VIDEO PRESENTATION LINK

Link: https://drive.google.com/file/d/1riJaQHRpdk-AQqe1KpdROvd8kgvoSKav/view?usp=sharing

Members:

Saptarshi Acharya (22BCE1134)

Rishabh Shital Ramdhave (22BCE1824)

Abhishek Kumar Singh (22BCE1867)

OPTIMIZED RETRIEVAL AND QUERY PROCESSING: A SYNERGISTIC APPROACH FOR IMPROVED DATABASE PERFOMACE

Saptarshi Acharya 22BCE1134

Rishabh Shital Ramdhave 22BCE1824

Abhishek Kumar Singh 22BCE1867

INTRODUCTION

Problem with Traditional Systems:

Traditional DBMS face challenges in handling large datasets efficiently, with issues like scalability and slow query performance due to reliance on basic techniques like Boolean retrieval and simple indexing.

Proposed Solution:

This paper introduces a novel approach that combines Caching Techniques (Splay Trees and Randomized Search Trees) with Cost-Based Query Optimization (CBO) to improve both data retrieval and query processing.

Caching with Splay Trees and RSTs:

Splay Trees and RSTs could be used in DBMS, optimize data retrieval by maintaining dynamic and efficient access paths, reducing retrieval time and improving cache performance.

Impact on Performance:

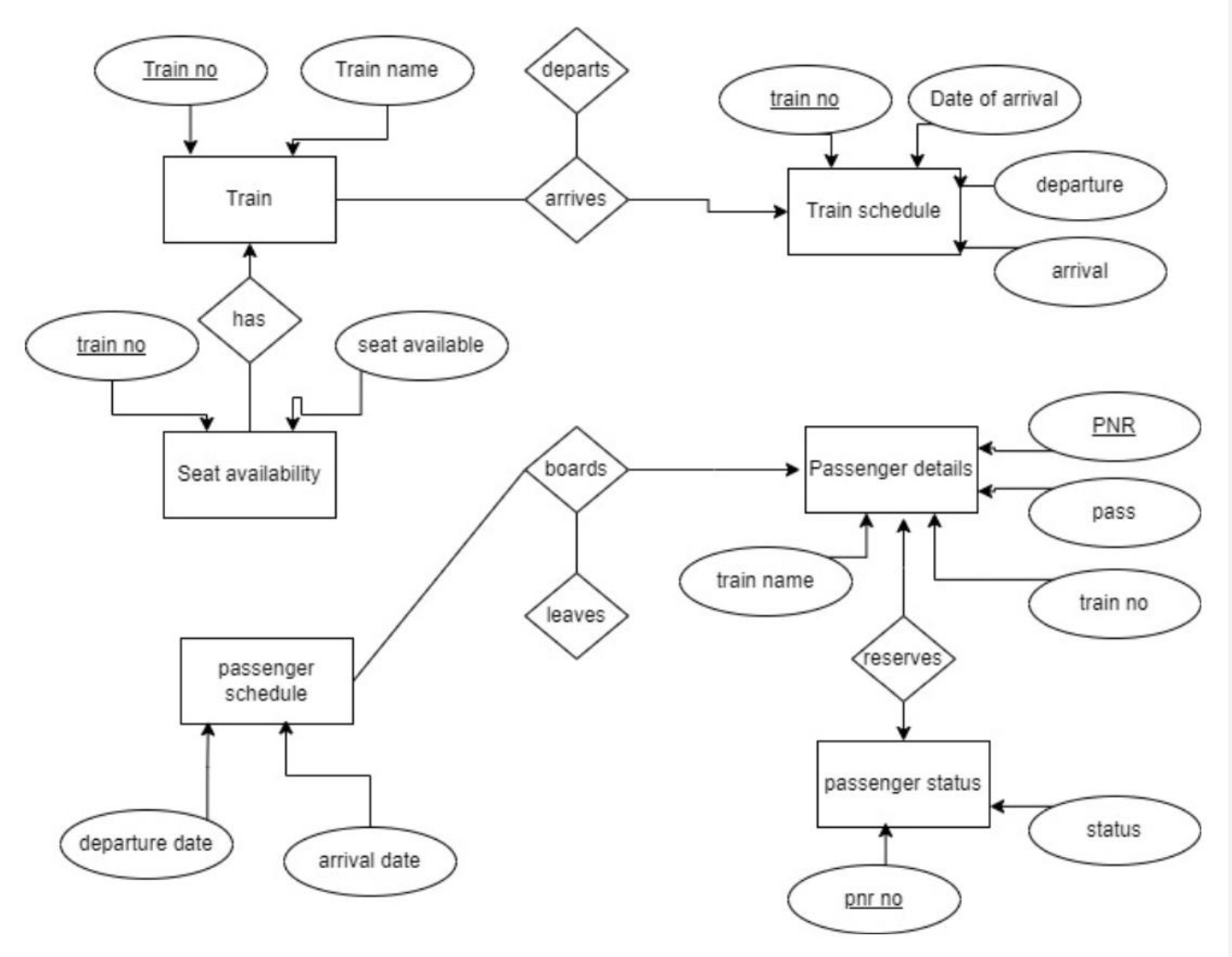
By integrating Caching and Cost-Based Optimization, the proposed method reduces query latency, enhances retrieval speed, and improves system scalability, addressing the performance bottlenecks in traditional systems.

