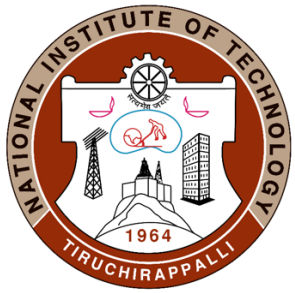


NATIONAL INSTITUTE OF TECHNOLOGY
TIRUCHIRAPPALLI

INDEXING POSSIBILISTIC NUMERICAL DATA USING INTERVAL B+ TREE

PREPARED BY

YESVIN V (106122143)
SANKARA NARAYANAN (106122107)
APRIL 17 2024

ABSTRACT :

This research introduces an indexing methodology tailored to manage extensive record sets featuring interval data. The study proposes a new indexing method called Interval B+-tree (IBPT), specifically designed to handle large datasets containing interval data, commonly encountered in possibility-based relational databases and various application domains. The IBPT is an innovative adaptation of the popular B+-tree indexing technique that addresses the challenge of efficient storage and retrieval of interval data, offering enhanced capabilities to expedite query processing. This paper explains the design and implementation of the IBPT index and showcases its application to optimize the evaluation of fuzzy queries in possibilistic relational databases. Empirical evaluation demonstrates a significant improvement in query performance, highlighting the IBPT's potential to enhance the efficiency and functionality of systems dealing with interval-based data structures.

INTRODUCTION :

A relational database comprises structured records stored across sectors of a hard disk. Users employ databases to store and retrieve information through querying, which involves filtering records based on specific criteria. Query evaluation entails retrieving records that meet the query criteria. To expedite this process, databases utilize indices to avoid loading unnecessary disk blocks. These indices, tailored for various data types, rely on attribute values for record lookups. However, classical indexing techniques are inadequate for possibility-based databases, where attribute values can be fuzzy. Therefore, new indexing techniques are necessary. This paper introduces the Interval B+-tree (IBPT), an adaptation of the composite B+-tree, tailored for possibilistic values denoting fuzzy intervals. The IBPT offers improved performance compared to existing B+-tree-based index structures for fuzzy values. The subsequent sections detail possibility-based database modelling, B+-tree indices, and the IBPT index, followed by experimental results.

METHODOLOGY AND PROCEDURES:

The research design of this study uses an experimental approach to create and evaluate the Interval B+-tree (IBPT) indexing technique, which is specifically tailored for interval data within possibility-based relational databases. In the developmental phase of the study, the traditional B+-tree structure is modified to accommodate interval data, requiring the creation of new node structures and algorithms designed for interval operations. Following this, a comprehensive evaluation of IBPT's effectiveness and efficiency is conducted through a series of meticulously designed experiments. Synthetic datasets are created to simulate a wide range of scenarios, while real-world datasets are selected to provide insight into IBPT's performance in practical applications. The study utilizes quantitative analysis to measure key metrics such as query execution time, index construction time, and storage overhead, allowing for a thorough comparison with existing indexing techniques such as 2(A)BPT and CBPT. In addition, qualitative insights are provided to explain the unique advantages and potential limitations of IBPT based on empirical findings. Lastly, the research findings undergo rigorous validation through the peer review process to ensure the experimental methodology is robust, and the results are credible

POSSIBLITY BASED DATABASE MODELLING :

When retrieving records with specific attribute values from a possibilistic database, it is essential to represent any uncertainty inherent in the dataset accurately. If the exact attribute value of a record is unknown, the focus shifts from binary satisfaction to assessing the degree of possibility that a record meets the query criteria. This assessment produces two vital measures: the possibility degree (π) and the necessity degree (N). These metrics indicate the extent to which a record's attribute value plausibly and confidently aligns with the query condition.

The process of querying a possibilistic database involves identifying records where the possibility degree (π) exceeds zero, indicating potential matches. Conversely, records with a possibility degree of zero are unequivocally excluded from the results due to their inability to fulfil the criteria.

If the query involves fuzzy criteria characterized by fuzzy sets, the compatibility operator “IS” is utilized. This operator gauges the similarity between an uncertain attribute value (represented by π_A) and a fuzzy user preference (represented by p_A) based on their respective membership functions. The compatibility operator helps compute the possibility and necessity degrees, denoting query satisfaction, accordingly.

1. A IS P(r) = sup {min(π_A(v), p_A(v)) : v ∈ dom(T)}
2. N_A IS P(r) = inf {max(1 - π_A(v), p_A(v)) : v ∈ dom(T)}

PRESELECTION PRINCIPLE :

In the realm of fuzzy databases, determining whether a fuzzy datum aligns with a fuzzy query can be resource intensive. This process involves comparing functions point by point and identifying extreme values for each record, which is a computationally demanding task. However, there are instances where this evaluation process can be circumvented.

