **Resource A** (held by T1)

Now:

- **T1** waits for **B** (locked by **T2**)
- **T2** waits for **A** (locked by **T1**)

🔁 Both transactions are waiting for each other → **Deadlock occurs**.

---

3. Conditions for Deadlock:

Deadlock can occur if all four of these conditions hold simultaneously:

1. **Mutual Exclusion** – A resource can be used by only one transaction at a time.
2. **Hold and Wait** – A transaction holding one resource is waiting for another.
3. **No Preemption** – A resource cannot be forcibly taken away.
4. **Circular Wait** – A cycle of transactions exists, each waiting for the next.

---

4. Deadlock Prevention Policies:

Deadlock prevention aims to **avoid at least one of the above four conditions**.
Common prevention techniques include:

(a) Resource Ordering

- Assign a **unique order (number)** to each resource.
- Transactions must request resources in **increasing order** only.
- This eliminates **circular wait**.

(b) Wait–Die Scheme (Non-Preemptive)

- Based on **timestamps**:
    - If **older** transaction requests a resource held by a **younger** one → it **waits**.
    - If **younger** transaction requests a resource held by an **older** one → it **dies (aborts)** and restarts later.

(c) Wound–Wait Scheme (Preemptive)

- Also uses **timestamps**:
    - If **older** transaction requests a resource held by a **younger** one → **younger is rolled back** (preempted).
    - If **younger** transaction requests a resource held by an **older** one → it **waits**.

These prevent **circular waiting** and ensure progress.

(d) No Wait / Timeout

- A transaction requesting a locked resource is **immediately rolled back** if it cannot get the lock (No Wait).
- Or, it **waits only for a fixed time** before being rolled back (Timeout).

---

5. Summary Table:

Method	Approach	Drawback
Resource Ordering	Prevent circular wait	Difficult to maintain for many resources
Wait–Die	Non-preemptive, uses timestamp	Many rollbacks possible
Wound–Wait	Preemptive, uses timestamp	Frequent rollbacks of younger transactions
Timeout	Simple and practical	May cause unnecessary rollbacks

✅ **In Short:**
A **deadlock** is a situation where transactions wait forever for each other's resources.
Prevention policies like *resource ordering*, *wait–die*, and *wound–wait* avoid deadlocks by controlling how transactions acquire locks

10) Define locking protocol. Describe the Strict Two-Phase Locking protocol with an example. 5 M

Ans:

1. Definition:

A **locking protocol** is a set of rules that **governs how and when transactions acquire and release locks** on data items in order to ensure **concurrency control** and **maintain database consistency**.

Locks prevent multiple transactions from accessing the same data item simultaneously in a conflicting way.

---

2. Types of Locks:

- **Shared Lock (S-lock):** Allows a transaction to **read** a data item.
- **Exclusive Lock (X-lock):** Allows a transaction to **read and write** a data item.

---

3. Strict Two-Phase Locking (Strict 2PL) Protocol:

This is the most widely used **locking protocol** that guarantees **conflict serializability** and also **recoverability**.

Rules of Strict 2PL:

1. A transaction must obtain a **lock** on a data item **before** accessing it (read/write).

2. **All locks are released only after the transaction commits or aborts.**
3. The execution has two phases:
    - **Growing Phase:** Transaction acquires locks (no unlocks allowed).
    - **Shrinking Phase:** Begins after the first unlock — but in Strict 2PL, all unlocks happen **after commit/abort**, so effectively, no shrinking occurs during execution.

4. Example:

Consider two transactions **T1** and **T2**:

Step	Transaction T1	Transaction T2
1	Lock-X(A)	
2	Write(A)	
3		Lock-X(A) (waits)
4	Commit → Unlock(A)	
5		Now Lock-X(A), Write(A), Commit

Explanation:

- T1 locks A, writes it, and commits — releasing the lock only after completion.
- T2 waits until T1 commits.
- This prevents **dirty reads** or **uncommitted data access**, ensuring **strict serializability**.

5. Advantages:

- Guarantees **conflict-serializable schedules**.
- Prevents **cascading rollbacks** because data written by uncommitted transactions is never read by others.

6. Disadvantage:

- Can lead to **reduced concurrency** due to long lock holding.

In short:
A **locking protocol** ensures safe concurrent access to data.
The **Strict Two-Phase Locking protocol** holds all locks until commit, ensuring serializability and recoverability at the cost of lower parallelism.

11) Discuss Implementation of Isolation 5 M

Ans:

1. Definition:

Isolation is one of the **ACID properties** of a transaction.
It ensures that **each transaction executes as if it were the only transaction in the system**, even though

multiple transactions may run concurrently.
In other words, **the intermediate results of a transaction are invisible to other transactions** until it commits.

2. Need for Isolation:

Without proper isolation, problems such as:

- **Dirty Read**
- **Non-repeatable Read**
- **Phantom Read**
  can occur due to concurrent access.

