Threshold Interval Indexing for Complicated Uncertain Data

Andrew Knight
Department of Computer Science
Rochester Institute of Technology
Rochester, New York, USA
Email: alk1234@rit.edu

Qi Yu
Department of Information Sciences and Technologies
Rochester Institute of Technology
Rochester, New York, USA
Email: qyu@it.rit.edu

Manjeet Rege
Department of Computer Science
Rochester Institute of Technology
Rochester, New York, USA
Email: mr@cs.rit.edu

Abstract—Uncertain data is an increasingly prevalent topic in database research, given the advance of instruments which inherently generate uncertainty in their data. In particular, the problem of indexing uncertain data for range queries has received considerable attention. To efficiently process range queries, existing approaches mainly focus on reducing the number of disk I/Os. However, due to the inherent complexity of uncertain data, processing a range query may involve high computational cost in addition to the I/O cost. In this paper, we present a novel indexing strategy focusing on one-dimensional uncertain continuous data, called threshold interval indexing. Threshold interval indexing is able to balance between I/O cost and computational cost to achieve an optimal overall query performance. A key ingredient of the proposed indexing structure is a dynamic interval tree. The dynamic interval tree is much more resistant to skew than R-trees, which are widely used in other indexing structures. We also present a more efficient version of our index, called the memory-loaded threshold interval index, which reduces the storage size so the primary tree can be loaded into memory. We perform experiments to demonstrate the effectiveness and efficiency of the proposed indexing strategy.

I. INTRODUCTION

The term uncertain data defines data collected with an inherent and distinctly quantifiable level of uncertainty [1]. Whereas values for certain data are given as exact constants, values for uncertain data are instead given by probability measures, most notably probability distribution functions (PDFs). Uncertain data can be generated from many applications, such as scientific measurements, sensor networks, GPS, and mobile object tracking.

The problem of indexing uncertain continuous data for efficiently processing range queries has received considerable attention in the database community. A set of key strategies have been proposed and the most representative ones include threshold index [2], [3], the U-tree [4], and 2D mapping techniques [2], [5]. Most of these approaches assume that disk I/Os are the dominating factor that determines the overall query performance. Thus, the indexing structures are usually designed to optimize the number of disk I/Os. However, uncertain data is inherently more complicated than certain data. Computing a range query on uncertain data usually involves complicated computations, which incur high CPU cost. This makes disk I/Os no longer the solely dominating factor that determines the overall query performance. Therefore, new indexing strategies need to be developed to optimize the overall performance of range queries on uncertain data.

Uncertain continuous data is usually modeled by PDFs. Some PDFs, like the uniform PDF, may be very simple to calculate. However, many widely used PDFs involve complicated computations, such as multimodal probability models for cluster analysis [6]. Computing a complicated PDF may involve high computational cost. Numerical approaches, such as Monte Carlo integration, have been exploited to improve the performance [4]. Riemann sum, however, provides a better strategy for one-dimensional cases. Even though Riemann sums can be faster than Monte Carlo integrations, they still incur high computational cost. As shown in Figure 1, their runtime is significant when compared to disk I/Os. For high accuracy, probability calculations take even longer than disk I/Os. Therefore, the number of probability calculations must also be considered if the distribution of the uncertain data is complicated. In this case, the indexing strategy needs to balance between disk I/Os and the CPU cost to achieve an optimal overall query performance.

In this paper, we present a novel indexing strategy focusing...
on one-dimensional uncertain continuous data, called threshold interval indexing. It addresses the limitations of existing indexing structures on uncertain data, particularly for handling complicated PDFs, by treating uncertain objects as intervals and thereby leveraging interval tree techniques. The proposed indexing structure is also inspired by the optimized interval techniques from [7], [8] to build a dynamic primary tree and store objects in nodes at different levels depending on the objects’ sizes. The notion of using an interval tree to index uncertain data was suggested by Cheng et al. in [2] but disregarded in favor of an R-tree with extra probability limits called x-bounds. We assert that x-bounds can just as easily be applied to interval trees to index uncertain data with special benefits.

The rest of the paper is organized as follows. Section II explains the motivation for developing our new strategy, given the problem statement and previous research. Section III presents the threshold interval index. Section IV improves the design of our index by reducing the storage size to reduce the number of disk I/Os and probability calculations. Section V gives experimental results of our two indexes versus the probability threshold index. Section VI concludes the paper by offering direction for future research.

II. MOTIVATION FROM PREVIOUS INDEXES

Existing indexes are inadequate for handling complicated uncertain continuous data. This section will first explain the common data model for handling uncertainty before explaining the shortcomings of existing structures. The model is known as the uncertain object model or the probabilistic uncertainty model [2], [5], [9].