The preselection principle, pioneered by Bosc et al., identifies specific conditions under which the resulting possibility and necessity outcomes consistently yield zero, indicating clear incompatibility with the query criteria. This principle essentially allows for the immediate rejection of records based on predetermined criteria, thereby bypassing the need for complex compatibility evaluations.

The key condition stipulates that if both the fuzzy value and the fuzzy query exhibit convex membership functions, it suffices to verify that their supports do not overlap. For instance, when evaluating a query criterion denoted as A IS P, where A represents an attribute and P signifies user preferences expressed as a trapezoidal fuzzy set, rejecting a record can be determined by simple checks ensuring that the record's support boundaries do not intersect with the upper or lower boundaries of the query's support.

This principle has spurred the development of various indexing techniques tailored for interval and possibilistic data. For example, Bosc et al. proposed an index utilizing superimposed coding to annotate the support and core of possibility distributions, while Yazici et al. introduced a multidimensional index catering to similarity-based fuzzy data. Additionally, Liu et al. presented 1GT, a technique leveraging G-trees, and Barranco et al. proposed 2BPT, employing two B+-trees to index convex trapezoidal possibility distributions.

Subsequent works have further explored indexing structures for scalar fuzzy data and addressed necessity-based queries. Furthermore, extensive evaluations of these indexes have been conducted in both possibility and necessity contexts.

$$\Pi r\,(a) > Pd\,(3) \quad \& \quad \Pi r\,(d) < Pa$$

BALANCED TREE INDEX STRUCTURES:

1.) B+TREES : ROBUST INDEXING METHODOLOGY

B+-tree (BPT) is one of the most efficient and widely used indexing techniques for organizing numerical values in block-based data stores, such as hard disks. The balance of B+-trees is inherent, which means all leaf nodes are uniformly situated from the root, making them highly effective in block-based storage systems. In B+-trees, data is arranged in an ordered manner, which facilitates quick execution of exact value look-ups and range queries. You can see the visual representation of BPT nodes' structural composition in Figure 3. The order of B+-trees, also known as 'b', signifies the maximum fan-out of each internal node. The determination of this order is pivotal and is typically informed by the size of disk blocks. This ensures seamless correspondence between nodes and disk blocks, which is critical in computing the requisite number of disk blocks to satisfy a given query criterion. Further experiments will elucidate this alignment.

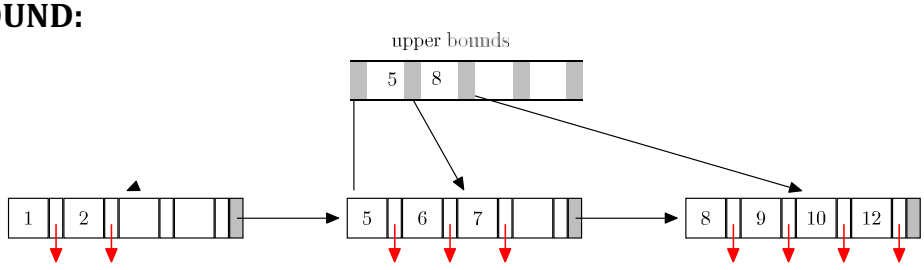
2.) COMPOSITE B+ TREES : OPTIMIZING QUERY EFFICIENCY

When a single attribute index fails to improve query performance, the database optimizer may perform a full table scan. This happens when the attribute lacks distinctiveness, such as in the case of gender categorization resulting in limited groups. Composite indexing, which uses multiple attributes, is employed to overcome this limitation and speed up queries. The essence of composite B+-tree (CBPT) indexing lies in the strategic arrangement of attributes. By sorting the indexed data based on one attribute and using subsequent attributes as tiebreakers, CBPT maximizes distinctiveness. For instance, sorting names by last name followed by first name creates a composite index with enhanced discernibility. With n attributes, CBPT allows for the creation of n! distinct composite indices.

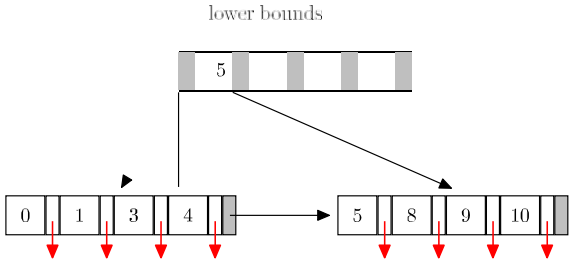
During querying, the database uses a composite index if the query constraints align with the first attribute indexed. In a CBPT, traversal begins with the first attribute, navigating the tree to locate the leaf node containing the initial record that satisfies the query constraint. Then, all records within this leaf node are examined to identify matches. Subsequently, the Pnext list facilitates navigation across leaf nodes until the first non-matching record for the initial constraint is found. Given the index structure, subsequent records are guaranteed to not satisfy the initial constraint.