INTRODUCTION

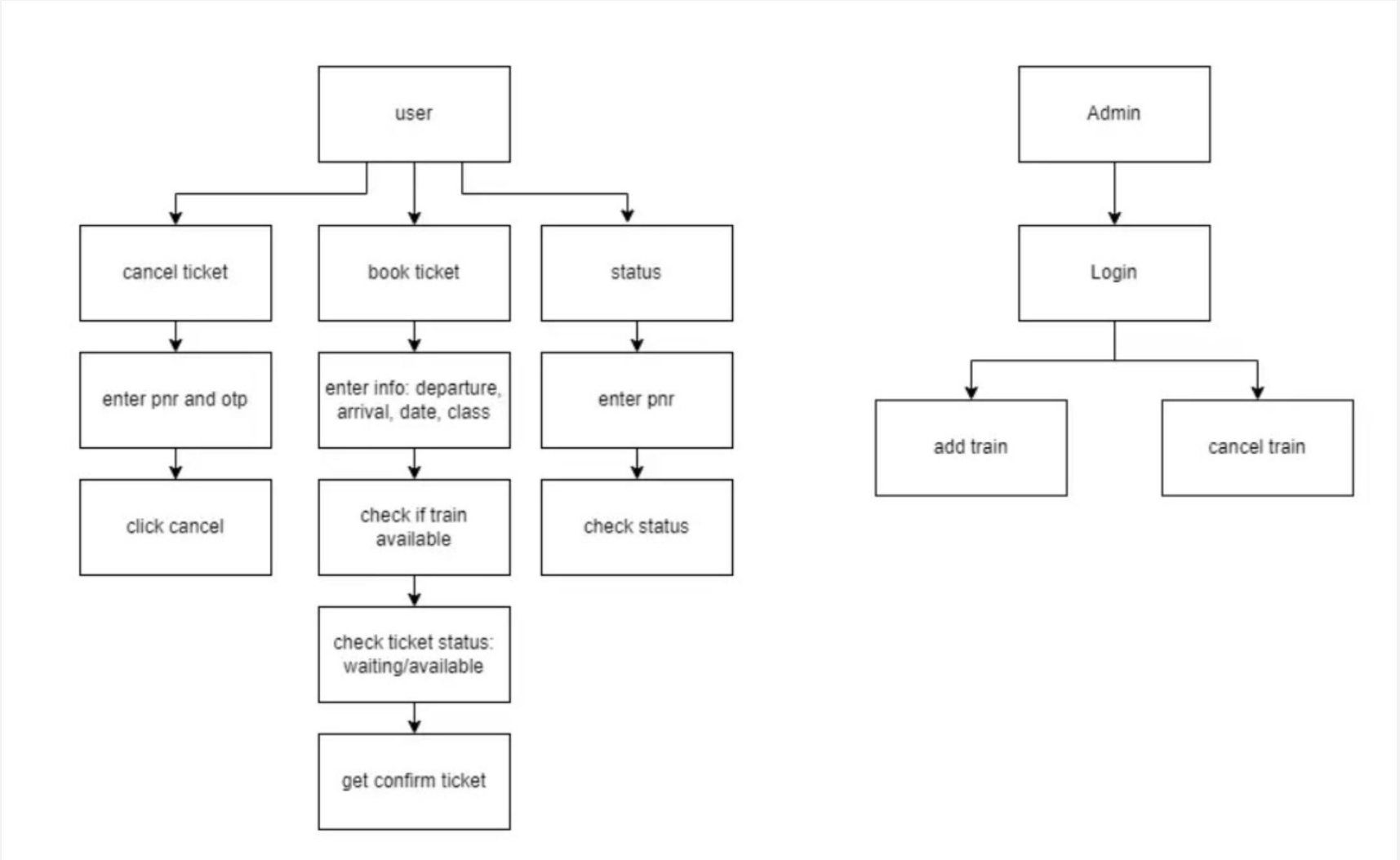
Potential Implementation in Indian Railways:

The proposed approach could be applied to improve the performance of the Indian Railway's Reservation and Ticketing System (IRCTC), which currently suffers from slow query processing. By implementing these optimization techniques, query response times can be significantly reduced, enhancing user experience and system efficiency.

ER DIAGRAM


Train no
Train name
departs
train no
Date of arrival
Train
arrives
Train schedule
departure
arrival
has
train no
seat available
Seat availability
boards
Passenger details
PNR
pass
train name
train no
leaves
reserves
passenger schedule
passenger status
status
departure date
arrival date
pnr no

FUNCTIONAL DIAGRAM (HIGH LEVEL DIAGRAM)



CONVENTIONAL STRATEGIES AND TECHNIQUES

Boolean Retrieval:

Boolean retrieval uses logical operators (AND, OR, NOT) to filter and retrieve documents matching specific conditions. This approach is straightforward and computationally inexpensive, commonly used for exact matches.

Demerit: It lacks the ability to rank documents by relevance, leading to either an overwhelming number of results or excluding potentially relevant information in complex queries, thus limiting accuracy.

Vector Space Model (VSM):

VSM represents both queries and documents as vectors in a high-dimensional space, where the relevance of each document is calculated based on cosine similarity between vectors. This approach enables ranked retrieval of results by similarity scores.

Demerit: It is computationally expensive for large datasets due to high-dimensional vector calculations and lacks semantic understanding, making it difficult to capture the deeper meaning of complex or ambiguous queries.

CONVENTIONAL STRATEGIES AND TECHNIQUES

B-tree Indexing:

B-tree indexing organizes data hierarchically, allowing for efficient search and retrieval, especially for range queries. B-trees are widely used due to their balanced structure, which maintains query efficiency across different data sizes.

Demerit: With frequent insertions, deletions, and updates, B-trees can require frequent rebalancing, increasing maintenance overhead. Additionally, large datasets can cause B-trees to become deep, resulting in slower searches as more nodes are traversed.

Materialized Views:

Materialized views store precomputed query results, which can then be quickly retrieved without recalculating. This approach is effective for complex queries or reporting, as it reduces the computational cost for repeated queries on static datasets.

Demerit: Materialized views require substantial storage and constant maintenance to stay synchronized with underlying data. In dynamic datasets, updating views frequently can add significant overhead, potentially offsetting the performance gains from caching results.

OUR PROPOSED WORK

We propose the following strategies and techniques to improvise the database management system:

1. Caching Techniques – Splay trees and Treaps (Randomized Search Trees)
2. Cost Optimized Techniques – Heuristic based filtering and searching for query optimizations

CACHING TECHNIQUES

- We will primarily be focusing on the following two types of trees for the sake of caching optimizations:
  - **Splay Trees**
  - **Treaps (Randomized Search Trees)**

- While traditional DBMS caching techniques like B-trees are optimized for disk access and work well for larger datasets, **splay trees and treaps excel in in-memory caching** where frequent, dynamic access patterns are present.