A. Problem Statement

Given a database table $T$, a query interval $[a, b]$ for an uncertain attribute $e$, and a threshold probability $\tau$, a range query returns all uncertain objects $u_i$ from $T$ for which $Pr(u_i, e \in [a, b]) \geq \tau$.

B. External Interval Tree Index

Interval trees are not specifically designed for handling uncertain data, but one-dimensional uncertain objects may be treated as intervals by using their PDF endpoints. Arge et al. [7] propose two optimal external interval tree indexes. Both indexes use a primary tree for layout and secondary structures to store the objects at each node, but one has a dynamic primary tree instead of a static one. Stabbing queries are used to return results. However, the downfall of both interval indexes is that if many uncertainty intervals overlap with the query interval’s endpoints, then few objects are pruned from the search, and a lot of time is wasted in calculating probabilities.

C. Probability Threshold Index

The probability threshold index (PTI) [2] allows range queries to prune more branches from searching than interval indexes allow by using a one-dimensional R-tree as a base tree with stricter boundaries called x-bounds. The PTI has many advantages. It is an elegant solution, and it is fairly easy to implement. The tree is dynamic as well. All boundaries are calculated when objects are added. Multiple x-bounds can be stored in each node, so queries can choose the most appropriate bounds for its threshold. Required storage space for internal nodes is relatively small.

The PTI is not without weaknesses, however. The primary weakness pointed out by Cheng et al. is that differences in interval sizes will skew the balance of the tree [2]. Cheng et al. also do not provide an optimal rectangle layout strategy for the PTI’s base tree, the R-tree. The best strategy for any R-tree is to make MBRs as disjoint as possible. When MBRs overlap too much, extra disk I/Os and probability calculations must be performed because fewer nodes can be pruned. Adding new objects, especially objects of vastly different interval lengths, exacerbates overlap. Simply put, sloppy R-trees are inefficient, but optimal R-trees are very difficult to maintain.

When rectangles overlap, not all objects which fall completely within the query interval can be immediately accepted. Since MBRs might overlap, every node must be checked. There is no exclusivity between node intervals. Nodes may not be stored in any order if their intervals are stretched. Objects might appear in the overlapping portions of nodes, too. These compounding factors force probability calculations on all objects in each unpruned node. This wastes lots of time, especially when the query interval is much larger in size than most uncertainty intervals.

D. 2D Mapping Indexes

Cheng et al. first suggested 2D mapping techniques as an alternative to the PTI for uniform PDFs [2]. Agarwal et al. then expanded 2D mapping techniques to histogram PDFs [5]. Histogram PDFs can easily be transformed into linear piecewise threshold functions and stored as a set of line segments. The structures of the indexes presented in [5] manipulate the line segments. They are efficient for uniform and histogram PDFs, but they are inapplicable for more general PDFs. Furthermore, each index is rigidly based upon one threshold value; separate indexes must be constructed for additional thresholds. This is starkly different from the PTI, which can manage several threshold values in one structure.

III. THRESHOLD INTERVAL INDEXING

We now present the threshold interval index (TII). The TII is like a dynamic external interval tree with x-bounds borrowed from the PTI. This structure presents two key advantages. The first advantage is that the structure intrinsically and dynamically maintains balance all the time. The second advantage is that the interval-based structure makes all uncertain objects which fall entirely within the query interval easy to find and, therefore, possible to add to the results set without further calculation. The PTI does not allow this because its MBRs might overlap. Furthermore, adding x-bound avoids the interval index’s problem for when many uncertainty intervals overlap the query interval.
A. Structure

The TII has a primary tree to manage interval endpoints. It also has secondary structures at internal nodes of the primary tree to store objects. When an object is added to the index, the endpoints of its uncertainty interval are added to the primary tree. Then, the object itself is added to the secondary structures of the appropriate tree node. Each object is also assigned a unique id if it does not already have one. X-bounds are stored for each internal node.

1) Primary Tree: The primary tree is a weight-balanced B-tree with branching parameter \( r > 4 \) and leaf parameter \( k > 0 \). The weight of a node is the number of items (in this case, endpoints) below it. All leaves are on level 0. All endpoints are stored at the leaves, and internal nodes hold copied values of endpoints [7]. The weight-balanced B-tree provides an effective way to dynamically manage intervals and spread.