It's worth noting that a CBPT incurs a larger footprint compared to a regular BPT of the same order due to either increased node count or denser node occupancy, a consequence of storing additional bytes per indexed record for each additional attribute.

UPPERBOUND:



LOWER BOUND:



3.) 2BPT : ENHANCING FUZZY NUMERICAL DATA

In 2008, Barranco et al. introduced the 2BPT index as an advanced technique for handling fuzzy numerical data. Unlike CBPT, which also caters to uncertain data, 2BPT uses two B+-trees together, earning its name. The 2BPT index is based on the integration of preselection criteria and set theory. Each preselection criterion acts as a constraint, delineating a subset of the data set. For a given query, the preselection set represents the intersection of these subsets. The 2BPT index is constructed by identifying intervals that do not satisfy specific criteria, forming the basis of range queries on B+-trees constructed from interval data. The index involves inserting interval upper bounds into one B+-tree index and lower bounds into another.

To compute the preselection set for a query with convex membership function P, three steps are involved: first, finding intervals where the upper bound constraint is not met; second, identifying intervals where the lower bound constraint is not met; and finally, intersecting these sets to determine the preselection set. It is important to note that the performance of the 2BPT index is tied to the number of leaf nodes, which impacts both efficiency and relevance of disk block utilization. Therefore, there is a potential for optimizing the 2BPT index to minimize unnecessary disk block reads and enhance overall performance.

4.) **2ABPT**:

In their work presented in [16], Medina et al. proposed a modification of the 2BPT index called 2ABPT. This new index replaces the BPT indices with CBPT indices, one focusing on upper bounds and the other on lower bounds. This change simplifies the computation of the preselection set by consolidating all interval data within each tree. It also eliminates the need to traverse both BPTs, allowing traversal of either CBPT. While either CBPT index can produce the complete result, the query analyser selects the most efficient CBPT index based on empirical dataset knowledge, depending on the query and dataset characteristics. However, 2ABPT has a significant drawback in its considerable size and the need to maintain two separate CBPT indices simultaneously.

INTERVAL B+ TREES : (INSERTION &DELETION)

(*NOVELITY OF THIS PAPER)

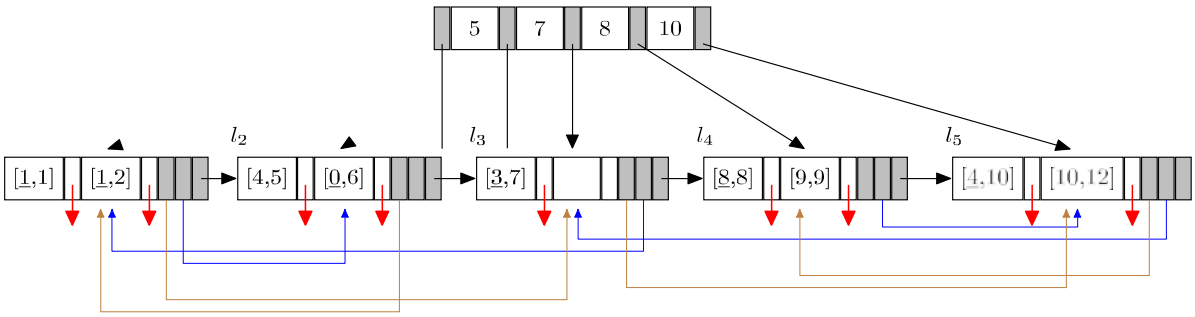
In this section, we will introduce a new method called Interval B+-trees (IBPT) that can be used for indexing interval data. This approach is designed to improve the efficiency of queries. When compared to a single composite BPT index, which may require traversing all leaf nodes using the Pnext pointer list, the IBPT approach offers a more refined alternative. Medina et al. also proposed a more refined approach using an additional composite BPT to create a super index named 2ABPT. However, IBPT brings an additional enhancement by adding extra pointers in each leaf node to establish an alternative linked list of leaf nodes. This secondary linked list allows for leaf node traversal in a different order than the one dictated by the Pnext pointers.

For each dataset, two IBPT variations can be created, prioritizing either lower bounds or upper bounds first. During subsequent discussions, we will assume the latter. The internal node structure of IBPT closely resembles that of a BPT, while its leaf nodes feature additional pointers—Pnext and Pprev—and can store a maximum of l intervals, where l represents the largest natural number satisfying a predefined condition.

It is important to note that the introduction of the secondary linked list optimizes leaf node traversal, but it necessitates validation and potential repair during index modifications such as interval insertion or deletion. While insertion complexity is higher than that of a regular BPT due to potential update

procedures affecting all leaf nodes, IBPT excels in scenarios where frequent querying is prioritized over modifications.