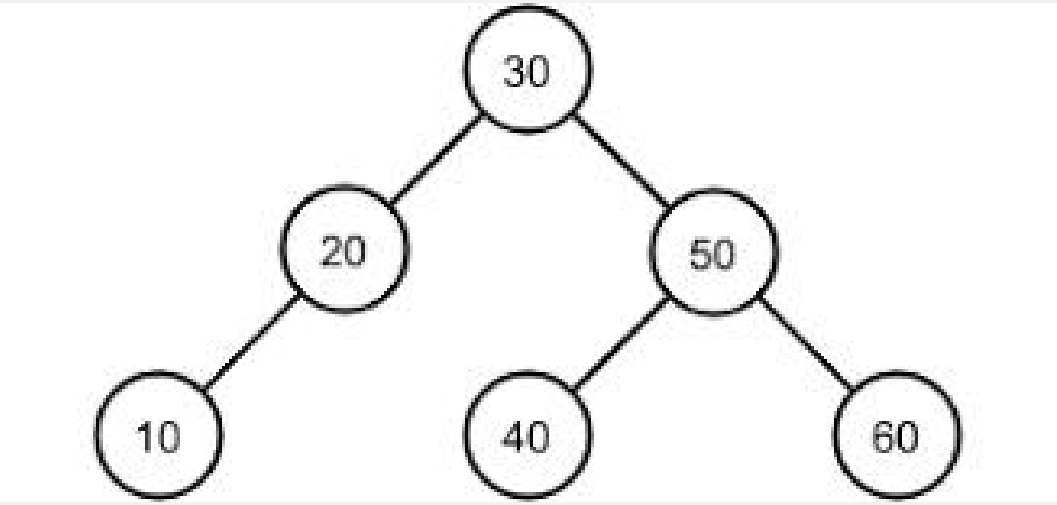
- Their **self-balancing properties** and **lower metadata requirements** make them more efficient in scenarios requiring adaptability, frequent updates, and low overhead.

SPLAY TREES

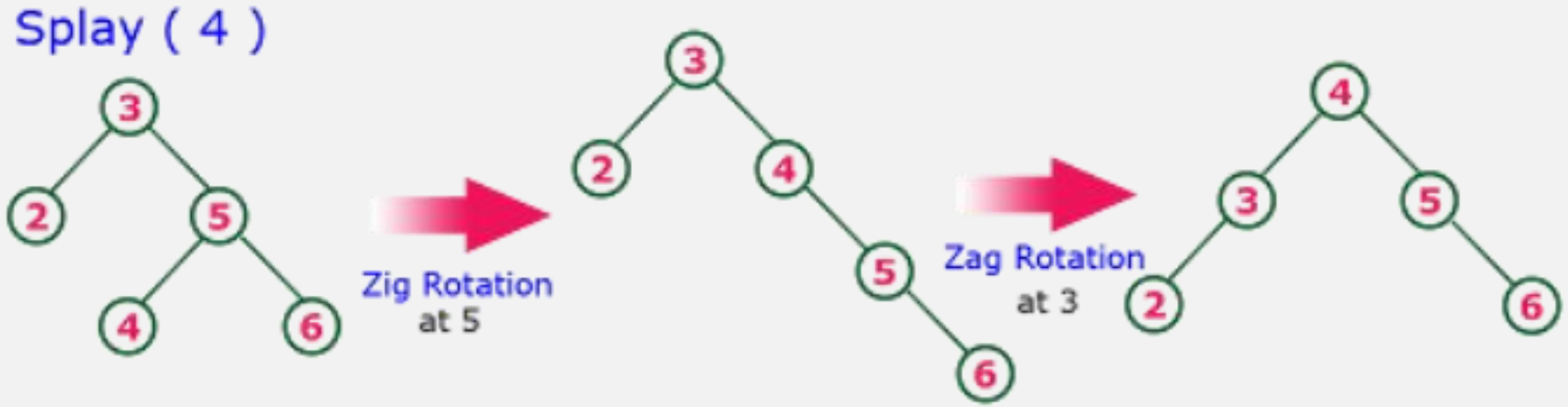
- A **splay tree** is a type of **self-adjusting binary search tree** that reorganizes itself to move recently accessed elements closer to the root. It achieves this reorganization through a process called "splaying," where a node is repeatedly rotated to the root of the tree whenever it is accessed, inserted, or deleted. This means that elements accessed frequently become faster to access over time, a property known as *temporal locality*.

