UNIT 4:DATA STORAGE AND QUERY PROCESSING

1. Physical Storage Media

Types:

1. **Primary Storage**:
    - Main memory (RAM)
    - Fastest access, volatile
2. **Secondary Storage**:
    - Magnetic disks (HDDs)
    - Non-volatile, large capacity
3. **Tertiary Storage**:
    - Optical disks (CD/DVD), tapes
    - Used for backups and archival
4. **Flash Memory**:
    - SSDs, USB drives
    - Faster than HDD, non-volatile

Characteristics:

- **Access Time**: Time to locate data
- **Transfer Rate**: Speed of reading/writing data
- **Capacity**: Amount of data stored
- **Volatility**: Loss of data on power-off

RAID (Redundant Array of Independent Disks)

Purpose:

- Improve reliability and performance of data storage using multiple disks.

Key Concepts:

- **Redundancy**: Provides fault tolerance.
- **Striping**: Distributes data across multiple disks.
- **Mirroring**: Replicates data on two or more disks.

RAID Levels

Level	Description	Fault Tolerance	Performance
RAID 0	Striping only, no redundancy	❌	✅ High read/write
RAID 1	Mirroring	✅	✅ High read, ❌ write
RAID 2	Bit-level striping with Hamming code	✅	❌ Rarely used
RAID 3	Byte-level striping with parity	✅	✅ Sequential access
RAID 4	Block-level striping with parity	✅	❌ Parity bottleneck
RAID 5	Block-level striping + distributed parity	✅	✅ Balanced
RAID 6	Like RAID 5 + extra parity	✅✅	✅ Better than RAID 5

File Organization

Methods of storing records in a file:

1. **Heap File (Unordered)**:
    - Records are inserted as they arrive.
    - No ordering.
    - Slow for search.
2. **Sequential File**:
    - Records are sorted based on a key.
    - Efficient for range queries.
    - Insertions/deletions may require reordering.
3. **Hash File**:
    - Uses hash function on key.
    - Fast access for exact match queries.
4. **Clustered File**:
    - Related records from different tables stored together.

Fixed and Variable Length Records

Fixed-Length:

- Each record has the same size.
- Simple to process.
- Example: Student(ID, Name, Marks) – fixed byte size.

Variable-Length:

- Records differ in size.
- More flexible, efficient storage.
- Need delimiters or offset tables.

Various Organizations of Records

Type	Description	Pros	Cons
Heap	Unordered	Fast insert	Slow search
Sorted	Ordered by key	Fast binary search	Slow insert/delete
Hashed	Based on hash function	Fast exact search	No range queries

Indexing – Basic Concepts

- **Index**: A data structure that speeds up retrieval of records.
- **Index Entry**: (Search key value, Pointer to record)

Types:

1. **Primary Index**: Built on primary key. Records sorted by key.
2. **Secondary Index**: Built on non-primary attributes.
3. **Dense Index**: Every search key appears in index.
4. **Sparse Index**: Index only on some search keys.
5. **Clustering Index**: Index on a non-key field that determines physical record order.

Types of Indexing

Index Type	Key Feature
Single-level index	One level of index
Multilevel index	Index of indexes (tree structure)
B-Tree index	Balanced tree structure
B+ Tree index	All values at leaf level, supports range queries
Hash index	Uses hash function for quick lookup

B-Tree Index Files

- Balanced m-ary search tree.
- Every node (except root) must be at least half full.
- Internal nodes store keys and pointers.
- **Supports**: Search, insert, delete in logarithmic time.
- Useful for range and point queries.

B+ Tree Index Files

- Extension of B-Tree.
- All keys appear at leaf level.
- Leaves are linked for fast range queries.
- Internal nodes only store keys (no data pointers).
- **Advantages**:
  - Efficient range queries.
  - Better space utilization.

Static Hashing

- Uses a fixed hash function.
- Each record is placed into a bucket.
- **Problems**:
  - Overflow if many records hash to same bucket.
  - Difficult to handle dynamic growth.