2) Secondary Structures: Each internal node \( v \) represents an interval \( I_v \), which spans all interval endpoints represented by children of \( v \). Thus, the \( c \) children of \( v \) (for \( \frac{1}{4} r \leq c \leq 4r \)) naturally partition \( I_v \) into subintervals called slabs [7]. Each slab is denoted by \( I_v \) (for \( 1 \leq i \leq c \)), and a contiguous region of slabs, such as \( I_{v_1}, I_{v_2}, I_{v_3}, \) is called a multislab [7]. All slab boundaries within \( I_v \) are stored in \( v \). Note that \( I_{v_i} \) is the interval for the child node \( v_i \).

An uncertain object is stored at \( v \) if its uncertainty interval falls entirely within \( I_v \) but overlaps one or more boundaries of any child node’s \( I_{v_i} \). (A leaf stores uncertain objects whose PDF endpoints are contained completely within the leaf’s interval endpoints.) Each object is stored at exactly one node in the tree, as shown in Figure 2. Let \( U_v \) denote the set of uncertain objects stored in \( v \). In the external dynamic interval index, these objects are stored in secondary structures called slab lists [7], partitioned by the slab boundaries. However, only two secondary structures are needed per node for the TII because range queries (described later in this section) work slightly differently than stabbing queries. The left endpoint list stores all uncertain objects in increasing order of their uncertainty intervals’ left endpoints. The right endpoint list stores all uncertain objects in increasing order of their uncertainty intervals’ right endpoints. This is drastically simpler than the optimal external interval tree, which requires a secondary structure for each multislab [7]. If the uncertain objects hold extra data or large PDFs, it might be advantageous to store only uncertainty interval boundary points and object references in the two lists. The actual objects can be stored in a third structure to avoid duplication.

3) Applying X-bounds: X-bounds were introduced as part of the probability threshold index [2] and can easily be applied to the TII.

Definition 1: An x-bound is a pair of values \((L_x, R_x)\) for a PDF \( f(x) \) with uncertainty interval \([L, R]\) such that

\[
x = \int_{L}^{R} f(x) dx = \int_{R}^{R_x} f(x) dx = \int_{L_x}^{R_x} f(x) dx
\]

Fig. 2: A node \( v \) with three child nodes. The dotted lines denote slab boundaries. Note how objects are only stored within intervals which can completely contain them.

\( L_x \) is the left x-bound, and \( R_x \) is the right x-bound. Note that \( x \) is a probability value, meaning \( 0 \leq x \leq 1 \). For example, if \( x = 0.25 \), then there is a 25% chance that the object’s value appears in the interval \([L, L_{0.25}]\). Furthermore, there would be a 25% chance it appears in \([R_{0.25}, R]\) and a 50% chance it appears in \([L_{0.25}, R_{0.25}]\).

The notion of x-bounds can be applied to tree nodes as well as to PDFs, as seen in Figure 3. The left x-bound for a node is the minimum left x-bound of all child nodes and objects, and the right x-bound is the maximum right x-bound of all child nodes and objects. Specifically, for a node \( v \), left and right x-bounds are calculated for \( I_v \). A child node’s x-bounds must be considered when calculating \( v \)'s x-bounds: a child node might have tighter x-bounds than any of the uncertain objects stored at \( v \). The interval \( I_v \) accounts for all uncertain objects stored at \( v \) and in any child nodes of \( v \), and so should the x-bounds. The x-bounds for \( v \)'s slabs are given by the x-bounds on \( v \)'s child nodes. All of \( v \)'s x-bounds are stored in \( v \)'s parent. In this way, the interval \( I_v \) is analogous to a minimum bounding rectangle in an R-tree, and intervals are tightened by x-bounds in the same way as MBRs are tightened in the PTI [2]. X-bounds for more than one probability \( x \) can be stored as well.

B. Range Query Evaluation

Evaluating range queries for objects in \([a, b]\) with a threshold \( \tau \) on the TII is like evaluating stabbing queries on a regular interval tree. Two stabs are executed for each endpoint of the query interval: a left stab and a right stab. The nature of the query forces these stabs to be performed slightly differently.
from how they are described in [7]. Once the stabs are made, a series of grabs can be performed for all objects in between. This is called the stab 'n grab search.

1) The Left Stab: The left stab is the most complicated part of the stab 'n grab search. The search starts at the root node and continues down one path through child nodes until it hits the leaf containing the closest x-bound to a within its boundaries. This leaf is called the left boundary leaf. X-bounds are used to prune this search. If a node’s right x-bound is less than a or if a node’s left x-bound is greater than b, then the node can be pruned, because the probability that any of its objects falls within the query interval must be less than the query’s threshold. Objects are checked at nodes along the stab to see if they belong to the result set.