- **Key Features of Splay Trees:**
  - **Self-Adjusting**: Splay trees dynamically restructure based on access patterns, promoting frequently accessed nodes to the top, which helps reduce access time for hot items.
  - **No Balance Factor**: Unlike AVL or Red-Black trees, splay trees **do not require additional data for balancing**. The self-adjusting nature of splay trees provides amortized O(log n) performance over a sequence of operations.
  - **Amortized Efficiency**: Splay trees guarantee O(log n) amortized time for basic operations (search, insert, delete), though individual operations can take up to O(n) in the worst case. However, the amortized efficiency makes them ideal for applications where access patterns are skewed toward certain items.

IMPLEMENTATION



- In **splay trees**, operations like insertion, deletion, and search are followed by a restructuring process called *splaying*, which brings the accessed or modified node to the root of the tree.
- This self-adjusting property ensures that frequently accessed nodes remain close to the top, optimizing the tree based on access patterns.
- All the operations are followed by zig-zag rotations which maintain this property of splay trees.



WHY SPLAY TREES?

- **Better Suitability for Caching Layers**: Conventional trees like B-trees or AVL trees are highly structured and balanced, designed for consistent performance across all data points. In caching scenarios, where certain data points are accessed more frequently, the self-adjusting behavior of splay trees reduces access times for hot data without the need for an explicit MRU structure.
- **Adaptability to Dynamic Workloads**: DBMS workloads can vary widely over time, and splay trees are adaptable to changing access patterns. For example, if a particular data entry becomes the focus of repeated queries, a splay tree will automatically bring it closer to the root, minimizing access times as long as it remains relevant.
- **Reduced Overhead in Memory-Constrained Scenarios**: Splay trees are efficient in terms of space, with no need for balance-tracking pointers or color markers. In-memory databases or caching layers within DBMSs that require quick, space-efficient access can benefit from this simplicity.
- **Amortized Performance in High-Churn Environments**: In applications with high turnover of accessed elements, such as user activity logs or session data, splay trees handle frequent insertions and deletions efficiently on average. Their amortized O(log n) performance is advantageous when large numbers of elements are regularly updated or removed.
- **Simpler Implementation for MRU Behavior**: Splay trees inherently bring the most recently accessed nodes to the root, reducing the need for additional structures to track recent data. In comparison, LRU caches implemented with AVL trees require auxiliary data structures or costly rebalancing.

TREAPS: RANDOMIZED SEARCH TREES

- A **treap** is a type of **randomized binary search tree** (BST) that combines properties of a binary search tree and a heap. It is also known as a **randomized search tree** (RST). Treaps are designed to maintain a balanced tree structure probabilistically, using a random priority assigned to each node in addition to the key values that maintain the binary search tree property.
- **Key Features:**
  - **Combination of BST and Heap**:

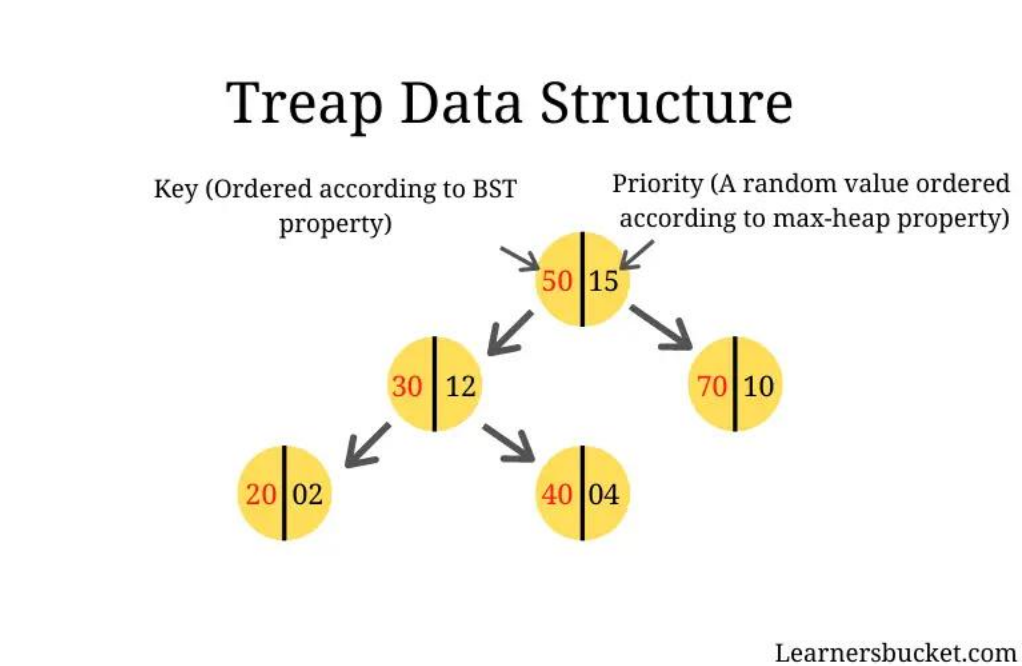
    It combines the key ordering of a binary search tree and the heap property based on random priorities.
  - **Randomized Balance**:

    It ensures the tree remains balanced on average by relying on random priorities rather than rigid balancing rules.
  - **Efficient Search, Insert, and Delete**:

    These operations are efficient with $O(\log n)$ expected time complexity.
  - **Space Efficiency**:

    Treaps only require space for the nodes, as they don't need extra data for balancing (like AVL trees or RB trees).

IMPLEMENTATION



- **Search**: Searching in a treap follows the same procedure as in a standard BST. Due to the randomized balancing, the expected time complexity for searching is O(log n) , though it can be worse in the worst case.
- **Insert and Delete**: Insertion and deletion operations also follow the same BST principles. However, after an insertion or deletion, the heap property is restored by performing rotations (like in a heap) to ensure the parent node's priority is greater than its children's priorities.
- These operations generally run in O(log n) time on average, though worst-case time can degrade depending on the random priority distribution.

WHY TREAPS?

- **Probabilistic Balancing**: Treaps use random priorities for balancing, avoiding complex rebalancing operations like in AVL or Red-Black trees.
- **Simpler Operations**: Insertions, deletions, and searches are simpler with fewer balancing steps, leading to easier implementation.
- **Efficient Expected Time Complexity**: Expected time for search, insertion, and deletion due to randomized balancing.
- **No Explicit Rebalancing**: Eliminates the need for explicit balancing (rotations), making operations faster in dynamic environments.
- **Space Efficiency**: Requires only the key and priority, reducing overhead compared to AVL or Red-Black trees that store extra balancing information.
- **Ideal for Dynamic Data**: Suitable for systems with frequent insertions, deletions, and updates, providing consistent performance.
- **Good for Randomized Access**: Performs well in scenarios with unpredictable or random access patterns, unlike B-trees or Red-Black trees.
- **Implicit Priority Queue**: Supports priority-based operations efficiently, useful for task scheduling and query optimization.

COST BASED OPTIMIZATIONS

- A query optimization method that chooses the lowest-cost execution plan based on estimated resource consumption.
- **How It Works**:
  - Uses data statistics (table sizes, indexes, data distribution).
  - Estimates costs in terms of CPU, memory, and I/O usage.
- **Goal**: Selects the most efficient query plan to reduce execution time and resource use.

LOGICAL AND PHYSICAL TRANSFORMATIONS

- **Logical Transformation**:
  Focuses on changing the structure of a query without altering its result.
  Examples: Join reordering, predicate pushdown.

- **Physical Transformation**:
  Refines the actual execution of the query with different access methods.
  Examples: Index scans, different join algorithms.

- **Purpose**:
  Both transformations aim to reduce execution cost but operate at different stages of query processing.

INTERLEAVING AND JUXTAPOSITION

- **Interleaving**:
  - Combines transformations during the optimization process, allowing for more dynamic and adaptable query plans.
  - Example: Combining join reordering with predicate pushdown for better optimization.
- **Juxtaposition**:
  - Separates transformations into distinct steps, each optimizing a specific aspect.
  - Can simplify optimization but may miss opportunities for combined efficiencies.
- **Trade-Off**:
  - Interleaving offers flexibility, while juxtaposition is simpler but potentially less effective.

STATE SPACE SEARCH TREES

Exhaustive Search:

Evaluates all possible plans to find the optimal solution.
Pros: Guarantees optimal result.
Cons: Time-consuming, not practical for complex queries.