Bucket Overflow Handling:

- **Overflow chaining**: Link overflow buckets.
- **Open addressing**: Use probing to find next free slot.

Dynamic Hashing

- Hash table grows/shrinks dynamically.
- Uses **directory** and **bucket** structure.
- **Directory**: Points to buckets, may grow in size.
- **Extendible Hashing**:
  - Increases the number of bits used in hash function.
  - Handles growth efficiently.
- **Linear Hashing**:
  - Uses a series of hash functions.
  - Buckets split gradually.

Query Processing – Overview

Definition:

Query processing is the series of steps a DBMS uses to translate a high-level query (e.g., SQL) into a low-level sequence of operations that access data efficiently.

Phases:

1. **Parsing and Translation**: SQL is parsed and translated into a relational algebra expression.
2. **Optimization**: Multiple strategies are considered; the best (least cost) one is chosen.
3. **Evaluation**: Execution plan is run to get the result.

Components:

- **Query Parser**: Checks syntax and converts to internal representation.
- **Query Optimizer**: Chooses the best strategy based on cost.
- **Query Executor**: Executes the optimized query.

Measures of Query Cost

Goal: Minimize the total cost of query execution.

Key Cost Measures:

1. **Disk I/O Cost**:
   - Reading/writing data blocks from/to disk.
   - Most significant cost in query processing.
2. **CPU Cost**:
   - Includes comparisons, hash computations, sorting, etc.
   - Important for in-memory operations.
3. **Communication Cost** (in distributed systems):
   - Cost to send data over a network.

Total Cost = Disk I/O + CPU (dominantly disk I/O in large databases)

Selection Operation

Purpose:

Retrieve rows from a table that satisfy a given condition (σ condition(R)).

Evaluation Strategies:

1. **Linear Search**:
    - Scan each record.
    - Costly: O(n)
2. **Binary Search**:
    - On sorted file.
    - Cost: O(log n)
3. **Index Search**:
    - Uses primary/secondary index.
    - Efficient for equality or range search.
4. **Selection with Hashing**:
    - Use hash function if the search condition matches hash key.

Examples:

- σ_RollNo = 10(Student)
- σ_Age > 20(Employee)

Sorting

Purpose:

Required for operations like ORDER BY, merge-join, and duplicate elimination.

Algorithms:

1. **External Merge Sort**:
    - For large data that can't fit in memory.
    - Steps:
        - Create sorted runs in memory.
        - Merge runs.
    - Cost: O(n log n)
2. **Two-Way Merge Sort**:
    - Used when limited buffer space is available.
3. **Replacement Selection**:
    - Create longer runs using heap; improves efficiency.

Join Operation

Purpose:

Combine related tuples from two relations.

Common Join Types:

- **Theta Join (R ⋈θ S)**: Condition-based.
- **Equi-Join**: Condition is equality.
- **Natural Join**: Equi-join with duplicate attributes removed.

Join Algorithms:

1. **Nested Loop Join**:
   - For each tuple in R, scan S.
   - Cost: O(m × n)
2. **Block Nested Loop Join**:
   - Loads a block of tuples to reduce disk I/O.
   - More efficient than simple nested loop.
3. **Index Nested Loop Join**:
   - Uses index on inner relation.
   - Good for small outer, indexed inner.
4. **Sort-Merge Join**:
   - Sort both relations, then merge.
   - Efficient for sorted data.
5. **Hash Join**:
   - Build phase: hash one relation.
   - Probe phase: match with other relation.
   - Best for equality joins and large datasets.

Evaluation of Expressions

Goal:

Efficiently evaluate relational algebra expressions using an execution strategy.

Expression Tree:

- Tree representation of relational algebra expressions.
- Leaf nodes: base relations.
- Internal nodes: operations (σ, π, ⋈).

Evaluation Techniques:

1. **Materialization**:
   - Compute and store intermediate results on disk.
   - Simple but uses more space.
2. **Pipelining**:
   - Pass results from one operation to the next without storing.
   - Saves space and improves performance.

Choice Depends On:

- Available memory
- Expected intermediate result size
- Operator associativity and commutativity