Before moving to the next child node, the uncertain objects stored in secondary structures at the current node must be investigated, because their uncertainty intervals may overlap the query interval. If they overlap the query interval, then they might be valid query results. Between the secondary structures, only the right endpoint list is needed. A quick binary search can be performed to find which objects fall within the query interval. Any object whose right endpoint is less than a can be disregarded. Any object whose both endpoints are within the query interval is added to the result set automatically. Otherwise, a probability calculation must be performed using the object’s PDF to determine if it meets the threshold probability. This is shown in Figure 4. The same strategy applies for the left boundary leaf. All valid objects are added to the result set.

2) The Right Stab: The right stab is analogous to the left stab, except it searches with b instead of a. The leaf found at the bottom of the stab is called the right boundary leaf. X-bound pruning is performed for the rightmost child nodes, not the leftmost. The process for searching the secondary structures is the same as in the left stab, except “left” and “right” are switched wherever mentioned. Furthermore, nodes visited during the left stab can be skipped during the right stab, because the process for investigating uncertain objects accounts for both endpoints of the uncertainty interval. This is why references to visited nodes are stored during the left stab.

3) The Grabs: The two stabs find the two boundary leaves and some uncertain objects in the result set. The remaining objects to investigate reside in the nodes between the two boundary leaves. Thankfully, all objects in between can be added to the result set without any probability calculations. Remember, intervals for nodes on the same level do not overlap, so all objects stored at nodes between the boundary leaves must fall entirely within the query. The most effective way to grab all of these uncertain objects is to perform a post-order tree traversal starting at the left boundary leaf and ending at the right boundary leaf, skipping each node that has already been visited. No extra searching needs to be done on the secondary structures. Figure 5 illustrates a full stab 'n grab query.

4) Time Bounds: A range query can be answered within the following time bounds using the stab 'n grab search:

**Theorem 1:** Let I be a TII storing N uncertain objects, whose primary tree has branching parameter r and leaf parameter k. Assume any calculation on an uncertain object’s PDF takes O(d) time. A range query Q with query interval [a, b] and threshold τ can return all T uncertain objects stored in I which fall within the query interval with probability p ≥ τ in O(kdlog_r(N/k) + T/k) time.

**Proof:** The height of the primary tree is O(log_r(N/k)) [7]. If the number of child nodes of any internal node is O(a), then the total number of nodes in the tree is O(Σ_i=0^log_r(N/k)r^i) = O(r^log_r(N/k)) = O(N/k). Since the N uncertain objects are distributed relatively uniformly over the tree, each node stores O(N/(N/k)) = O(k) objects. A stab, either right or left, visits O(log_r(N/k)) nodes from root to boundary leaf and must visit all objects stored at a node in the worst case, calculating probabilities for each. Hence, the stabs are performed in O(kdlog_r(N/k)) time. The grabs are performed in O(T/k) time, because extra checking at each node in between the leaf boundaries is unnecessary. All nodes visited by the grabs are guaranteed to be valid results, so T is used instead of N for the time bound. Therefore, two stabs and all grabs can be performed in a combined time of O(kdlog_r(N/k) + T/k).

C. **Externalization**

The TII can easily be externalized by setting k and r for the primary tree appropriately, albeit differently than how described in [7]. Let B be the block size; specifically, the number of data units which can be stored in a block. For
IV. MEMORY-LOADED THRESHOLD INTERVAL INDEXING

Although the TII aptly balances the data, the primary tree uses a lot of storage space. For every object, the primary tree must also store two endpoints. This can severely inflate the number of blocks when many objects are indexed externally. In this section, we introduce the memory-loaded threshold interval index (MTII) as an alternative external TII to reduce the number of disk I/Os during range queries. Since the primary tree is significantly smaller, all nodes can be preloaded into memory before the query runs to improve runtime.

In the TII, every uncertain object’s endpoints are stored in the leaves. However, in the MTII, only those endpoints which form the slab boundaries, e.g., the minimum and maximum values for each leaf’s interval, must be stored. This means that a leaf stores only two endpoints instead of \(k\) endpoints. An internal node stores its own slab boundaries and pointers to child nodes. It does not need to store the slab boundaries or x-bounds for its child nodes.