3. Implementation Techniques for Isolation:

DBMS implements isolation using **Concurrency Control Mechanisms** such as:

(a) Lock-Based Protocols

- Transactions use **locks** on data items before reading or writing.
- Types:
  - **Shared Lock (S):** For reading (multiple transactions can hold it).
  - **Exclusive Lock (X):** For writing (only one transaction can hold it).
- Examples:
  - **Two-Phase Locking (2PL)**
  - **Strict Two-Phase Locking (Strict 2PL)**

These protocols ensure serializable and isolated execution.

(b) Timestamp-Based Protocols

- Each transaction is assigned a **unique timestamp** when it starts.
- The order of transactions is determined by their timestamps.
- The DBMS ensures that transactions execute **in timestamp order**, preventing conflicts.
- Examples:
  - **Basic Timestamp Ordering**
  - **Thomas Write Rule**

(c) Optimistic Concurrency Control

- Transactions execute **without locking**, assuming minimal conflict.
- Before commit, a **validation phase** checks whether the transaction conflicts with others.
  - If no conflict → Commit.
  - If conflict → Rollback and restart.

This is efficient when data conflicts are rare.

(d) Multiversion Concurrency Control (MVCC)

- The system keeps **multiple versions** of a data item.
- Readers can access an **old version** while writers update a **new version**.
- Thus, readers and writers do not block each other.
- Used in databases like **PostgreSQL** and **Oracle**.

4. Isolation Levels (as per SQL Standard):

DBMSs allow different **levels of isolation** for performance tuning:

Level	Phenomena Prevented
Read Uncommitted	No isolation — dirty reads possible
Read Committed	Prevents dirty reads
Repeatable Read	Prevents dirty & non-repeatable reads
Serializable	Full isolation — prevents all anomalies

✅ **In Short:**
Isolation ensures that concurrent transactions do not interfere with each other.
It is implemented using **locks**, **timestamps**, **optimistic checks**, or **multiversion control**, maintaining consistency and correctness in concurrent environments.

12) How to handle bucket overflow in hash-based indexing mechanism? Explain 5 M

Ans:

1. Concept:

In **hash-based indexing**, records are stored in **buckets** based on a **hash function** applied to the search key.
A **bucket overflow** occurs when **multiple records hash to the same bucket**, and the bucket **runs out of space** to store new records.

---

2. Causes of Bucket Overflow:

- Poor choice of **hash function** (many keys hash to same bucket).
- **Too many records** inserted compared to bucket capacity.
- **Non-uniform distribution** of hash values.

---

3. Techniques to Handle Bucket Overflow:

(a) Open Hashing (Overflow Chaining / Separate Chaining)

- Maintain a **linked list (chain)** for each bucket.
- When a bucket is full, new records are added to an **overflow linked list**.

Example:

Bucket 1: [101] → [106] → [111]

All records hashing to bucket 1 are connected using a chain.

Advantages: Simple to implement, no data movement needed.
Disadvantage: Searching may become slow if the chain grows long.

(b) Closed Hashing (Open Addressing)

- Instead of creating overflow lists, the system **searches for another empty bucket** according to a probe sequence.

Common Methods:

1. **Linear Probing:** Next bucket = (h(key) + 1) mod N
2. **Quadratic Probing:** Next bucket = (h(key) + i^2) mod N
3. **Double Hashing:** Use a second hash function to find another location.

Advantage: No extra overflow list needed.
Disadvantage: May cause **clustering** and slow performance.

(c) Extendible Hashing

- Uses a **directory** that can grow dynamically.
- When a bucket overflows:
    1. The bucket splits into two.
    2. The directory size doubles if needed.
- The **hash function** uses more bits to distinguish between new buckets.

Advantage: Handles overflow efficiently and scales well with data.
Disadvantage: Requires extra directory storage.

(d) Linear Hashing

- Buckets are split **gradually** as the table grows (no full directory doubling).
- Uses a series of hash functions (h0, h1, …) and a **split pointer** that moves sequentially.

Advantage: Smooth expansion and efficient use of space.
Disadvantage: Slightly more complex management.

---

4. Summary Table:

Method	Approach	Advantages	Drawbacks
Overflow Chaining	Linked list of overflow records	Simple	Search time increases
Open Addressing	Search for next empty slot	No extra space	Causes clustering
Extendible Hashing	Directory doubling and bucket splitting	Dynamic and efficient	More complex
Linear Hashing	Gradual splitting	Smooth growth	Complex implementation

---

✅ **In Short:**
Bucket overflow in hash-based indexing occurs when multiple keys hash to the same bucket.
It is handled using techniques like **overflow chaining**, **open addressing**, **extendible hashing**, or **linear hashing**, each balancing simplicity, speed, and space efficiency.

*****