- **Iterative Search**:

Gradually refines the plan through iterative improvements.
Pros: Balances optimization and efficiency.
Cons: May not find the absolute optimal plan.

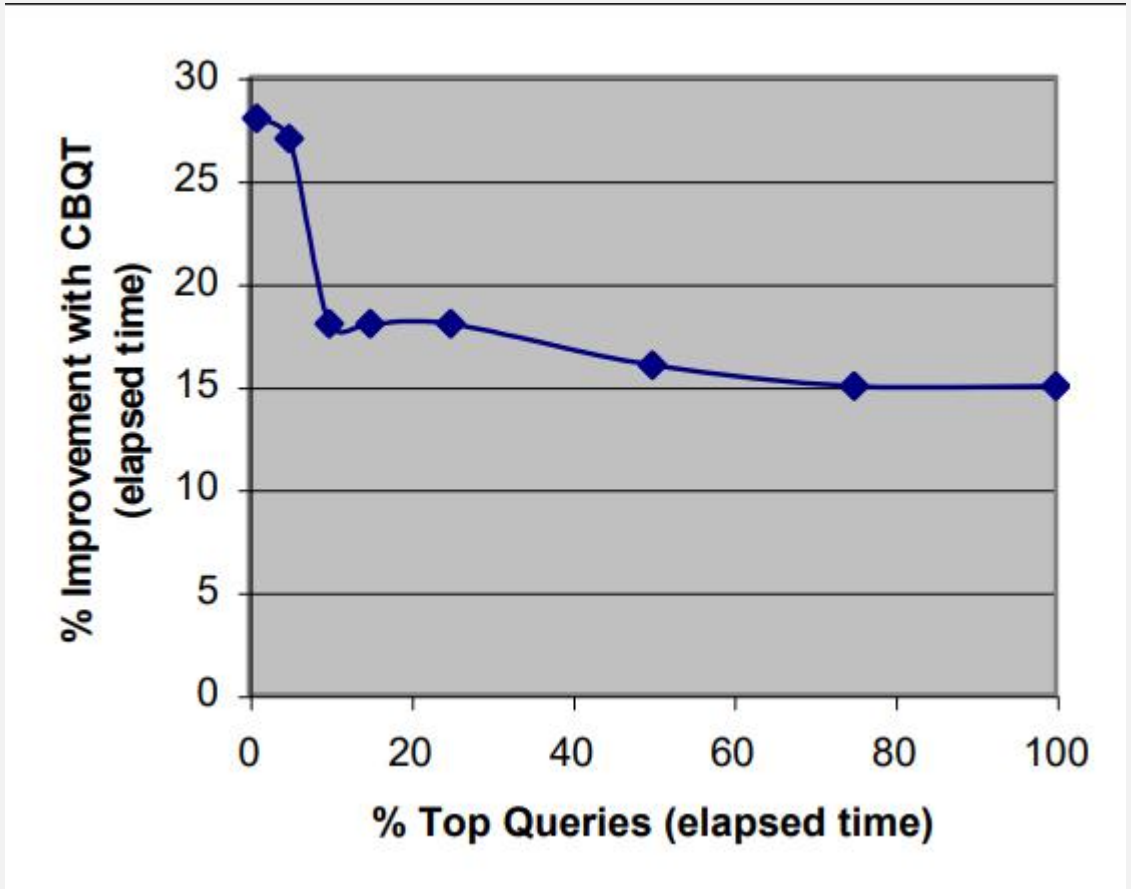
- **Linear Search**:

Follows a fixed set of transformations with no backtracking.
Pros: Fast and simple.
Cons: Limited flexibility, may result in suboptimal plans.

- **Optimization Strategy**:

Choosing the appropriate search method depends on query complexity and performance requirements.

IMPACT OF COST-BASED OPTIMIZATIONS



- **Performance Improvement**:
  Reduction in execution time and resource consumption.

- **Efficiency Gains**:
  Graph illustrates comparisons with and without cost-based optimization, showing enhanced performance in optimized queries.

- **Conclusion**:
  Cost-based optimization provides crucial performance benefits, especially for complex, high-volume queries.