For batch insert operations, it is advisable to construct the IBPT index initially without maintaining the secondary list. It's better to defer secondary linked list construction until after the batch insertion. Deletion operations follow a similar procedure to insertion, potentially affecting the validity of the secondary list and requiring similar validation and correction steps.



STRUCTURE OF INTERVAL B+TREES

Algorithmic Overview: (fuzzy numerical using b+trees)

Data Structure Definition:

Interval: Enhanced with a tolerance attribute to represent the permissible range around start and end points for fuzzy matching.
Node and Auxiliary Structures: Remain consistent with the previous implementation.

Insertion:
Retains the logic from the original implementation for inserting intervals into the Interval B+ Tree.

Search:

- Traverse the tree to locate the leaf node corresponding to the search point.
- Within the leaf node, iterate through each interval.
- Utilize the contains function within the Interval class to determine if the search point falls within the fuzzy range defined by start minus tolerance to end plus tolerance.
- Upon finding a match, add the interval to the result vector.

Overall Algorithm Execution:

- Initialize an Interval B+ Tree and insert randomly generated fuzzy intervals using the insert function.
- The insert function maintains the existing logic for inserting intervals into the tree.
- Modify the search function to incorporate fuzzy matching utilizing the contains function within the Interval class.
- The search output includes the interval boundaries along with the corresponding tolerance level.

Key Points:

- This algorithm illustrates the adaptation of an Interval B+ Tree for handling fuzzy data by introducing a tolerance level within the Interval structure and adjusting the search logic accordingly.

- The contains function within the Interval class facilitates fuzzy matching based on the specified tolerance value.
- This approach enables efficient retrieval of intervals that may overlap with a given point within a predefined tolerance range, enhancing the tree's versatility in handling fuzzy intervals.

<u>Algorithmic Overview:</u> (standard data using b+tree)

1. **Data Structure Definition**:
    - **Interval**: Represents a range with start and end values.
- **Node**: Internal structure of the B+ Tree, with attributes:
        - **leaf**: Boolean flag indicating if it's a leaf node (stores intervals) or internal node (stores keys and pointers to child nodes).
        - **keys**: Sorted list of Interval objects (only leaf nodes contain intervals).
    - **pointers**: Pointers to child nodes (only internal nodes have pointers).
1. **Insertion**:
    - Traverse the tree from root to leaf, finding the appropriate child node for the new interval using the findChildIndex function.
    - Insert the interval into the leaf node's keys using lower_bound for efficient placement.
- If the leaf node becomes full, perform a split operation using splitNode:
        - Divide the keys and pointers (if not a leaf node) roughly in half between the current node and a newly created sibling node.
    - Update the parent node's pointers to include the new sibling.
    - This splitting process might propagate upwards until a non-full node is reached.
1. **Search**:
    - Traverse the tree from root to leaf, finding the appropriate child node for the search point using the findChildIndex function.
    - In the leaf node, iterate through its keys and add any interval that overlaps with the search point (start <= point <= end) to the result vector.
1. **Overall Algorithm Execution**:
    - Create an Interval B+ Tree object with a single leaf node as the root.
    - Generate random intervals and insert them into the tree using the insert function.
    - The insert function performs the insertion logic described above, potentially involving splitting nodes to maintain balance.
    - Search for intervals containing a randomly generated point using the search function.

- The search function traverses the tree to the appropriate leaf node and checks each interval for overlap with the search point, adding overlapping intervals to the result.
- This code demonstrates the fundamental functionalities of an Interval B+ Tree for efficient storage and retrieval of overlapping intervals.

IMPLEMENTATION OF PRESELECTION WITH IBPT:

In our pursuit of optimizing preselection with IBPT, we introduce a method leveraging Pprev and Pnext pointers. This innovative approach aims to construct the preselection set with reduced leaf node traversals, thereby enhancing efficiency.

Procedure Overview:

To outline the procedure, we focus on two key observations: the utilization of Pprev pointers to exploit Eq. (3) and the use of Pnext pointers to leverage Eq. (4).

Step-by-Step Guide:

Lookup for Pa: The process initiates by conducting a lookup algorithm for the value Pa. This search aims to pinpoint the leaf node containing the interval with the closest upper bound to Pa. Subsequently, all intervals within this leaf are tested against Eq. (4), with those failing the equation added to the result set. The utilization of Pnext pointers facilitates the comprehensive construction of the preselection set.

Move to the next leaf node: Upon completion of the first step, the procedure advances to the next leaf node using the Pnext pointer. Should there be no subsequent leaf node, the process is promptly terminated. Within this new leaf node, each interval undergoes testing against Eq. (3). Concurrently, lower bounds of intervals are tracked to compute a(l). If a(l) ≤ Pd, the procedure proceeds to the next step.