For the primary tree, let \(r = 2\) and let \(k = \frac{1}{2}B\) to make it a binary tree. Storing the primary tree is trickier for the MTII, since one block will hold more than one nodes. For each node, a node id, left slab boundary, and right slab boundary must be stored, which requires only three units of data. Blocks store tree nodes in a top-down, breadth-first fashion: start at the root, and store each successive level of the tree left, ordering nodes least to greatest for their intervals. The ordering and slab boundaries will inherently denote parent-child relationships. For example, a root node may have the interval [0, 1000]. Its two children might have [0, 400] and [400, 1000]. When reading the nodes the jump from a right endpoint of 1000 to a left endpoint of 0 denotes a new level in the tree. Since the root contains all nodes between 0 and 1000, the two nodes read after the root must be its children.

Uncertain objects are stored in the same way as for the TII, using secondary structures: the left and right endpoint lists are stored externally in blocks. X-bounds for each primary tree node are stored in a copy tree. For each x-bound value, another primary tree structure is created as mentioned above, only instead of storing slab boundaries, it stores x-bound pairs as if they were slab boundaries. This x-bound tree is stored the same way as the primary tree. Multiple x-bound trees can be created, one for each \(x\) value. This way, a query can load the appropriate x-bound tree and not waste disk I/Os on unnecessary x-bound values. A query will only read blocks for the primary tree and one x-bound tree.

Range queries using the MTII are executed similarly as for the TII. First, the primary tree and appropriate x-bound tree must be preloaded. Note that these structures may remain in memory if multiple queries will be executed. Then the stab ‘n grab search is performed, just like for the TII.

V. EXPERIMENTAL RESULTS

This section presents an evaluation of our experimental results. We use the probability threshold index as a benchmark against which to test both the standard and memory-loaded threshold interval indexes. The PTI used is a “practical” PTI in that to objects were removed and replaced to introduce a small level of skew.

The purpose for testing these three indexes is to compare their range query performance. Performance is measured by three primary metrics: number of disk I/Os, number of probability calculations, and runtime (in milliseconds).

Four different performance tests are run. Each test builds the index from a common data set and runs range queries on each index. Descriptions are given in Table I. Datasets are generated synthetically. Uncertain objects contain two attributes: an id and a PDF. The PDF interval is determined randomly based on test parameters, given in Table II. X-bounds are calculated for the probability values \(\{0.1, 0.3, 0.5, 0.7, 0.9\}\) on each index. The block size is 4096 bytes.

Each test is run with two types of PDFs. Just like for previous tests against the PTI, one PDF used is a uniform PDF [2]. The second PDF is the multimodal Gaussian distribution with four peaks, which is significantly more complicated. Each PDF can be stored by left and right endpoints and can be stretched to the appropriate interval length. All probability calculations are performed by using Riemann sums. 1000 rectangles are used for each Riemann sum to keep the average error margin around 0.1%. Range queries must also be generated. For each test, 100 queries are generated. Each query interval is random within a given domain. Each query is run against each index, and results for all queries are tabulated aggregately. The probability threshold used is \(\tau = 0.3\).

The Same, Different, Dense, and Sparse tests all test the spread of uncertain objects. Figures 6 gives results for these tests. It is clear that object spread and size significantly affects performance. All indexes have worse performance for
TABLE II: Test Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Same</th>
<th>Different</th>
<th>Dense</th>
<th>Sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Objects</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>Min Object Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max Object Value</td>
<td>10000</td>
<td>10000</td>
<td>1000</td>
<td>1000000</td>
</tr>
<tr>
<td>Min PDF Length</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max PDF Length</td>
<td>100</td>
<td>500</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

This number is most staggering for sparse indexes: the TII and MTII make relatively no calculations. Overall, the total runtime favors threshold indexes for complicated probability functions, particularly the MTII.

The trends between uniform and multimodal PDFs are generally the same for disk I/Os and probability calculations. This is not too surprising, because PDF shape has only a small affect on index structure. The major difference is in total runtime, as seen in Figures 6e and 6f. Since the multimodal PDF is more complicated, calculations take longer. Thus, the margin by which the TII and MTII outperform the PTI is much larger for multimodal PDFs than for uniform PDFs.

VI. CONCLUSION

In this paper, we present threshold interval indexing, a new strategy for indexing complicated uncertain continuous data of one dimension. We present two structures: a standard threshold interval index and a more efficient memory-loaded variant. The key advantage of threshold interval indexing over existing strategies, such as the probability threshold index, is that it handles balance better with its intervals.

There is plenty more to research on uncertain data. Specifically, future research should focus more on exploring the effects of different types of PDFs on uncertain indexes. PDFs should not be overlooked, because they are inseparable from what makes the data uncertain. Perhaps certain rectangle management strategies could improve the performance of the PTI. Although our paper focuses on range queries, these indexes could also be used for joins. It is also important for uncertain data strategies to be incorporated into database management systems. Parallelization should also be explored further.

REFERENCES


