**UNIT 4:DATA STORAGE AND QUERY PROCESSING**

# Physical Storage Media

**Types:**

1. **Primary Storage**:
   * Main memory (RAM)
   * Fastest access, volatile (Primary memory is faster than secondary storage because it has a direct connection to the CPU, allowing quicker data access and processing.
2. **Secondary Storage**:
   * Magnetic disks (HDDs)
   * Non-volatile, large capacity ("non-volatile" means that data remains intact even when power is removed or the system is shut down.)
3. **Tertiary Storage**: (a long-term, off-line storage system, typically used for archiving and backup)
   * Optical disks (CD/DVD), tapes o 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)

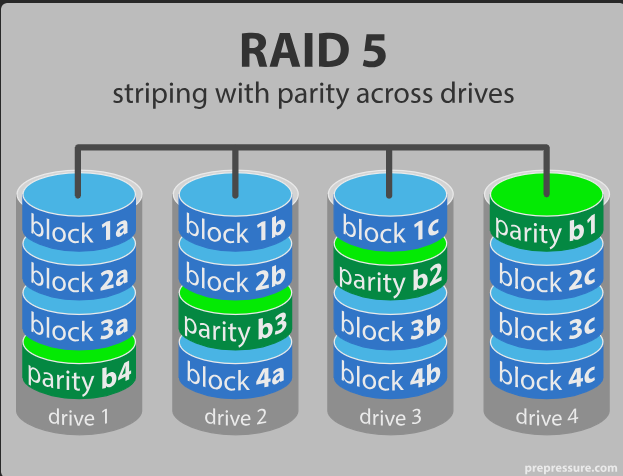
is a technology that combines multiple physical disk drives into a single logical unit for data storage.

**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)**:

o Records are inserted as they arrive.

o No ordering.

* + Slow for search.

1. **Sequential File**:
   * Records are sorted based on a key.
   * Efficient for range queries.
   * Insertions/deletions may require reordering.
2. **Hash File**:
   * Uses hash function on key.
   * Fast access for exact match queries.
3. **Clustered File**: ( we use clusters concepts )
   * 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) : **An index entry** is a **basic data structure component** used to **speed up the retrieval of rows** from a table.

**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**: o Efficient range queries. o Better space utilization.

# Static Hashing

* Uses a fixed hash function.
* Each record is placed into a bucket.
* **Problems**:

o Overflow if many records hash to same bucket. o 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. o 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**: o Scan each record.
   * Costly: O(n)
2. **Binary Search**: o 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**:

o Used when limited buffer space is available.

1. **Replacement Selection**:
   * Create longer runs using heap; improves efficiency.

# Join Operation

**Purpose:**

Combine related tuples from two relations.

**Common Join Types:**

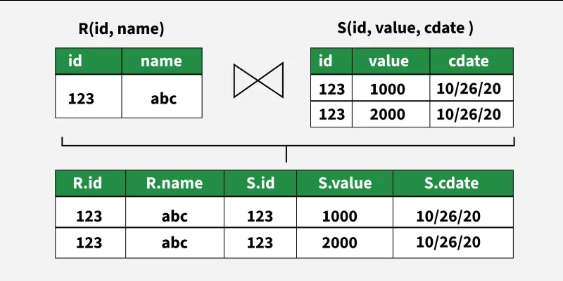
* **Theta Join (R** **θ S)**: Condition-based.
* **Equi-Join**: Condition is equality.
* **Natural Join**: Equi-join with duplicate attributes removed.

**Join Algorithms:**

1. **Nested Loop Join**:

o For each tuple in R, scan S. o Cost: O(m × n)

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



# Evaluation of Expressions:

**Evaluation of expressions** in databases refers to the process of **computing the result of a query or expression** using the available data, typically involving operations like selection, projection, joins, aggregation, and arithmetic or logical computations.

**Goal:**

Efficiently evaluate relational algebra expressions using an execution strategy.

The main purposes are:

1. **To compute correct results**:  
   Ensure the database returns the expected answer for a given query or expression.
2. **To optimize performance**:  
   Choose the most **efficient way to execute** a query — especially important for large datasets. This includes minimizing disk access, memory use, and processing time.
3. **To transform high-level queries into executable steps**:  
   Convert complex SQL queries into **low-level operations** (like scans, joins, sorts) that the database engine can perform.
4. **To choose appropriate algorithms and data structures**:  
   Decide how to evaluate each part of a query — for example, whether to use **hash join** vs **nested loop join**, or **index scan** vs **table scan**.

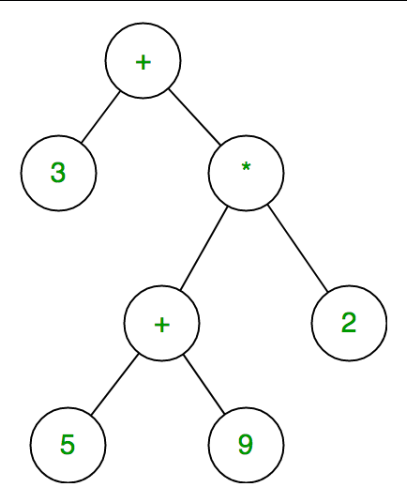
**Expression Tree: (**The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand)

### **Key Features of Expression Tree**

* **Tree Structure**:  
  Each node in the tree represents either an operator (like **select**, **project**, or **join**) or an operand (like a **table/relation**).
* **Leaf Nodes**:  
  These are the **base relations (tables)** involved in the query.

Example: Employees, Departments

* **Internal Nodes**:  
  These are **operators** such as:
  + **σ (Selection)**: Filters rows
  + **π (Projection)**: Selects columns
  + **⋈ (Join)**: Combines tables
  + **∪, ∩, −**: Set operations
  + **ρ (Rename)**, **× (Cartesian product)**



**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.

### 🔹 **2. Materialization**

**Definition**:  
**Stores intermediate results** of subqueries or operations in temporary tables before continuing with the next step.

**How it works**:  
Each step of the query is fully executed and stored, then passed to the next operator.

### 🔹 **3. Pipelining**

**Definition**:  
The output of one operation is **immediately fed** into the next without storing intermediate results on disk.

**How it works**:  
Operators are connected in a chain, and each processes input and returns output on the fly.

**Choice Depends On:**

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