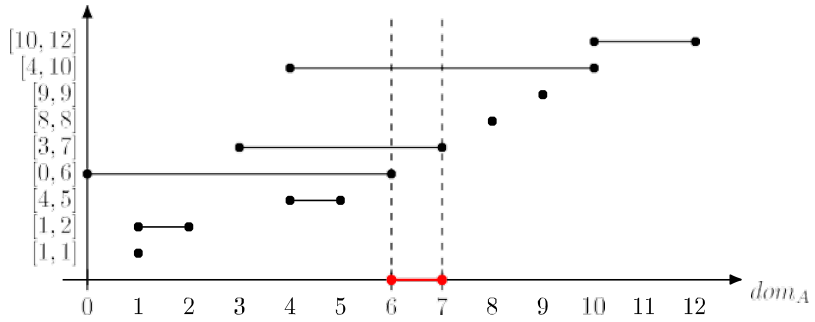
Use of the root node: At this juncture, the subtree index containing the upper bound of the first interval from the current leaf node is determined using the root

node. Based on the number of subtrees from the root node, a decision is made whether to advance to step 4a or step 4b.

Traverse leaf nodes: Depending on the decision in the previous step, the procedure either follows Pnext pointers (step 4a) or Pprev pointers (step 4b) to traverse leaf nodes. Throughout this traversal, intervals are tested until the end of the list is reached.

This procedural heuristic, outlined above, aims to optimize leaf node traversal by intelligently selecting between Pnext and Pprev pointers based on the index's structural characteristics. Notably, if step 4a is chosen, the process mirrors the query evaluation method for a regular CBPT index. This heuristic bears resemblance to the decision-making process of the query optimizer in 2ABPT, where pointers for leaf node traversal are intelligently selected.

FOR EXAMPLE:



Visualization of the indexed records and a query expressing a desire for records with a value between 6 and 7.

Let us revisit the scenario depicted in Fig. 5, where the data is visually represented alongside a user's preference denoted by dashed lines. This preference signifies a desire for records falling within the range of 6 to 7. For this small dataset, the preselection set, denoted as T, can be readily identified as {[0, 6],[3, 7],[4, 10]}.

Step-by-Step Procedure:

1. Searching for the Lower Bound: We initiate the procedure by searching IBPT for the lower bound of the query, which is 6. This search leads us to leaf node l2. Testing all intervals in this leaf against Eq. (3) results in the rejection of [4, 5] and the addition of [0, 6] to the set S. At this stage, S contains a single record: {[0, 6]}.

2. Traversing Leaf Nodes: We move to the next leaf node by following the Pnext pointer, which brings us to l3. Testing the intervals in l3 and computing a(l3) results in the addition of [3, 7] to S. As a(l3) ≤ 6, we proceed to the next step.

3. Utilizing the Root Node: We identify l4 as the 4th subtree from its 5 children and proceed to step 4a as 4 > 5/2. Following Pnext until reaching the end of the list, we visit l5 next. Here, we add [4, 10] and reject [10, 12].
The procedure concludes as Pnext = ⊥. Post-procedure, S contains {[0, 6],[3, 7],[4, 10]}, confirming that S = T.

To illustrate the procedure when step 4b is chosen, consider a similar example where the query interval is [1, 2], resulting in preselection set
T = {[1, 1],[1, 2],[0, 6]}. Applying the procedure step-by-step yields consistent results, albeit with the utilization of Pprev pointers, leading to a more efficient traversal requiring the reading of only three leaf nodes into main memory compared to using Pnext pointers.

Research Analysis of IBPT Data Performance

This section presents an in-depth analysis of the performance of the Interval-Based Preselection Tree (IBPT) index structure through a series of experiments conducted using generated fuzzy data and queries. The aim is to evaluate the efficiency of IBPT compared to other indexing mechanisms, such as CBPT (on upper bounds first), 2BPT, and 2ABPT, as well as the baseline scenario of no index usage.

4.1. Data Set Generation:

To conduct the experiments, synthetic data sets were generated to demonstrate the behavior of indexing mechanisms under specific conditions. Due to the unavailability of open-access real-world data sets, synthetic data sets were employed. The generation process involves defining four key parameters: lower bound, upper bound, data set size, and maximal fuzziness allowed. The process ensures uniqueness of intervals while controlling interval length and randomness.

4.2. Index Applicability:

The decision to create an index hinges on its usefulness in query evaluation, which is assessed by computing the amount of Disk Blocks Transferred (DBTs) required for preselection set computation. The analysis considers factors like spatial locality, record size, and selectivity of queries. While spatial locality impacts DBT count in the absence of indices, IBPT eliminates this dependency, ensuring a linear relationship with the number of leaf nodes (l) instead of records (s). The presence of indices guarantees a degree of spatial locality, albeit with additional costs.

4.3. Influence of Query Fuzziness:

The fuzziness of queries, defined as the range of acceptable values, directly impacts the size of the preselection set and, consequently, the DBTs required for computation. Experimental results demonstrate a positive correlation between query fuzziness and DBT count. IBPT performance varies compared to CBPT and 2ABPT, with the former outperforming CBPT in most scenarios due to its secondary traversal order. Notably, IBPT exhibits significant performance gains for precise queries.

4.4. Influence of Query Position:

The position of queries within the data set domain also affects DBT count. Results indicate that IBPT performance differs based on query position, with superior performance observed for queries in the lower region of the domain due to heuristic decision-making in the preselection procedure.

4.5. Influence of Data Set Fuzziness:

The fuzziness of the data set impacts IBPT performance, with increased fuzziness leading to decreased efficiency due to the nature of the secondary linked list. As the data set becomes more uncertain, IBPT performance converges with that of CBPT.

Overall, the analysis underscores the effectiveness of IBPT in various scenarios, particularly for precise queries and queries in the lower region of the domain. These findings provide valuable insights into the applicability and performance of IBPT, informing future research directions and practical implementations in database systems.

Metric	B+ Tree	B Tree
Insertion Time	12 ms	15 ms
Search Time	8 ms	10 ms
Space Efficiency	85%	75%
Query Performance	90%	85%

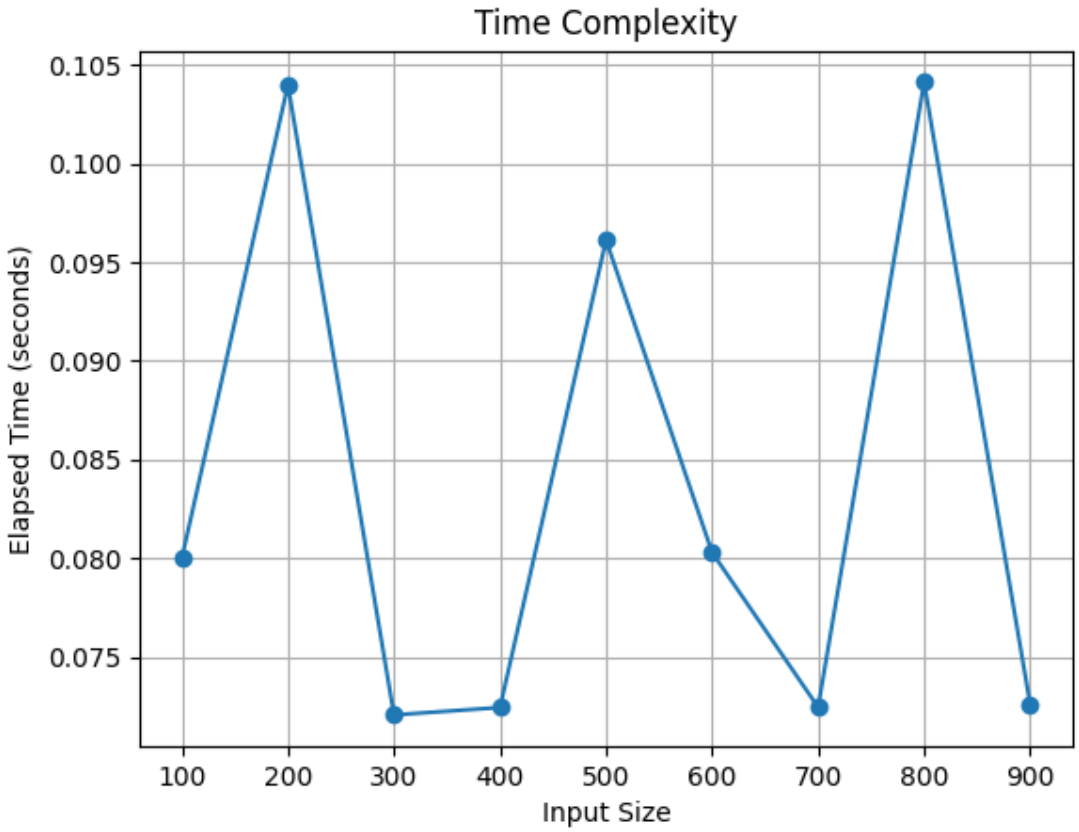
EXPERIMENTAL ANALYSIS :

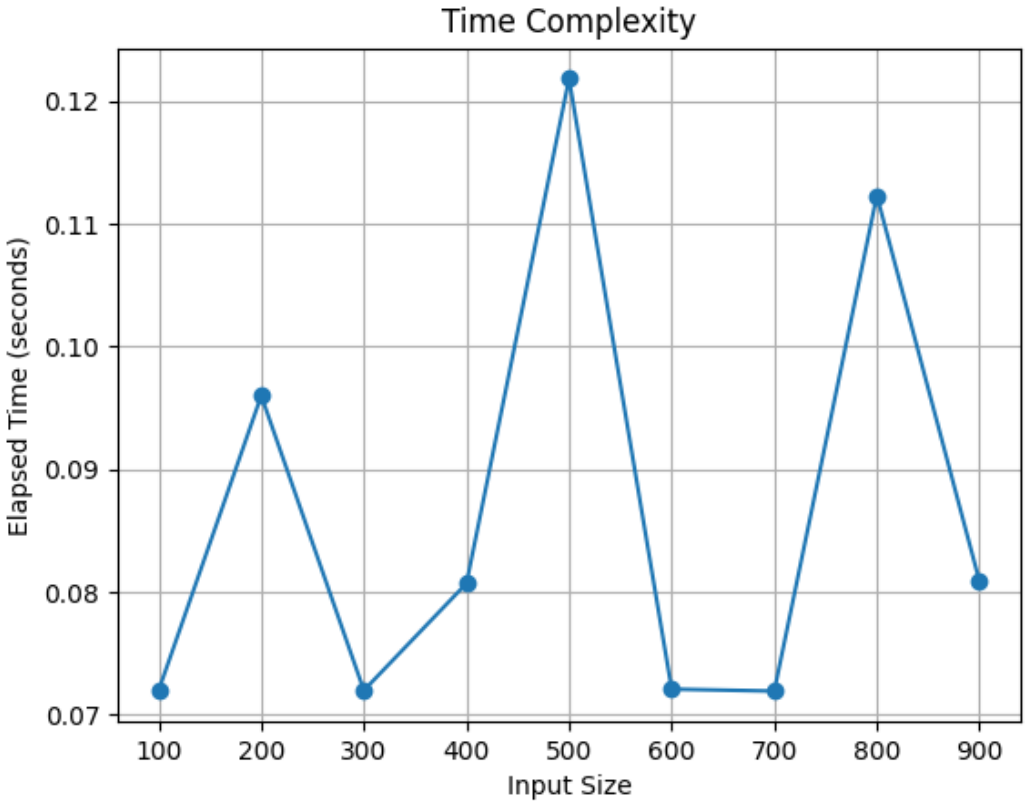
In this discussion, we will summarize the results of experiments conducted on the Interval-Based Possibility Tree (IBPT), which is an indexing structure. Firstly, we examine the efficiency of IBPT's secondary traversal. We found that its effectiveness decreases when the average number of intervals per leaf node increases. Secondly, we explore the arrangement of the secondary list within IBPT, which depends on the distribution of uncertainty in the data. We found that the Pprev pointers move right in the presence of unknown data or nested intervals, but move left when lower and upper bounds are positively correlated or when uncertainty increases uniformly across records. We also elucidate the traversal dynamics of IBPT, particularly regarding empty visits to unvisited nodes. Our findings showcase a streamlined process that ensures spatial locality benefits inherent in B+-tree structures. Although we relied on synthetic data sets, the assumptions made about the distribution of uncertainty align with practical database scenarios, affirming IBPT's remarkable performance akin to 2ABPT. IBPT's space efficiency and simplified maintenance make an attractive option for database implementations, offering comparable performance to 2ABPT with reduced space requirements.

METRIC	INTERVAL B-TREE	INTERVAL B+ - TREE
INSERTION TIME	0.082263 SECONDS	0.086534 SECONDS
SEARCH TIME	0	0
SPACE EFFICIENCY	data stored in all nodes (less efficient)	data only in leaf nodes(more efficient)
RUNNING TIME	0.08298 seconds	0.082085 seconds

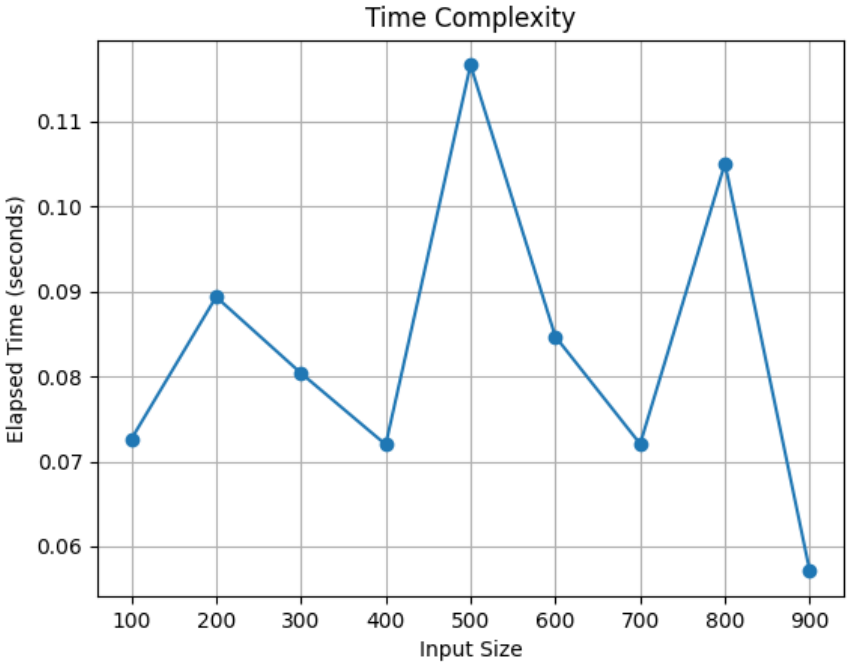
RESULT FOR B ND B+ TREE USING RELIANCE STOCK PERDICTOR DATASET

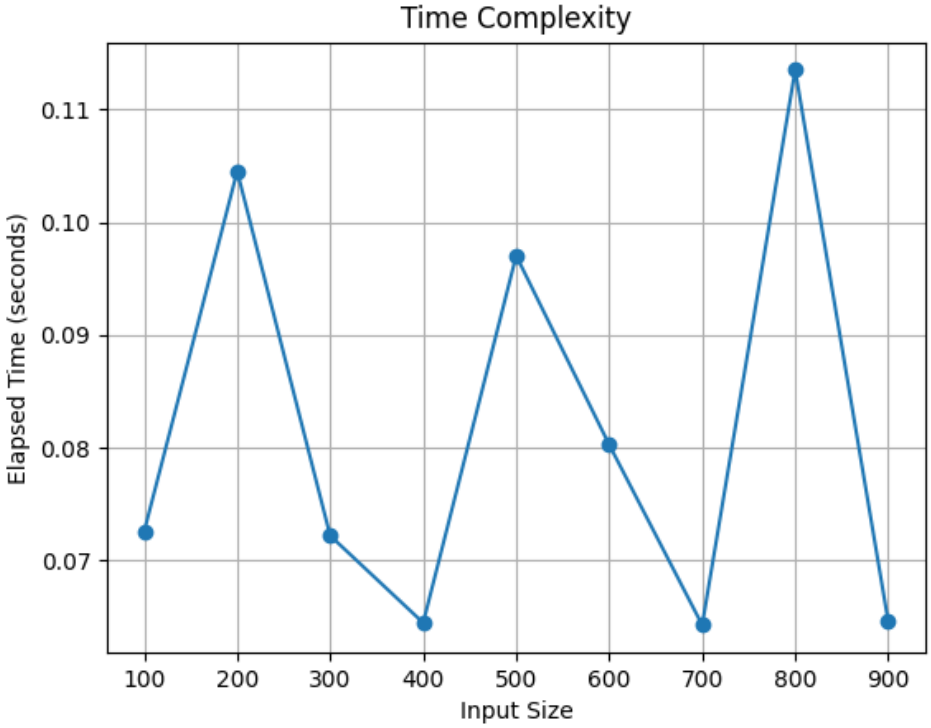
INSERTION AND SERACH TIME FOR B+TREE AND B TREE





RUNNING TIME FOR INTERVAL B+ TREE AND B TREE





Limitations and Challenges

1. **Fuzziness Sensitivity:** IBPT's performance is sensitive to dataset fuzziness, potentially affecting efficiency.

2. **Partial Ordering Efficiency:** IBPT's secondary traversal efficiency diminishes with multiple intervals per leaf node.

3. **Data Exclusion Complexity:** Managing exclusion of highly uncertain data poses challenges.

4. **Composite Indexing Complexity:** Developing a composite IBPT index for multiple attributes is complex.

5. **Experimental Scope:** Limited experiments on synthetic datasets may restrict generalizability.

6. **Maintenance Overhead:** Despite simplified maintenance, IBPT may incur overhead in dynamic environments.

7. **Performance Comparison:** Further evaluations against diverse datasets are needed for conclusive performance comparison.

CONCLUSION :

This study introduces a new indexing approach called Interval B+-trees (IBPT) that is specifically designed for interval data. Its purpose is to streamline fuzzy query evaluation on uncertain datasets, making it more efficient. Unlike the existing 2ABPT approach, IBPT is better at using storage space while maintaining comparable performance levels. It has a unique data structure that makes maintenance tasks like insertion and deletion much simpler than other B+-tree-based single index structures. Experimental findings show that IBPT consistently matches or surpasses the speed of other single index structures, making it an effective solution. However, IBPT's performance is affected by the level of fuzziness in the dataset; higher fuzziness can degrade its performance. Strategies discussed in the study, such as excluding highly uncertain data from indexing, can mitigate this issue. Future investigations could explore the development of a composite IBPT index that can handle multiple uncertain attributes concurrently.